1. 两数之和

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* twoSum(int* nums, int numsSize, int target, int* returnSize){
    int i, j;
    int *ans;
    ans = (int*)malloc(sizeof(int) * 2);
    *returnSize = 2;

    for (i = 0; i < numsSize - 1; i++) {
        for (j = i + 1; j < numsSize; j++) {
            if (nums[i] + nums[j] == target) {
                ans[0] = i;
                ans[1] = j;
                break;
            }
        }
    }
    return ans;
}
```

2. 两数相加

给出两个 **非空** 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 **逆序** 的方式存储的，并且它们的每个节点只能存储 **一位** 数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例:

```
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

不要忘记最后进位的情况

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */


struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2){
    struct ListNode *head;
    struct ListNode *iter;
    struct ListNode *newNode;
    int carry = 0;
    int num;

    newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
    newNode->next = NULL;
    iter = newNode;
    head = newNode;

    while (l1 && l2) {
        num = l1->val + l2->val + carry;
        carry = num / 10;
        num = num % 10;
        newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
        newNode->val = num;
        newNode->next = NULL;
        iter->next = newNode;
        iter = newNode;
        l1 = l1->next;
        l2 = l2->next;
    }
    while (l1) {
        num = l1->val + carry;
        carry = num / 10;
        num = num % 10;
        newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
        newNode->val = num;
        newNode->next = NULL;
        iter->next = newNode;
        iter = newNode;
        l1 = l1->next;
    }
    while (l2) {
        num = l2->val + carry;
        carry = num / 10;
        num = num % 10;
        newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
        newNode->val = num;
        newNode->next = NULL;
        iter->next = newNode;
        iter = newNode;
        l2 = l2->next;
    }

    if (carry) {
        newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
        newNode->val = carry;
        newNode->next = NULL;
        iter->next = newNode;
        iter = newNode;
        carry = 0;
    }
    return head->next;
}
```

3. 无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

```
输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

示例 3:

```
输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

暴力方法，最后一个用例无法通过

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        dic = {}
        line = 0
        ans = 0
        for i in range(len(s)):
            dic = {}
            line = 0
            for j in range(i, len(s)):
                if s[j] not in dic:
                    dic[ s[j] ] = 1
                    line += 1
                else:
                    break
            ans = max(ans, line)
        return ans
```

在暴力方法的基础上优化代码，每次找到重复元素的时候，不是从头开始，而是计算一下需要删除的字符，直接接着开始，这样能减少不少不必要的操作。

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        dic = {}
        start = 0
        end = 0
        ans = 0
        for i in range(len(s)):
            if s[i] not in dic:
                dic[ s[i] ] = i
                ans = max(ans, i - start + 1)
            else:
                end = dic[ s[i] ]
                for j in range(start, end+1):
                    del dic[ s[j] ]
                start = end + 1
                dic[ s[i] ] = i
                ans = max(ans, i - end)

        return ans
```

如果使用C语言，可以假设字母都是ASCII能够表示的范围，这样定义一个256长度的数组足够用来当哈希表记录出现的字符。需要注意，初始化为-1，因为索引从0开始。

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define C 256
int map[C];

void init()
{
    int i;

    for (i = 0; i < C; i++) {
        map[i] = -1;
    }
}
int lengthOfLongestSubstring(char * s)
{
    int start = 0;
    int end = 0;
    int ans = 0;
    int i, j;

    if (s == NULL) {
        return ans;
    }

    init();
    for (i = 0; i < strlen(s); i++) {
        if (map[ s[i] ] == -1) {
            map[ s[i] ] = i;
            ans = MAX(ans, i - start + 1);
        } else {
            end = map[ s[i] ];
            for (j = start; j <= end; j++) {
                map[ s[j] ] = -1;
            }
            start = end + 1;
            map[ s[i] ] = i;
        }
    }
    return ans;
}
```

5. 最长回文子串

给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

示例 1：

```
输入："babad"
输出："bab"
注意："aba" 也是一个有效答案。
```

示例 2：

```
输入："cbbd"
输出："bb"
```

暴力方法，无法通过最后6个用例 97 / 103 个通过测试用例

```
#define TRUE 1
#define FALSE 0
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int isH(char *s, int left, int right)
{
    while (left < right) {
        if (s[left] != s[right]) {
            return FALSE;
        }
        left += 1;
        right -= 1;
    }
    return TRUE;
}
char * longestPalindrome(char * s)
{
    int maxLen = 0;
    int start = 0;
    int end = 0;
    int i, j;
    char *ans;

    if (s == NULL) {
        return NULL;
    }

    if (strlen(s) == 0) {
        ans = (char*)malloc(sizeof(char));
        ans[0] = '\0';
        return ans;
    }

    for (i = 0; i < strlen(s) - 1; i++) {
        for (j = strlen(s) - 1; j > i; j--) {
            if (isH(s, i, j)) {
                if (j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    start = i;
                    end = j;
                }
            }
        }
    }

    ans = (char*)malloc(sizeof(char) * (end - start + 2));
    int ansEnd = 0;
    for (i = start; i <= end; i++) {
        ans[ansEnd++] = s[i];
    }
    ans[ansEnd] = '\0';
    return ans;
}
```

加入适当的剪纸操作，可以在规定的时间内通过。

```
#define TRUE 1
#define FALSE 0
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int isH(char *s, int left, int right)
{
    while (left < right) {
        if (s[left] != s[right]) {
            return FALSE;
        }
        left += 1;
        right -= 1;
    }
    return TRUE;
}
char * longestPalindrome(char * s)
{
    int maxLen = 0;
    int start = 0;
    int end = 0;
    int i, j;
    char *ans;

    if (s == NULL) {
        return NULL;
    }

    if (strlen(s) == 0) {
        ans = (char*)malloc(sizeof(char));
        ans[0] = '\0';
        return ans;
    }

    for (i = 0; i < strlen(s) - 1; i++) {
        for (j = strlen(s) - 1; j > i; j--) {
            if (isH(s, i, j)) {
                if (j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    start = i;
                    end = j;
                }
                /* 因为是从后向前扫描，如果找到回文，再向前就没有意义，这里直接结束本次循环，将i移动一位，继续寻找下一个可能
的回文 */
                break;
            }
        }
    }

    ans = (char*)malloc(sizeof(char) * (end - start + 2));
    int ansEnd = 0;
    for (i = start; i <= end; i++) {
        ans[ansEnd++] = s[i];
    }
    ans[ansEnd] = '\0';
    return ans;
}
```

6. Z 字形变换

将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "LEETCODEISHIRING" 行数为 3 时，排列如下：

```
L   C   I   R
E T O E S I I G
E   D   H   N
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如："LCIRETOESIIGEDHN"。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1:

```
输入: s = "LEETCODEISHIRING", numRows = 3
输出: "LCIRETOESIIGEDHN"
```

示例 2:

```
输入: s = "LEETCODEISHIRING", numRows = 4
输出: "LDREOEIIECIHNTSG"
解释:

L     D     R
E   O E   I I
E C   I H   N
T     S     G
```

方案一： 模拟整个过程，使用一个矩阵存放，使用\0填充空白的地方。 优点是容易想到，不易出错，缺点是占用空间多，代码长。

```
char * convert(char * s, int numRows){
    if (s == NULL || numRows == 0) {
        return NULL;
    }

    if (strlen(s) == 0 || numRows == 1) {
        return s;
    }

    int col;
    int subrow = numRows;
    int subcol = numRows - 1;
    int sublen = subrow + subcol - 1;
    int nsub = strlen(s) / sublen + 1;
    char **matrix;
    int i, j;
    int sindex = 0;
    int slen = strlen(s);
    int zflag = 0;

    col = nsub * subcol;
    /* 申请 numrows 行 col 列临时存储空间存放模拟过程 */
    matrix = (char**)malloc(sizeof(char*) * numRows);
    for (i = 0; i < numRows; i++) {
        matrix[i] = (char*)malloc(sizeof(char) * col);
        for (j = 0; j < col; j++) {
            matrix[i][j] = '\0';
        }
    }

    /* 模拟之字过程存放字符串 */
    i = 0;
    j = 0;
    while (sindex < slen) {
        matrix[i][j] = s[sindex++];
        if (i == 0) {
            i = i + 1;
            zflag = 0;
        } else if (i == numRows - 1) {
            i = i - 1;
            j = j + 1;
            zflag = 1;
        } else {
            if (zflag) {
                i = i - 1;
                j = j + 1;
            } else {
                i = i + 1;
            }
        }
    }

    /* 按照行遍历取出字符串 */
    char *ans = (char*)malloc(sizeof(char) * (slen + 1));
    int ansEnd = 0;
    for (i = 0; i < numRows; i++) {
        for (j = 0; j < col; j++) {
            if (matrix[i][j] != '\0') {
                ans[ansEnd++] = matrix[i][j];
            }
        }
    }
    ans[ansEnd] = '\0';
    return ans;
}
```

方案二： 不模拟过程，而是计算下标，直接输出。 第一行和最后一行容易确定，输出的都是间隔 2 * n - 2 的元素，如果当前输出s[i], 那么下一个字母应该输出s[i + 2*n-2].

中间的行i, 距离顶部的距离是i + 1, 距离底部的距离是numRows - i, 都包括自身。把该值记为p, 则任意元素的下一个元素只与向下还是向上有关，我们设置一个标记 down = true, 表示开始的时候是向下的。

对于任意一行i, 起始输出元素是s[i], 下一个输出的元素下标可以计算得到： 向下：i + 2 *p - 2, down = false 向上 : i + 2 *p - 2, down = true 循环上面的直到下标超出字符串的长度。

```
char * convert(char * s, int numRows){
    char *ans;
    int ansEnd = 0;
    int i = 0;
    int slen;
    int sindex = 0;
    int down = 1;
    int step;

    if (s == NULL || numRows == 0) {
        return NULL;
    }

    slen = strlen(s);
    if (slen == 0 || numRows == 1 || numRows >= slen) {
        return s;
    }

    ans = (char*)malloc(sizeof(char) *(strlen(s) + 1));
    for (i = 0; i < numRows; i++) {
        ans[ansEnd++] = s[i];
        sindex = i;
        down = 1;
        while (sindex < slen) {
            if (down) {
                step = 2 * (numRows - i) - 2;
                down = 0;
            } else {
                step = 2 * (i + 1) - 2;
                down = 1;
            }
            /* 第一行和最后一行的时候, step 可能为0 */
            sindex += step;
            if (sindex < slen && step > 0) {
                ans[ansEnd++] = s[sindex];
            }
        }
    }
    ans[ansEnd] = '\0';
    return ans;
}
```

7. 整数反转

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1:

```
输入: 123
输出: 321
```

示例 2:

```
输入: -123
输出: -321
```

示例 3:

```
输入: 120
输出: 21
```

注意:

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为 $[-2^{31}, 2^{31} - 1]$。请根据这个假设，如果反转后整数溢出那么就返回 0。

```
int reverse(int x){
    long long lx = x;
    long long ans = 0;

    while (x) {
        ans = ans * 10 + (x % 10);
        x = x / 10;
    }

    if (ans > INT_MAX || ans < INT_MIN) {
        return 0;
    }

    return (int)ans;
}
```

8. 字符串转换整数 (atoi)

请你来实现一个 `atoi` 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。接下来的转化规则如下：

- 如果第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字字符组合起来，形成一个有符号整数。
- 假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成一个整数。
- 该字符串在有效的整数部分之后也可能会存在多余的字符，那么这些字符可以被忽略，它们对函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换，即无法进行有效转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0 。

提示：

- 本题中的空白字符只包括空格字符 `' '` 。
- 假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 $[-2^{31}, 2^{31} - 1]$。如果数值超过这个范围，请返回 INT_MAX $(2^{31} - 1)$ 或 INT_MIN (-2^{31}) 。

示例 1:

```
输入："42"
输出：42
```

示例 2:

```
输入："   -42"
输出：-42
解释：第一个非空白字符为 '-', 它是一个负号。
     我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。
```

示例 3:

```
输入："4193 with words"
输出：4193
解释：转换截止于数字 '3' ，因为它的下一个字符不为数字。
```

示例 4:

```
输入："words and 987"
输出：0
解释：第一个非空字符是 'w', 但它不是数字或正、负号。
     因此无法执行有效的转换。
```

示例 5:

```
输入："-91283472332"
输出：-2147483648
解释：数字 "-91283472332" 超过 32 位有符号整数范围。
     因此返回 INT_MIN (−2^31) 。
```

按照题目的要求，仔细做好每一步的检查。

```
int myAtoi(char * str){
    int negtive = 0;
    int maxvalue = 0;
    int start, end, i, tmpstart;
    long long tmpans;

    if (str == NULL) {
        return 0;
    }

    int len = strlen(str);
    if (len == 0) {
        return 0;
    }

    /* 跳过开头的空格 */
    start = 0;
    while (start < len && str[start] == ' ') {
        start += 1;
    }

    if (start >= len) {
        return 0;
    }

    /* 第一个非空字符只能是+，-，数字 */
    if (str[start] != '+' && str[start] != '-' &&
            (!(str[start] >= '0' && str[start] <= '9'))) {
        return 0;
    }

    if (str[start] == '-') {
        negtive = 1;
        start += 1;
    } else if (str[start] == '+') {
        negtive = 0;
        start += 1;
    }

    /* 如果正负号后面不是紧跟数字，转换失败 */
    if (start >= len || !(str[start] >= '0' && str[start] <= '9')) {
        return 0;
    }

    /* end 指向最后一个数字之后的位置 */
    for (end = start; end < len; end++) {
        if (!(str[end] >= '0' && str[end] <= '9')) {
            break;
        }
    }

    /* 略过前导0找到第一个数字，先通过位数判断是否超过范围 */
    tmpstart = start;
    while (tmpstart < len && str[tmpstart] == '0') {
        tmpstart += 1;
    }
    if (end - tmpstart > 10) {
        maxvalue = 1;
    } else {
        tmpans = 0;
        for (i = start; i < end; i++) {
            tmpans = tmpans * 10 + (str[i] - '0');
        }
        if (negtive) {
            tmpans = -tmpans;
        }
        if (tmpans > INT_MAX || tmpans < INT_MIN) {
            maxvalue = 1;
        }
    }

    /* 如果超过最大值，返回最值，否则，返回真实值 */
    if (maxvalue) {
        if (negtive) {
```

```
            return INT_MIN;
        } else {
            return INT_MAX;
        }
    } else {
        return (int)tmpans;
    }
}
```

9. 回文数

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

示例 1:

```
输入：121
输出：true
```

示例 2:

```
输入：-121
输出：false
解释：从左向右读，为 -121 。 从右向左读，为 121- 。因此它不是一个回文数。
```

示例 3:

```
输入：10
输出：false
解释：从右向左读，为 01 。因此它不是一个回文数。
```

进阶:

你能不将整数转为字符串来解决这个问题吗?

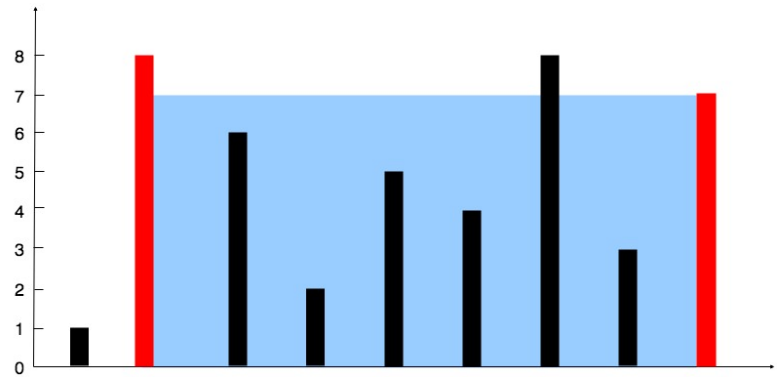
翻转之后可能超过 int 的表示范围，用 long long 类型

```
bool isPalindrome(int x){
    long lx = 0;
    long sx = x;

    if (x < 0) {
        return false;
    }

    while (x) {
        lx = lx * 10 + (x % 10);
        x = x / 10;
    }

    if (lx == sx) {
        return true;
    }

    return false;
}
```

11. 盛最多水的容器

给你 n 个非负整数 a_1，a_2，...，a_n，每个数代表坐标中的一个点 (i, a_i) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例：

```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
```

暴力方法，遍历每一种情况； 贪心方法，设置两个指针，每次移动较小的那个指针，遍历一遍即可。

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) > (b) ? (b) : (a))
int maxArea(int* height, int heightSize){
    int left = 0;
    int right = heightSize - 1;
    int area = 0;
    int tmparea;

    /* 贪心算法，每次比较两个指针大小，较小的移动 */
    while (left < right) {
        tmparea = MIN(height[left], height[right]) * (right - left);
        area = MAX(tmparea, area);
        if (height[left] < height[right]) {
            left += 1;
        } else {
            right -= 1;
        }
    }
    return area;
}
```

12. 整数转罗马数字

罗马数字包含以下七种字符： `I`， `V`， `X`， `L`， `C`， `D` 和 `M`。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如， 罗马数字 2 写做 `II` ，即为两个并列的 1。12 写做 `XII` ，即为 `X + II` 。 27 写做 `XXVII`, 即为 `XX + V + II` 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 `IIII`，而是 `IV`。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 `IX`。这个特殊的规则只适用于以下六种情况：

- `I` 可以放在 `V` (5) 和 `X` (10) 的左边，来表示 4 和 9。
- `X` 可以放在 `L` (50) 和 `C` (100) 的左边，来表示 40 和 90。
- `C` 可以放在 `D` (500) 和 `M` (1000) 的左边，来表示 400 和 900。

给定一个整数，将其转为罗马数字。输入确保在 1 到 3999 的范围内。

示例 1:

```
输入: 3
输出: "III"
```

示例 2:

```
输入: 4
输出: "IV"
```

示例 3:

```
输入: 9
输出: "IX"
```

示例 4:

```
输入: 58
输出: "LVIII"
解释: L = 50, V = 5, III = 3.
```

示例 5:

```
输入: 1994
输出: "MCMXCIV"
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

```
#define N 200
int base[13] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
char label[27] = "M0CMD0CDC0XCL0XLX0IXV0IVI0";

char * intToRoman(int num){
    char *ans;
    int count, i;
    int end = 0;
    int label_end = 0;
    ans = (char*)malloc(sizeof(char) * N);
    for (i = 0; i < 13; i++) {
        count = num / base[i];
        if (count > 0) {
            while (count--) {
                ans[end++] = label[label_end++];
                if (label[label_end] != '0') {
                    ans[end++] = label[label_end++];
                } else {
                    label_end++;
                }
                label_end = label_end - 2;
            }
            num = num % base[i];
        }
        label_end += 2;
    }
    ans[end] = '\0';
    return ans;
}
```

14. 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 `""`。

示例 1:

```
输入: ["flower","flow","flight"]
输出: "fl"
```

示例 2:

```
输入: ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。
```

说明:

所有输入只包含小写字母 `a-z` 。

```
#define TRUE 1
#define FALSE 0

int notSame(char **strs, int strsSize, int index)
{
    int i;
    for (i = 0; i < strsSize; i++) {
        if (index >= strlen(strs[i])) {
            return TRUE;
        }
        if (i != strsSize - 1 && strs[i][index] != strs[i + 1][index]) {
            return TRUE;
        }
    }
    return FALSE;
}
char * longestCommonPrefix(char ** strs, int strsSize){
    int index = 0;
    char *ans;
    int ansEnd = 0;
    int i;
    int maxlen;

    if (strs == NULL || strsSize == 0) {
        ans = (char*)malloc(sizeof(char));
        ans[0] = '\0';
        return ans;
    }

    /* 获取字符串中最长的那一个 */
    maxlen = 0;
    for (i = 0; i < strsSize; i++) {
        if (strlen(strs[i]) > maxlen) {
            maxlen = strlen(strs[i]);
        }
    }
    for (i = 0; i < maxlen; i++) {
        if (notSame(strs, strsSize, index)) {
            break;
        }
        index += 1;
    }
    ans = (char*)malloc(sizeof(char) * (index + 1));
    for (i = 0; i < index; i++) {
        ans[ansEnd++] = strs[0][i];
    }
    ans[ansEnd] = '\0';
    return ans;
}
```

---

15. 三数之和

给你一个包含 n 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 $a，b，c$，使得 $a + b + c = 0$？请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

```
给定数组 nums = [-1, 0, 1, 2, -1, -4],

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

使用双指针法，固定一个，移动另外两个指针

```
class Solution(object):
    def threeSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        sorted_nums = sorted(nums)
        i = 0
        n = len(nums)
        ans = []
        while i < n - 2:
            # 跳过重复元素
            if i > 0 and sorted_nums[i] == sorted_nums[i-1]:
                i += 1
                continue

            j = i + 1
            k = n - 1
            while j < k:
                tmp = sorted_nums[j] + sorted_nums[k]
                if tmp == -sorted_nums[i]:
                    # 跳过重复元素
                    if j == i + 1 or j > i + 1 and sorted_nums[j] != sorted_nums[j-1]:
                        ans.append( [sorted_nums[i], sorted_nums[j], sorted_nums[k]] )
                    j += 1

                elif tmp < -sorted_nums[i]:
                    j += 1
                else:
                    k -= 1
            i += 1
        return ans
```

C 语言的实现

```c
#define MAXANS 1000
#define TRUE 1
#define FALSE 0

int compare(const void *p, const void *q)
{
    int a = *(int *)p;
    int b = *(int *)q;

    if (a < b) {
        return -1;
    } else if (a > b) {
        return 1;
    }

    return 0;
}
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller calls free().
 */
int** threeSum(int* nums, int numsSize, int* returnSize, int** returnColumnSizes){
    if (nums == NULL || numsSize <= 0) {
        return NULL;
    }

    int **ans;
    int ansEnd = 0;
    int *returnColumnSizesTmp;
    int returnColumnSizesTmpEnd = 0;
    int i, j, k, tmp;

    ans = (int**)malloc(sizeof(int*) * MAXANS);
    returnColumnSizesTmp = (int*)malloc(sizeof(int) * MAXANS);
    for (i = 0; i < MAXANS; i++) {
        ans[i] = (int*)malloc(sizeof(int) * 3);
        returnColumnSizesTmp[i] = 3;
    }

    qsort(nums, numsSize, sizeof(int), compare);

    for (i = 0; i < numsSize - 2; i++) {
        /* 如果和前面的一样，说明是重复答案 */
        if (i > 0 && nums[i] == nums[i - 1]) {
            continue;
        }
        j = i + 1;
        k = numsSize - 1;
        printf("%d %d, ", j, k);
        while (j < k) {
            tmp = nums[j] + nums[k];
            if (tmp + nums[i] == 0) {
                /* 如果是重复答案，不记录 */
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    j += 1;
                    continue;
                }
                ans[ansEnd][0] = i;
                ans[ansEnd][1] = j;
                ans[ansEnd][2] = k;
                ansEnd += 1;
                j += 1;
            } else if (tmp > -nums[i]) {
                k -= 1;
            } else {
                j += 1;
            }
        }
    }

    *returnSize = ansEnd;
    return ans;
}
```

18. 四数之和

给定一个包含 n 个整数的数组 `nums` 和一个目标值 `target`，判断 `nums` 中是否存在四个元素 a，b，c 和 d，使得 $a + b + c + d$ 的值与 `target` 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

示例：

```
给定数组 nums = [1, 0, -1, 0, -2, 2]，和 target = 0。

满足要求的四元组集合为：
[
  [-1,  0, 0, 1],
  [-2, -1, 1, 2],
  [-2,  0, 0, 2]
]
```

固定两个数字，另外两个数字使用双指针逼近

```
class Solution(object):
    def fourSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        sns = sorted(nums)
        n = len(sns)
        if n < 4:
            return []

        i = 0
        ans = []
        while i < n - 3:
            if i > 0 and sns[i] == sns[i-1]:
                i += 1
                continue

            j = i + 1
            while j < n - 2:
                if j > i+1 and sns[j] == sns[j-1]:
                    j += 1
                    continue

                p = j + 1
                q = n - 1
                while p < q:
                    tmp = sns[p] + sns[q] + sns[i] + sns[j]
                    if tmp == target:
                        if p == j + 1 or p > j + 1 and sns[p] != sns[p-1]:
                            ans.append([sns[i],sns[j],sns[p], sns[q]])
                        p += 1
                    elif tmp < target:
                        p += 1
                    else:
                        q -= 1
                j += 1
            i += 1
        return ans
```

19. 删除链表的倒数第N个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

```
给定一个链表：1->2->3->4->5，和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 *n* 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

删除链表的导数第N个节点

常规解法

假设链表为link, 节点个数为N，倒数第n个节点，就是正数第(N-n+1)个节点，删除该节点，需要找到它前面的节点，也就是第(N-n)个节点。 如果(N-n)==0, 直接用头结点指向第二个节点，即删除了第一个节点； 其他情况，找到第(N-n)节点，next指针指向下下个节点；

一次遍历的解法

设置两个指针，间隔n, 然后一起移动两个指针，前面的到结尾，后面的指针正好指向倒数第n个节点前一个。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */


struct ListNode* removeNthFromEnd(struct ListNode* head, int n){
    struct ListNode *g;
    struct ListNode *c;
    struct ListNode *p;

    g = (struct ListNode*)malloc(sizeof(struct ListNode));
    g->next = head;

    if (head == NULL) {
        return head;
    }

    // 提前走n+1步
    n = n + 1;
    c = g;
    while (c && n) {
        c = c->next;
        n = n - 1;
    }

    // 一起走知道c为空
    p = g;
    while (c) {
        c = c->next;
        p = p->next;
    }

    // p的下一个结点为要删除的节点
    p->next = p->next->next;
    return g->next;
}
```

---

25. K 个一组翻转链表

给你一个链表，每 *k* 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例：

给你这个链表：`1->2->3->4->5`

当 $k = 2$ 时，应当返回：`2->1->4->3->5`

当 $k = 3$ 时，应当返回：`3->2->1->4->5`

说明：

- 你的算法只能使用常数的额外空间。
- **你不能只是单纯的改变节点内部的值**，而是需要实际进行节点交换。

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */


struct ListNode* reverseKGroup(struct ListNode* head, int k){
    int tmpK;
    struct ListNode *start;
    struct ListNode *end;
    struct ListNode *preStart;
    struct ListNode *endNext;
    struct ListNode *iter1;
    struct ListNode *iter2;
    struct ListNode *preHead;
    // k == 1, 不用翻转
    // 没有结点，不用翻转
    if (k == 1 || head == NULL) {
        return head;
    }

    // 创建一个指向第一个结点的结点
    preHead = (struct ListNode*)malloc(sizeof(struct ListNode));
    preHead->next = head;

    preStart = preHead;
    endNext = preHead;
    while (1) {
        // 查找从start开始的第K个结点
        tmpK = k + 1;
        while (endNext && tmpK--) {
            endNext = endNext->next;
        }
        if (tmpK > 0) {
            break; // 不足k个结点退出
        }

        // 翻转K个结点
        start = preStart->next;
        iter1 = start;
        end = start->next;
        while (end && end != endNext) {
            iter2 = end->next;
            end->next = start;
            start = end;
            end = iter2;
        }
        // 连接翻转之后的链表和其前后结点
        preStart->next = start;
        iter1->next = endNext;

        // 更新循环变量
        preStart = iter1;
        endNext = iter1;
    }
    return preHead->next;
}
```

30. 串联所有单词的子串

给定一个字符串 **s** 和一些长度相同的单词 **words**。找出 **s** 中恰好可以由 **words** 中所有单词串联形成的子串的起始位置。

注意子串要与 **words** 中的单词完全匹配，中间不能有其他字符，但不需要考虑 **words** 中单词串联的顺序。

示例 1：

输入：
  s = "barfoothefoobarman",
  words = ["foo","bar"]
输出：[0,9]
解释：
从索引 0 和 9 开始的子串分别是 "barfoo" 和 "foobar" 。
输出的顺序不重要, [9,0] 也是有效答案。

示例 2:

输入：
  s = "wordgoodgoodgoodbestword",
  words = ["word","good","best","word"]
输出：[]

```
from collections import Counter

class Solution(object):
    def findSubstring(self, s, words):
        """
        :type s: str
        :type words: List[str]
        :rtype: List[int]
        """
        if not s or not words:
            return []

        n = len(s)
        m = len(words)
        k = len(words[0])

        # 每次取 k * m 个字符计算k个字符组成的单词的出现次数
        # 处理 words 为每个单词出现的次数
        # 比较两个结果是否相同
        wordsCount = Counter(words)
        ans = []
        for i in range(n - m * k + 1):
            tmpstr = s[i : i + k * m]
            tmpwords = []
            for j in range(0, m * k, k):
                tmpwords.append( tmpstr[j:j+k])
            a = Counter(tmpwords)
            if a == wordsCount:
                ans.append(i)
        return ans
```

---

33. 搜索旋转排序数组

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 $O(\log n)$ 级别。

示例 1:

输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4

示例 2:

输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1

```
int search(int* nums, int numsSize, int target){
    int i, j, mid;
    i = 0;
    j = numsSize - 1;
    while (i <= j) {
        mid = (i + j) / 2;
        if (target == nums[mid]) {
            return mid;
        } else if (target > nums[mid]) {
            if (nums[mid] <= nums[j] && nums[j] < target) {
                j = mid - 1;
            } else {
                i = mid + 1;
            }
        } else if (target < nums[mid]){
            if (nums[i] <= nums[mid] && nums[i] > target) {
                i = mid + 1;
            } else {
                j = mid - 1;
            }
        }
    }
    return -1;
}
```

39. 组合总和

给定一个**无重复元素**的数组 `candidates` 和一个目标数 `target` ，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 `target`）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7,
所求解集为：
[
  [7],
  [2,2,3]
]
```

示例 2：

```
输入：candidates = [2,3,5], target = 8,
所求解集为：
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        def dfs(candidates, i, target, line, ans):
            if target == 0:
                ans.append( line[:] )
                return

            for j in range(i, n):
                if candidates[j] > target:
                    break
                line.append(candidates[j])
                dfs(candidates, j, target - candidates[j], line, ans)
                line.pop()
            return

        candidates = sorted(candidates)
        line = []
        ans = []
        n = len(candidates)
        dfs(candidates, 0, target, line, ans)

        return ans
```

40. 组合总和 II

给定一个数组 `candidates` 和一个目标数 `target` ，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 解集不能包含重复的组合。

示例 1:

```
输入: candidates = [10,1,2,7,6,1,5], target = 8,
所求解集为:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

示例 2:

```
输入: candidates = [2,5,2,1,2], target = 5,
所求解集为:
[
  [1,2,2],
  [5]
]
```

没有技巧，就是遍历所有的情况，使用深度优先搜索 两点需要注意：

1. 排序，提前剪枝
2. 去重

```
class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        def dfs(candidates, i, target, line, ans):
            if target == 0:
                ans.append( line[:] )
                return
            for j in range(i+1, len(candidates)):
                # 排序之后可以剪枝
                if candidates[j] > target:
                    break
                if j > i + 1 and candidates[j] == candidates[j-1]:
                    continue
                line.append(candidates[j])
                dfs(candidates, j, target-candidates[j], line, ans)
                line.pop()
            return

        candidates = sorted(candidates)
        ans = []
        n = len(candidates)
        line = []
        # 以每个点为起点深度优先搜索
        for i in range(n):
            if i > 0 and candidates[i] == candidates[i-1]:
                continue
            line.append(candidates[i])
            dfs(candidates, i, target-candidates[i], line, ans)
            line.pop()

        return ans
```

53. 最大子序和

给定一个整数数组 `nums` ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

```
输入: [-2,1,-3,4,-1,2,1,-5,4],
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。
```

进阶:

如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

简单的动态规划，只和前一个值有关，所以只需要常数个空间。

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int maxSubArray(int* nums, int numsSize){
    int ans = 0;
    int i;
    int pre;

    if (nums == NULL || numsSize == 0) {
        return ans;
    }

    ans = nums[0];
    pre = nums[0];
    for (i = 1; i < numsSize; i++) {
        if (pre + nums[i] > nums[i]) {
            pre = pre + nums[i];
            ans = MAX(ans, pre);
        } else {
            pre = nums[i];
            ans = MAX(ans, pre);
        }
    }
    return ans;
}
```

56. 合并区间

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

```
输入: [[1,3],[2,6],[8,10],[15,18]]
输出: [[1,6],[8,10],[15,18]]
解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

示例 2:

```
输入: [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

先按照起点排序，然后比较相邻的两个区间 如果重叠，合并 如果不重叠，之后的也不可能重叠 所以每次只需要比较相邻的两个区间是否重叠即可。

```
class Solution(object):
    def merge(self, intervals):
        """
        :type intervals: List[List[int]]
        :rtype: List[List[int]]
        """
        intervals = sorted(intervals)
        i = 0
        while i < len(intervals) - 1:
            j = i + 1
            if intervals[i][1] >= intervals[j][0]:
                new_interval = [intervals[i][0], max(intervals[i][1], intervals[j][1])]
                del intervals[i]
                del intervals[i] # 这里要注意，上面已经删除了一个
                intervals.insert(i, new_interval)
            else:
                i = i + 1
        return intervals
```

58. 最后一个单词的长度

给定一个仅包含大小写字母和空格 ' ' 的字符串 s，返回其最后一个单词的长度。如果字符串从左向右滚动显示，那么最后一个单词就是最后出

现的单词。

如果不存在最后一个单词，请返回 0 。

说明：一个单词是指仅由字母组成、不包含任何空格字符的 **最大子字符串**。

示例:

```
输入: "Hello World"
输出: 5
```

该题目不保证只有一个空格，不保证开头末尾没有空格，所以要保守一点，报情况都考虑到。

```
int lengthOfLastWord(char * s){
    if (s == NULL) {
        return 0;
    }

    int n = strlen(s);
    int end = n - 1;
    while (end >= 0 && s[end] == ' ') {
        end -= 1;
    }
    int start = end;
    while (start >= 0 && s[start] != ' ') {
        start -= 1;
    }
    return end - start;


}
```

---

62. 不同路径

一个机器人位于一个 *m x n* 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径?

例如，上图是一个7 x 3 的网格。有多少可能的路径?

示例 1:

```
输入: m = 3, n = 2
输出: 3
解释:
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右
```

示例 2:

```
输入: m = 7, n = 3
输出: 28
```

提示：

- `1 <= m, n <= 100`
- 题目数据保证答案小于等于 `2 * 10 ^ 9`

#define N 300 int r[N];

int uniquePaths(int m, int n){ int i; int j; for (i = 0; i < n; i++) { r[i] = 1; } for (i = 1; i < m; i++) { for (j = 1; j < n; j++) { r[j] = r[j-1] + r[j]; } } return r[n-1]; }

---

64. 最小路径和

给定一个包含非负整数的 *m* x *n* 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例:

输入:
```
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
```
输出: `7`
解释: 因为路径 `1→3→1→1→1` 的总和最小。

简单的动态规划，可以不使用额外的空间，使用原来的数组。

```
#define MIN(a, b) ((a) > (b) ? (b) : (a))

int minPathSum(int** grid, int gridSize, int* gridColSize){
    if (grid == NULL || gridColSize == NULL || gridSize == 0) {
        return 0;
    }

    int i, j;
    for (i = 1; i < gridSize; i++) {
        grid[i][0] = grid[i][0] + grid[i-1][0];
    }
    for (j = 1; j < gridColSize[0]; j++) {
        grid[0][j] = grid[0][j] + grid[0][j-1];
    }
    for (i = 1; i < gridSize; i++) {
        for (j = 1; j < gridColSize[i]; j++) {
            grid[i][j] = grid[i][j] + MIN(grid[i-1][j], grid[i][j-1]);
        }
    }
    return grid[gridSize-1][gridColSize[gridSize-1]-1];
}
```

---

66. 加一

给定一个由**整数**组成的**非空**数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储**单个**数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1:

输入: `[1,2,3]`
输出: `[1,2,4]`
解释: 输入数组表示数字 `123`。

示例 2:

输入： [4,3,2,1]
输出： [4,3,2,2]
解释： 输入数组表示数字 4321。

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* plusOne(int* digits, int digitsSize, int* returnSize){
    if (digits == NULL || digitsSize <= 0) {
        return digits;
    }

    int i;
    int num = digits[digitsSize - 1] + 1;
    int carry = num / 10;

    digits[digitsSize - 1] = num % 10;

    for (i = digitsSize - 2; i >= 0; i--) {
        num = digits[i] + carry;
        digits[i] = num % 10;
        carry = num / 10;
        if (carry == 0) {
            break;
        }
    }
    if (carry == 0) {
        *returnSize = digitsSize;
        return digits;
    }

    /* 如果加1之后最高位有进位，原来的数组已经无法放下了 */
    /* 申请新的空间返回 */
    int *ans = (int*)malloc(sizeof(int) * (digitsSize + 1));
    ans[0] = carry;
    for (i = 0; i < digitsSize; i++) {
        ans[i+1] = digits[i];
    }
    *returnSize = digitsSize + 1;
    return ans;
}
```

67. 二进制求和

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 **非空** 字符串且只包含数字 1 和 0。

示例 1:

输入： a = "11", b = "1"
输出： "100"

示例 2:

输入： a = "1010", b = "1011"
输出： "10101"

提示：

- 每个字符串仅由字符 '0' 或 '1' 组成。
- 1 <= a.length, b.length <= 10^4
- 字符串如果不是 "0" ，就都不含前导零。

```
char * addBinary(char * a, char * b){
    if (a == NULL) return b;
    if (b == NULL) return a;

    int alen = strlen(a);
    int blen = strlen(b);

    if (alen < blen) return addBinary(b, a);

    /* a 的长度大于等于 b */
    int aend = alen - 1;
    int bend = blen - 1;
    char *ans = (char*)malloc(sizeof(char) * (alen + 2));
    int ansEnd = 0;
    int carry = 0;
    int num;

    while (aend >= 0 && bend >= 0) {
        num = a[aend] - '0' + b[bend] - '0' + carry;
        ans[ansEnd++] = num % 2 + '0';
        carry = num / 2;
        aend -= 1;
        bend -= 1;
    }

    while (aend >= 0) {
        num = a[aend] - '0' + carry;
        ans[ansEnd++] = num % 2 + '0';
        carry = num / 2;
        aend -= 1;
    }

    if (carry) {
        ans[ansEnd++] = carry + '0';
    }

    ans[ansEnd] = '\0';

    int left = 0;
    int right = ansEnd - 1;
    char tmp;
    while (left < right) {
        tmp = ans[left];
        ans[left] = ans[right];
        ans[right] = tmp;
        left += 1;
        right -= 1;
    }
    return ans;
}
```

74. 搜索二维矩阵

编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1:

```
输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 3
输出: true
```

示例 2:

输入:
```
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 13
```
输出: false

矩阵中的二分查找，先在第一列二分查找确定行的位置 然后在行中二分查找确定最终是否在矩阵中

```
class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        # 先从第一列二分查找确定在哪一列
        n = len(matrix)
        if n == 0:
            return False
        m = len(matrix[0])
        if m == 0:
            return False
        start = 0
        end = n - 1
        while start <= end:
            mid = (start + end) // 2
            if matrix[mid][0] == target:
                return True
            if target > matrix[mid][0]:
                start = mid + 1
            else:
                end = mid - 1

        # start - 1 行是应该查找的行
        # 再在该行二分查找

        row = start - 1
        start = 0
        end = m - 1
        while start <= end:
            mid = (start + end) // 2
            if matrix[row][mid] == target:
                return True
            if target > matrix[row][mid]:
                start = mid + 1
            else:
                end = mid - 1
        return False
```

---

88. 合并两个有序数组

给你两个有序整数数组 *nums1* 和 *nums2*，请你将 *nums2* 合并到 *nums1* 中，使 *num1* 成为一个有序数组。

说明:

- 初始化 *nums1* 和 *nums2* 的元素数量分别为 *m* 和 *n* 。
- 你可以假设 *nums1* 有足够的空间（空间大小大于或等于 *m* + *n*）来保存 *nums2* 中的元素。

示例:

输入：
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],   n = 3

输出： [1,2,2,3,5,6]

```
/* 从 start 开始查找插入位置并返回下一个位置 */
int insert(int *nums, int size, int start, int value)
{
    int i = start;
    while (i < size && nums[i] < value) i++;
    if (i >= size) {
        nums[i] = value;
        return i;
    }
    for (int j = size - 1; j >= i; j--) {
        nums[j + 1] = nums[j];
    }
    nums[i] = value;
    return i;
}
void merge(int* nums1, int nums1Size, int m, int* nums2, int nums2Size, int n){
    if (nums1 == NULL || nums1Size <= 0 || nums2 == NULL || nums2Size <= 0) {
        return;
    }

    int i;
    int start = 0;
    for (i = 0; i < n; i++) {
        start = insert(nums1, m + i, start, nums2[i]);
    }
    return;
}
```

91. 解码方法

一条包含字母 A-Z 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

给定一个只包含数字的**非空**字符串，请计算解码方法的总数。

示例 1:

输入： "12"
输出： 2
解释： 它可以解码为 "AB"（1 2）或者 "L"（12）。

示例 2:

输入： "226"
输出： 3
解释： 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。

暴力方法，只能通过部分用例 235 / 258 个通过测试用例

```
void dfs(char *s, int i, int *ans)
{
    int num;

    if (i == strlen(s)) {
        (*ans)++;
        return;
    }
    // 消耗一个数字
    num = s[i] - '0';
    if (num < 1 || num > 9) {
        return;
    } else {
        dfs(s, i + 1, ans);
    }

    // 消耗两个数字
    if (i + 1 < strlen(s)) {
        num = (s[i] - '0') * 10 + (s[i + 1] - '0');
        if (num >= 1 && num <= 26) {
            dfs(s, i + 2, ans);
        }
    }
    return;
}
int numDecodings(char * s){
    int ans = 0;
    dfs(s, 0, &ans);
    return ans;
}
```

93. 复原IP地址

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

示例:

```
输入: "25525511135"
输出: ["255.255.11.135", "255.255.111.35"]
```

```
#define MAX_ANS 1000 /* 结果应该不会超过1000个 */
#define MAX_IP 16 /* IP地址最多16个字符可以存储下 */
#define TRUE 1
#define FALSE 0


void dfs(char *s, int start, char *line, char **lines, int *linesEnd)
{
    if (start == strlen(s)) {
        lines[linesEnd++], line
    }
}
char ** restoreIpAddresses(char * s, int* returnSize){
    if (s == NULL) {
        *returnSize = 0;
        return NULL;
    }

    int line[4];
    int lineEnd = 0; /* 4个整形存储一个IP地址 */

    char **lines;
    int linesEnd = 0; /* 字符串数组保存最终的结果 */

    lines = (char**)malloc(sizeof(cahr*) * MAX_ANS);
    for (i = 0; i < MAX_ANS; i++) {
        lines[i] = (char*)malloc(sizeof(char) * MAX_IP);
        memset(lines[i], 0, sizeof(cahr) * MAX_IP);
    }
    dfs(s, 0, line, &lineEnd, lines, &linesEnd);
    *returnSize = linesEnd;
    return lines;
}
```

102. 二叉树的层序遍历

给你一个二叉树，请你返回其按 **层序遍历** 得到的节点值。 （即逐层地，从左到右访问所有节点）。

示例：
二叉树：`[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层次遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

```
class Solution(object):
    def levelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        ans = []
        deque = []
        if root is None:
            return ans
        deque.append(root)
        ans.append([root.val])
        count = 1
        while len(deque) > 0:
            root = deque.pop(0)
            if root.left:
                deque.append(root.left)
            if root.right:
                deque.append(root.right)
            count -= 1
            if count == 0 and len(deque) > 0:
                ans.append([i.val for i in deque])
                count = len(deque)
        return ans
```

121. 买卖股票的最佳时机

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

找到最小值，和最小值后面的最大值，取差值最大的那一对

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        if not prices:
            return 0

        ans = 0
        i = 1
        low = prices[0]
        high = prices[0]
        while i < len(prices):
            if prices[i] < low:
                low = prices[i]
                high = prices[i]
            if prices[i] > high:
                high = prices[i]
            ans = max(ans, high - low)
            i += 1
        return ans
```

122. 买卖股票的最佳时机 II

给定一个数组，它的第 *i* 个元素是一支给定股票第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 7
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。
```

示例 2:

```
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

每一个相邻的增长都可以是最后的利润

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        ans = 0
        i = 1
        while i < len(prices):
            if prices[i] > prices[i-1]:
                ans += prices[i] - prices[i-1]
            i += 1
        return ans
```

还可以找到相邻的最低和最高点，求差值和

```
class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        ans = 0
        if not prices:
            return 0

        prices.insert(0, prices[0] + 1)
        prices.append(prices[-1] - 1)
        i = 1
        while i < len(prices) - 1:
                    # 注意有值相同的情况
            if prices[i] <= prices[i-1] and prices[i] <= prices[i+1]:
                low = prices[i]
            if prices[i] >= prices[i-1] and prices[i] >= prices[i+1]:
                ans += prices[i] - low
                low = prices[i]
            i += 1
        return ans
```

---

127. 单词接龙

给定两个单词（*beginWord* 和 *endWord*）和一个字典，找到从 *beginWord* 到 *endWord* 的最短转换序列的长度。转换需遵循如下规则：

1. 每次转换只能改变一个字母。
2. 转换过程中的中间单词必须是字典中的单词。

说明:

- 如果不存在这样的转换序列，返回 0。
- 所有单词具有相同的长度。
- 所有单词只由小写字母组成。
- 字典中不存在重复的单词。
- 你可以假设 *beginWord* 和 *endWord* 是非空的，且二者不相同。

示例 1:

```
输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",
     返回它的长度 5。
```

示例 2:

```
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: 0

解释: endWord "cog" 不在字典中，所以无法进行转换。
```

```
void freeMatrix(int **matrix, int n)
{
    int i;
    for (i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
    return;
}
// 返回 a 与 b 是否只相差一个字母
int isArrival(char *a, char *b)
{
    int i;
    int count = 0;
    const int m = strlen(a);
    const int n = strlen(b);
    if (n != m) {
        return 0;
    }
    for (i = 0; i < n; i++) {
        if (a[i] != b[i]) {
            count++;
        }
    }
    if (count == 1) {
        return 1;
    }
    return 0;
}
// 宽度优先搜索，返回 start --> end 的最短路径
int BFS(int **graph, int start, int end, int n)
{
    int i;
    int current;
    int road_current;

    int *deque;
    int dequeHead = 0;
    int dequeEnd = 0;

    int *road;
    int roadHead = 0;
    int roadEnd = 0;

    int *color;
    int ans;

    deque = (int*)malloc(sizeof(int) * n);
    road = (int*)malloc(sizeof(int) * n);
    color = (int*)malloc(sizeof(int) * n);

    memset(color, 0, sizeof(int) * n);
    memset(deque, 0, sizeof(int) * n);
    memset(road, 0, sizeof(int) * n);

    deque[dequeHead++] = start;
    color[start] = 1;
    road[roadHead++] = 0;
    while (dequeHead - dequeEnd > 0) {
        current = deque[dequeEnd++];
        road_current = road[roadEnd++];
        for (i = 0; i < n; i++) {
            if (graph[current][i] == 1 && color[i] == 0) {
                if (i == end) {
                    ans =  road_current + 1;
                    free(color);
                    free(deque);
                    free(road);
                    return ans;
                }
                deque[dequeHead++] = i;
                color[i] = 1;
                road[roadHead++] = road_current + 1;
            }
```

```
        }
    }
    free(color);
    free(deque);
    free(road);
    return 0;
}
int ladderLength(char * beginWord, char * endWord, char ** wordList, int wordListSize){

    const int n = wordListSize;
    int **graph;
    int i, j, start, end;
    int ans = 0;

    // 开辟空间存储无向图
    graph = (int**)malloc(sizeof(int*) * (n + 1));
    for (i = 0; i < n + 1; i++) {
        graph[i] = (int*)malloc(sizeof(int) * (n + 1));
        memset(graph[i], 0, sizeof(int) * (n + 1));
    }
    // 根据单词生成无向图
    for (i = 0; i < n - 1; i++) {
        for (j = i+1; j < n; j++) {
            if (i != j && isArrival(wordList[i], wordList[j])) {
                graph[i][j] = 1;
                graph[j][i] = 1;
            }
        }
    }
    // 找到起点的索引
    start = -1;
    for (i = 0; i < n; i++) {
        if (strcmp(wordList[i], beginWord) == 0) {
            start = i;
        }
    }
    if (start == -1) {
        // beginword 作为第 n 个结点加入图
        for (i = 0; i < n; i++) {
            if (isArrival(beginWord, wordList[i])) {
                graph[n][i] = 1;
                graph[i][n] = 1;
                ans = 1;
            }
        }
        start = n;
    }
    // 找到终点的索引
    end = -1;
    for (i = 0; i < n; i++) {
        if (strcmp(wordList[i], endWord) == 0) {
            end = i;
            break;
        }
    }
    // 终点不在单词列表中，返回0
    if (end == -1) {
        freeMatrix(graph, n + 1);
        ans = 0;
        return ans;
    }
    // 起点是n，终点是 end，现在问题转化为搜索两点之间的最短路径
    ans = BFS(graph, start, end, n + 1);
    if (ans == 0) {
        return 0;
    } else {
        return ans + 1;
    }
}
```

129. 求根到叶子节点数字之和

给定一个二叉树，它的每个结点都存放一个 0-9 的数字，每条从根到叶子节点的路径都代表一个数字。

例如，从根到叶子节点路径 1->2->3 代表数字 123 。

计算从根到叶子节点生成的所有数字之和。

说明: 叶子节点是指没有子节点的节点。

示例 1:

```
输入: [1,2,3]
    1
   / \
  2   3
输出: 25
解释:
从根到叶子节点路径 1->2 代表数字 12.
从根到叶子节点路径 1->3 代表数字 13.
因此，数字总和 = 12 + 13 = 25.
```

示例 2:

```
输入: [4,9,0,5,1]
    4
   / \
  9   0
 / \
5   1
输出: 1026
解释:
从根到叶子节点路径 4->9->5 代表数字 495.
从根到叶子节点路径 4->9->1 代表数字 491.
从根到叶子节点路径 4->0 代表数字 40.
因此，数字总和 = 495 + 491 + 40 = 1026.
```

```
void dfs(struct TreeNode *root, int num, int *sum)
{
    num = num * 10 + root->val;
    if (root->left == NULL && root->right == NULL) {
        *sum = *sum + num;
    }
    if (root->left) {
        dfs(root->left, num, sum);
    }
    if (root->right) {
        dfs(root->right, num, sum);
    }
    return;
}
int sumNumbers(struct TreeNode* root){
    int sum = 0;
    int num = 0;
    if (root == NULL) return sum;
    dfs(root, num, &sum);
    return sum;
}
```

131. 分割回文串

给定一个字符串 *s*，将 *s* 分割成一些子串，使每个子串都是回文串。

返回 *s* 所有可能的分割方案。

示例:

输入： "aab"
输出：
[
  ["aa","b"],
  ["a","a","b"]
]

```
class Solution(object):
    def partition(self, s):
        """
        :type s: str
        :rtype: List[List[str]]
        """
        def ish(string):
            """
            是否是回文
            """
            left = 0
            right = len(string) - 1
            while (left < right):
                if string[left] != string[right]:
                    return False
                left += 1
                right -= 1
            return True

        def dfs(start, end, line, ans):
            if end == len(s):
                if start == end:
                    ans.append(line[:])
                return
            # 如果当前是回文，可以从这里切割
            if ish( s[start : end+1] ):
                line.append( s[start : end+1] )
                dfs(end+1, end+1, line, ans)
                line.pop()
            # 无论是否是回文，都可以不从这里切割
            dfs(start, end + 1, line, ans)
            return

        ans = []
        line = []
        start = 0
        end = 0

        if not s:
            return []

        dfs(0, 0, line, ans)
        return ans
```

134. 加油站

在一条环路上有 *N* 个加油站，其中第 *i* 个加油站有汽油 `gas[i]` 升。

你有一辆油箱容量无限的的汽车，从第 *i* 个加油站开往第 *i+1* 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

- 如果题目有解，该答案即为唯一答案。
- 输入数组均为非空数组，且长度相同。
- 输入数组中的元素均为非负数。

示例 1:

输入：
gas = [1,2,3,4,5]
cost = [3,4,5,1,2]

输出： 3

解释：
从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。

示例 2:

输入：
gas = [2,3,4]
cost = [3,4,3]

输出： -1

解释：
你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。
我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油
开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油
开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油
你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。
因此，无论怎样，你都不可能绕环路行驶一周。

```
int canCompleteCircuit(int* gas, int gasSize, int* cost, int costSize){
    int total = 0; // 可以加的所有汽油之和
    int current = 0; // 当前邮箱的汽油
    int i;
    int start = 0;

    for (i = 0; i < gasSize; i++) {
        total += gas[i] - cost[i];
        current += gas[i] - cost[i];
        // 如果当前邮箱中的汽油无法到达下一个加油站，从下一个加油站重新开始
        if (current < 0) {
            current = 0;
            start = i + 1;
        }
    }

    // 如果加油站汽油之和 小于 要消耗的汽油之和， 肯定无法绕一圈
    if (total < 0) {
        return -1;
    }

    return start;
}
```

136. 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。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

输入： [2,2,1]
输出： 1

示例 2:

输入: [4,1,2,1,2]
输出: 4

```
int singleNumber(int* nums, int numsSize){
    int ans = 0;
    int i;

    if (nums == NULL || numsSize == 0) {
        return -1;
    }

    ans = nums[0];
    for (i = 1; i < numsSize; i++) {
        ans ^= nums[i];
    }

    return ans;
}
```

144. 二叉树的前序遍历

给定一个二叉树，返回它的 *前序* 遍历。

示例:

输入: [1,null,2,3]

```
   1
    \
     2
    /
   3
```

输出: [1,2,3]

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

```
#define N 1000
typedef struct TreeNode Node;
typedef Node* NodePtr;
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* preorderTraversal(struct TreeNode* root, int* returnSize){
    int *ansArray;
    int ansEnd = 0;
    NodePtr *stack;
    int stackEnd = 0;

    ansArray = (int*)malloc(sizeof(int) * N);
    stack = (NodePtr*)malloc(sizeof(NodePtr) * N);

    while (root != NULL || stackEnd > 0) {
        while (root != NULL) {
            ansArray[ansEnd++] = root->val;
            stack[stackEnd++] = root;
            root = root->left;
        }
        root = stack[--stackEnd];
        root = root->right;
    }
    *returnSize = ansEnd;

    /* 释放空间 */
    ansArray = realloc(ansArray,sizeof(int) * ansEnd);
    free(stack);
    return ansArray;
}
```

145. 二叉树的后序遍历

给定一个二叉树，返回它的 *后序* 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [3,2,1]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

```
#define N 1000
typedef struct TreeNode Node;
typedef Node* NodePtr;
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* postorderTraversal(struct TreeNode* root, int* returnSize)
{
    int *ansArray;
    int ansEnd = 0;
    NodePtr *stack;
    int stackEnd = 0;
    struct TreeNode *lastVisited = root;

    if (root == NULL) {
        *returnSize = 0;
        return NULL;
    }

    ansArray = (int*)malloc(sizeof(int) * N);
    stack = (NodePtr*)malloc(sizeof(NodePtr) * N);

    stack[stackEnd++] = root;
    while (stackEnd > 0) {
        root = stack[stackEnd-1];
        if (root->left == NULL && root->right == NULL ||
            root->right == NULL && lastVisited == root->left ||
            lastVisited == root->right) {
                ansArray[ansEnd++] = root->val;
                lastVisited = root;
                stackEnd--;
        } else {
            if (root->right != NULL) {
                stack[stackEnd++] = root->right;
            }
            if (root->left != NULL) {
                stack[stackEnd++] = root->left;
            }
        }
    }
    *returnSize = ansEnd;

    /* 释放空间 */
    ansArray = realloc(ansArray,sizeof(int) * ansEnd);
    free(stack);
    return ansArray;
}
```

164. 最大间距

给定一个无序的数组，找出数组在排序之后，相邻元素之间最大的差值。

如果数组元素个数小于 2，则返回 0。

示例 1:

```
输入: [3,6,9,1]
输出: 3
解释: 排序后的数组是 [1,3,6,9], 其中相邻元素 (3,6) 和 (6,9) 之间都存在最大差值 3。
```

示例 2:

```
输入: [10]
输出: 0
解释: 数组元素个数小于 2，因此返回 0。
```

说明:

- 你可以假设数组中所有元素都是非负整数，且数值在 32 位有符号整数范围内。
- 请尝试在线性时间复杂度和空间复杂度的条件下解决此问题。

排序之后直接计算可以通过，但是此题的本意不是如此

```
class Solution(object):
    def maximumGap(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        nums = sorted(nums)
        ans = 0
        for i in range(1, len(nums)):
            ans = max(ans, nums[i] - nums[i-1])
        return ans
```

171. Excel表列序号

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 1:

```
输入: "A"
输出: 1
```

示例 2:

```
输入: "AB"
输出: 28
```

示例 3:

```
输入: "ZY"
输出: 701
```

致谢:
特别感谢 @ts (http://leetcode.com/discuss/user/ts) 添加此问题并创建所有测试用例。

```
int titleToNumber(char * s){
    int n = strlen(s);
    long long base = 1;
    int i;
    long long num;
    long long ans = 0;

    for (i = n - 1; i >= 0; i--) {
        num = s[i] - 'A' + 1;
        ans += num * base;
        base = base * 26;
    }
    return (int)ans;
}
```

179. 最大数

给定一组非负整数，重新排列它们的顺序使之组成一个最大的整数。

示例 1:

```
输入: [10,2]
输出: 210
```

示例 2:

```
输入: [3,30,34,5,9]
输出: 9534330
```

说明: 输出结果可能非常大，所以你需要返回一个字符串而不是整数。

本质上是确定一种排序时比较大小的方式，确定好大小之后，排序，输出。

1. 注意0转换成字符串的时候需要特殊处理
2. 注意结果是全0的时候输出一个0，而不能输出一串0

```
#define N 100 // 数字的最大位数
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int num2strSubProcess(char *str, int end, int num)
{
    int left = end;
    int right;
    char tmp;

    // 0 需要特殊处理
    if (num == 0) {
        str[end++] = '0';
        return end;
    }

    while (num) {
        str[end++] = (char)(num % 10 + '0');
        num = num / 10;
    }

    right = end - 1;

    while (left < right) {
        tmp = str[left];
        str[left] = str[right];
        str[right] = tmp;
        left += 1;
        right -= 1;
    }

    return end;
}

void num2str(char *str, int num1, int num2)
{
    int end = 0;

    end = num2strSubProcess(str, end, num1);
    end = num2strSubProcess(str, end, num2);
    str[end] = '\0';

    return;
}
// num1 < num2 实现本题的关键比较逻辑
// 比较的方法是比较 num1num2 和 num2num1 的大小
int less(int num1, int num2)
{
    char *str1 = (char*)malloc(sizeof(char) * N * 2);
    char *str2 = (char*)malloc(sizeof(char) * N * 2);
    int i, n;

    num2str(str1, num1, num2);
    num2str(str2, num2, num1);
    n = strlen(str1);

    for (i = 0; i < n; i++) {
        if (str1[i] < str2[i]) {
            return 1;
        } else if (str1[i] > str2[i]) {
            return 0;
        }
    }

    return 0;
}
int partition(int *nums, int begin, int end)
{
    int x, i, j, tmp;

    x = nums[end];
    i = begin - 1;
    for (j = begin; j < end; j++) {
        if (less(nums[j], x) == 1) {
            i = i + 1;
```

```
            tmp = nums[i];
            nums[i] = nums[j];
            nums[j] = tmp;
        }
    }

    tmp = nums[i + 1];
    nums[i + 1] = nums[end];
    nums[end] = tmp;

    return i + 1;
}

void sorted(int *nums, int begin, int end)
{
    int mid;

    if (begin > end) {
        return;
    }

    mid = partition(nums, begin, end);
    sorted(nums, begin, mid - 1);
    sorted(nums, mid + 1, end);
    return;
}

char * largestNumber(int* nums, int numsSize){
    int i, num, left, right, digit;
    char tmp;
    char *ans;
    int ansEnd = 0;

    if (numsSize == 0 || nums == NULL) {
        return NULL;
    }

    // 先按照规则从小到大排序
    sorted(nums, 0, numsSize - 1);

    ans = (char*)malloc(sizeof(char) * numsSize * N);

    // 依次取出每个数字，转化成字符拼接在一起
    for (i = numsSize - 1; i >= 0; i--) {
        num = nums[i];

        // 0 需要特殊处理
        if (num == 0) {
            ans[ansEnd++] = '0';
            continue;
        }

        left = ansEnd;
        right = left;
        while (num) {
            digit = num % 10;
            num = num / 10;
            ans[right++] = (char)(digit + '0');
        }
        ansEnd = right;
        right -= 1;
        while (left < right) {
            tmp = ans[left];
            ans[left] = ans[right];
            ans[right] = tmp;
            left++;
            right--;
        }
    }
    ans[ansEnd] = '\0';

    // 如果 ans 中全部都是0 缩写成一个0
    int flag = 1;
    for (i = 0; i < ansEnd; i++) {
        if (ans[i] != '0') {
```

```
            flag = 0;
            break;
        }
    }
    if (flag) {
        ans[0] = '0';
        ans[1] = '\0';
    }
    return ans;
}
```

187. 重复的DNA序列

所有 DNA 都由一系列缩写为 A，C，G 和 T 的核苷酸组成，例如：“ACGAATTCCG”。在研究 DNA 时，识别 DNA 中的重复序列有时会对研究非常有帮助。

编写一个函数来查找 DNA 分子中所有出现超过一次的 10 个字母长的序列（子串）。

示例：

```
输入：s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
输出：["AAAAACCCCC", "CCCCCAAAAA"]
```

```
class Solution(object):
    def findRepeatedDnaSequences(self, s):
        """
        :type s: str
        :rtype: List[str]
        """
        dic = {}
        ans = []
        for i in range(len(s) - 9):
            cur = s[i : i + 10]
            if cur not in dic:
                dic[cur] = 1
            else:
                dic[cur] += 1
        for key, value in dic.iteritems():
            if value > 1:
                ans.append(key)
        return ans
```

198. 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警**。

给定一个代表每个房屋存放金额的非负整数数组，计算你**在不触动警报装置的情况下**，能够偷窃到的最高金额。

示例 1:

```
输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

示例 2:

```
输入: [2,7,9,3,1]
输出: 12
解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
     偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

深度优先搜索，遍历所有的情况，可以通过大部分用例，但是最后会超时。

```
#define MAX(a,b) ((a) > (b)) ? (a) : (b)
int maxv = 0;

void dfs(int *nums, int numsSize, int i, int tmpMax)
{
    if (i >= numsSize) {
        maxv = MAX(maxv, tmpMax);
        return;
    }

    // 选择i
    dfs(nums, numsSize, i + 2, tmpMax + nums[i]);

    // 不选择i
    dfs(nums, numsSize, i + 1, tmpMax);
}
int rob(int* nums, int numsSize){
    int ans = 0;
    maxv = 0;
    if (numsSize <= 0) {
        return maxv;
    }
    dfs(nums, numsSize, 0, ans);
    return maxv;
}
```

递归消耗时间太多，只适合N较小的情况，下面使用动态规划，计算以每个数字结尾的情况下偷的最多的钱，然后取最大值。

```

#define MAX(a,b) ((a) > (b)) ? (a) : (b)
int rob(int* nums, int numsSize){
    int *dp;
    int maxv = 0;
    int premax = 0;
    int i;

    if (numsSize <= 0) {
        return 0;
    }
    if (numsSize == 1) {
        return nums[0];
    }
    if (numsSize == 2) {
        return MAX(nums[0],nums[1]);
    }
    dp = (int*)malloc(sizeof(int) * numsSize);
    dp[0] = nums[0];
    dp[1] = nums[1];
    premax = nums[0];
    maxv = MAX(nums[0], nums[1]);

    for (i = 2; i < numsSize; i++) {
        dp[i] = premax + nums[i];
        premax = MAX(premax, dp[i-1]);
        maxv = MAX(maxv, dp[i]);
    }
    return maxv;
}
```

201. 数字范围按位与

给定范围 [m, n]，其中 0 <= m <= n <= 2147483647，返回此范围内所有数字的按位与（包含 m, n 两端点）。

示例 1:

```
输入: [5,7]
输出: 4
```

示例 2:

```
输入: [0,1]
输出: 0
```

```
int rangeBitwiseAnd(int m, int n){
    int diff = n - m;
    int ans = 0;
    long long base = 1;
    int i, j;
    if (diff == 0) {
        return m;
    }
    for (i = 0; i < 32; i++) {
        if (diff >= base) {
            m = m >> 1;
            n = n >> 1;
        } else {
            ans = m;
            for (j = m + 1; j <= n; j++) {
                ans = ans & j;
            }
            return ans << i;
        }
        base = base * 2;
    }
    return 0;
}
```

203. 移除链表元素

删除链表中等于给定值 ***val*** 的所有节点。

示例:

```
输入: 1->2->6->3->4->5->6, val = 6
输出: 1->2->3->4->5
```

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

struct ListNode* removeElements(struct ListNode* head, int val){
    struct ListNode *pre;
    struct ListNode *curr;
    struct ListNode *g;

    g = (struct ListNode*)malloc(sizeof(struct ListNode));
    g->next = head;
    pre = g;
    curr = head;

    while (curr) {
        if (curr && curr->val == val) {
            pre->next = curr->next;
            curr = curr->next;
        } else {
            pre = curr;
            curr = curr->next;
        }
    }
    return g->next;
}
```

204. 计数质数

统计所有小于非负整数 *n* 的质数的数量。

示例:

```
输入: 10
输出: 4
解释: 小于 10 的质数一共有 4 个, 它们是 2, 3, 5, 7 。
```

```
/* 是质数返回1，不是返回0 */
int isPrime(int a)
{
    int i;
    for (i = 2; i * i <= a; i++) {
        if (a % i == 0) {
            return 0;
        }
    }
    return 1;
}
int countPrimes(int n){
    int ans = 0;
    int i;

    /* 1不是质数 */
    for (i = 2; i < n; i++) {
        ans += isPrime(i);
    }
    return ans;
}
```

205. 同构字符串

给定两个字符串 *s* 和 *t*，判断它们是否是同构的。

如果 *s* 中的字符可以被替换得到 *t* ，那么这两个字符串是同构的。

所有出现的字符都必须用另一个字符替换，同时保留字符的顺序。两个字符不能映射到同一个字符上，但字符可以映射自己本身。

示例 1:

```
输入: s = "egg", t = "add"
输出: true
```

示例 2:

```
输入: s = "foo", t = "bar"
输出: false
```

示例 3:

```
输入: s = "paper", t = "title"
输出: true
```

说明:
你可以假设 *s* 和 *t* 具有相同的长度。

需要使用两个MAP来确保是一 一 映射的关系。

```
#define MAXV 256
int mapAB[MAXV]; /* 记录从A到B的映射关系 */
int mapBA[MAXV]; /* 记录从B到A的映射关系 */

void init()
{
    int i;
    for (i = 0; i < MAXV; i++) {
        mapAB[i] = -1;
        mapBA[i] = -1;
    }
    return;
}
bool isIsomorphic(char * s, char * t){
    if (s == NULL && t == NULL) return true;
    if (s == NULL || t == NULL) return false;
    if (strlen(s) != strlen(t)) return false;

    int n = strlen(s);
    int i;

    init();
    for (i = 0; i < n; i++) {
        if (mapAB[ s[i] ] == -1 && mapBA[ t[i] ] == -1) {
            mapAB[ s[i] ] = t[i];
            mapBA[ t[i] ] = s[i];
        } else if (mapAB[ s[i] ] == -1) {
            /* s[i] --> t[i] 且 other --> t[i], 两个字符映射到同一个 */
            if (mapBA[ t[i] ] != s[i]) {
                return false;
            }
            mapAB[ s[i] ] = t[i];
        } else {
            /* s[i] --> t[i] 且 s[i] --> other, 一个字符映射到两个 */
            if (mapAB[ s[i] ] != t[i]) {
                return false;
            }
            mapBA[ t[i] ] = s[i];
        }
    }
    return true;
}
```

207. 课程表

你这个学期必须选修 numCourse 门课程，记为 0 到 numCourse-1 。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：[0,1]

给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习？

示例 1:

```
输入: 2, [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。
```

示例 2:

```
输入: 2, [[1,0],[0,1]]
输出: false
解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。
```

提示：

1. 输入的先决条件是由 **边缘列表** 表示的图形，而不是 邻接矩阵 。详情请参见图的表示法 (http://blog.csdn.net/woaidapaopao/article/details/51732947)。
2. 你可以假定输入的先决条件中没有重复的边。
3. 1 <= numCourses <= 10^5

使用深度优先所有判断是否有环：

```
1. 如果再深度优先搜索的时候遇到灰色的节点，说明有环。
```

```
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        def dfs(g, i, color):
            """
            判断图中是否存在环
            """
            color[i] = 1
            for j in g[i]:

                if color[j] == 1:
                    return True
                if color[j] == 0:
                    ans = dfs(g, j, color)
                    if ans == True:
                        return True
            color[i] = 2
            return False

        g = [ [] for _ in range(numCourses)]
        for u, v in prerequisites:
            g[v].append(u)
        color = [0] * numCourses
        for i in range(numCourses):
            if color[i] == 0:
                ans = dfs(g, i, color)
                if ans == True:
                    return False
        return True
```

---

210. 课程表 II

现在你总共有 *n* 门课需要选，记为 `0` 到 `n-1` 。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: `[0,1]`

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

```
输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。
```

示例 2:

```
输入: 4, [[1,0],[2,0],[3,1],[3,2]]
输出: [0,1,2,3] or [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
     因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。
```

说明:

1. 输入的先决条件是由**边缘列表**表示的图形，而不是邻接矩阵。详情请参见图的表示法(http://blog.csdn.net/woaidapaopao/article/details/51732947)。
2. 你可以假定输入的先决条件中没有重复的边。

提示:

1. 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2. 通过 DFS 进行拓扑排序 (https://www.coursera.org/specializations/algorithms) - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3. 拓扑排序也可以通过 BFS (https://baike.baidu.com/item/%E5%AE%BD%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2/5224802?fr=aladdin&fromid=2148012&fromtitle=%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2) 完成。

---

python 实现拓扑排序，注意图不联通的时候，结果要加入最终结果的最前面 保证最后访问的节点最先输出。

```
class Solution(object):
    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """
        def dfs(i, ans):
            """
            搜索节点i，有环返回True，无环返回False
            如果无环，ans保存拓扑排序结果
            """
            color[i] = 1
            for j in g[i]:
                if color[j] == 1:
                    return True
                if color[j] == 0:
                    tmp = dfs(j, ans)
                    if tmp == True:
                        return True
            color[i] = 2
            ans.insert(0, i)
            return False

        g = [ [] for i in range(numCourses)]
        for a, b in prerequisites:
            g[b].append(a)

        color = [0 for _ in range(numCourses)]
        result = []
        for i in range(numCourses):
            if color[i] == 0:
                ans = []
                tmp = dfs(i, ans)
                if tmp == True:
                    return []
                else:
                    ans.extend(result)
                    result = ans[:]
        return result
```

215. 数组中的第K个最大元素

在未排序的数组中找到第 **k** 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4

说明:

你可以假设 k 总是有效的，且 $1 \le k \le$ 数组的长度。

使用堆排序完成：

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * 使用数组表示的左孩子是堆，位置0存放第一个元素
 * 节点 i  2 * i + 1
 * 节点 i 的右孩子是 2 * i + 2
 * 节点 i 的父亲是 (i - 1) / 2
 */
int left(int i)
{
    return (i << 1) + 1;
}
int right(int i)
{
    return (i << 1) + 2;
}
int parent(int i)
{
    if (i <= 0) {
        return -1;
    }
    return (i - 1) >> 1;
}
void swap(int *array, int index1, int index2)
{
    int tmp;
    tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
}
// 向下调整节点 i, 大顶堆
void adjustDown(int *heap, int heapSize, int i)
{
    int leftNode, rightNode;
    int largest = i;
    leftNode = left(i);
    if (leftNode < heapSize && heap[leftNode] > heap[largest]) {
        largest = leftNode;
    }
    rightNode = right(i);
    if (rightNode < heapSize && heap[rightNode] > heap[largest]) {
        largest = rightNode;
    }
    if (largest == i) {
        return;
    }
    swap(heap, i, largest);
    adjustDown(heap, heapSize, largest);
}
// 向上调整节点 i,  大顶堆
void adjustUp(int *heap, int heapSize, int i)
{
    int parentNode;
    parentNode = parent(i);
    if (i != -1 && heap[parentNode] < heap[i]) {
        swap(heap, i, parentNode);
        adjustUp(heap, heapSize, parentNode);
    }
    return;
}
// 建立堆
int* buildHeap(int *array, int arraySize, int *heapSize)
{
    int *heap;
    int n = arraySize / 2 - 1;
    int i;

    heap = (int*)malloc(sizeof(int) * arraySize);
    memset(heap, 0, sizeof(int) * arraySize);
    memcpy(heap, array, sizeof(int) * arraySize);

    *heapSize = arraySize;
    for (i = n; i >= 0; i--) {
```

```
        adjustDown(heap, arraySize, i);
    }

    return heap;
}
int findKthLargest(int* nums, int numsSize, int k){
    int *heap;
    int heapSize;
    int ans, i;
    int tmpK = k - 1;
    heap = buildHeap(nums, numsSize, &heapSize);
    while (tmpK--) {
        swap(heap, 0, heapSize-1);
        heapSize--;
        adjustDown(heap, heapSize, 0);
    }

    ans = heap[0];

    free(heap);
    return ans;
}
```

使用归并排序完成：

```
#include <stdio.h>
#include <stdlib.h>

void merge(int *array, int start, int mid, int end)
{
    int lstart = start;
    int lend = mid;
    int rstart = mid + 1;
    int rend = end;
    int *tmp;
    int tmpEnd = 0;
    int index, i;

    tmp = (int*)malloc(sizeof(int) * (end - start + 1));
    while (tmpEnd < (end-start+1)) {
        if (lstart <= lend && rstart <= rend && array[lstart] < array[rstart]) {
            tmp[tmpEnd++] = array[lstart++];
        } else if (lstart <= lend && rstart <= rend && array[lstart] > array[rstart]) {
            tmp[tmpEnd++] = array[rstart++];
        } else if (lstart > lend) {
            tmp[tmpEnd++] = array[rstart++];
        } else {
            tmp[tmpEnd++] = array[lstart++];
        }
    }
    index = start;
    for (i = 0; i < tmpEnd; i++) {
        array[index++] = tmp[i];
    }
    return;
}
void mergeSort(int *array, int start, int end)
{
    int mid;
    if (end <= start) {
        return;
    }
    mid = (end + start) / 2;
    mergeSort(array,start, mid);
    mergeSort(array, mid + 1, end);
    merge(array, start, mid, end);
    return;
}
int findKthLargest(int* nums, int numsSize, int k){
    return 0;
}
```

使用快速排序完成：

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int *array, int index1, int index2)
{
    int tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
}
int partition(int *array, int start, int end)
{
    int x = end;
    int i = start;
    int j;
    for (j = start; j < end; j++) {
        if (array[j] < array[end]) {
            swap(array, i, j);
            i++;
        }
    }
    swap(array, i, end);
    return i;
}
void quickSort(int *array, int start, int end)
{
    int mid;
    if (start >= end) {
        return;
    }
    mid = partition(array, start, end);
    quickSort(array, start, mid - 1);
    quickSort(array, mid + 1, end);
}
int findKthLargest(int* nums, int numsSize, int k){
    int mid;
    int start = 0;
    int end = numsSize - 1;
    while (start <= end) {
        mid = partition(nums, start, end);
        if (mid == numsSize - k) {
            return nums[mid];
        } else if (mid < numsSize - k) {
            start = mid + 1;
        } else {
            end = mid - 1;
        }
    }
    return 0;
}
```

221. 最大正方形

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

示例:

输入：

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

输出： 4

```
class Solution(object):
    def maximalSquare(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
        """
        maxEdge = 0
        n = len(matrix) + 1
        if n == 1:
            m = 1
        else:
            m = len(matrix[0]) + 1
        dp = [ [0] * m for i in range(n) ]
        for i in range(1,n):
            for j in range(1,m):
                if matrix[i-1][j-1] == '1':
                    dp[i][j] = min([ dp[i-1][j], dp[i][j-1], dp[i-1][j-1] ]) + 1
                    maxEdge = max([maxEdge, dp[i][j]])
        return maxEdge * maxEdge
```

231. 2的幂

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1:

```
输入: 1
输出: true
解释: 2^0 = 1
```

示例 2:

```
输入: 16
输出: true
解释: 2^4 = 16
```

示例 3:

```
输入: 218
输出: false
```

```
bool isPowerOfTwo(int n){
    while (n % 2 == 0) {
        n = n / 2;
    }
    if (n == 1) {
        return true;
    }
    return false;
}
```

242. 有效的字母异位词

给定两个字符串 *s* 和 *t* ，编写一个函数来判断 *t* 是否是 *s* 的字母异位词。

示例 1:

```
输入: s = "anagram", t = "nagaram"
输出: true
```

示例 2:

输入: *s* = "rat", *t* = "car"
输出: false

说明:
你可以假设字符串只包含小写字母。

进阶:
如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

```
int sc[26];
int tc[26];

bool isAnagram(char * s, char * t){
    int n1 = strlen(s);
    int n2 = strlen(t);
    int i;

    if (n1 != n2) {
        return false;
    }

    for (i = 0; i < 26; i++) {
        sc[i] = 0;
        tc[i] = 0;
    }

    for (i = 0; i < n1; i++) {
        sc[ s[i] - 'a' ] += 1;
        tc[ t[i] - 'a' ] += 1;
    }

    for (i = 0; i < 26; i++) {
        if (sc[i] != tc[i]) {
            return false;
        }
    }

    return true;
}
```

263. 丑数

编写一个程序判断给定的数是否为丑数。

丑数就是只包含质因数 `2`, `3`, `5` 的**正整数**。

示例 1:

输入: 6
输出: true
解释: 6 = 2 × 3

示例 2:

输入: 8
输出: true
解释: 8 = 2 × 2 × 2

示例 3:

输入: 14
输出: false
解释: 14 不是丑数，因为它包含了另外一个质因数 7。

说明:

1. `1` 是丑数。
2. 输入不会超过 32 位有符号整数的范围: $[-2^{31},\ 2^{31}-1]$。

bool isUgly(int num){ if (num == 0) { return false; } while (num % 2 == 0) { num = num / 2; } while (num % 3 == 0) { num = num / 3; } while (num % 5 == 0) { num = num / 5; } if (num == 1) { return true; } return false; }

283. 移动零

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

```
输入: [0,1,0,3,12]
输出: [1,3,12,0,0]
```

说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

```
void moveZeroes(int* nums, int numsSize){
    int i, j;
    int count = 0; // 0 的个数
    for (i = numsSize - 1; i >= 0; i--) {
        // 一旦遇到0，就把后面的移动到前面，最后补0，count+1
        if (nums[i] == 0) {
            for (j = i + 1; j < numsSize - count; j++) {
                nums[j - 1] = nums[j];
            }
            nums[j - 1] = 0;
            count += 1;
        }
    }
}
```

318. 最大单词长度乘积

给定一个字符串数组 `words`，找到 `length(word[i]) * length(word[j])` 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

示例 1:

```
输入: ["abcw","baz","foo","bar","xtfn","abcdef"]
输出: 16
解释: 这两个单词为 "abcw", "xtfn"。
```

示例 2:

```
输入: ["a","ab","abc","d","cd","bcd","abcd"]
输出: 4
解释: 这两个单词为 "ab", "cd"。
```

示例 3:

```
输入: ["a","aa","aaa","aaaa"]
输出: 0
解释: 不存在这样的两个单词。
```

```
class Solution(object):
    def maxProduct(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        bits = []
        for word in words:
            b = 0
            for c in word:
                b = b | ( 1 << (ord(c) - ord('a')) )
            bits.append(b)

        ans = 0
        for i in range(len(bits)):
            for j in range(len(bits)):
                if i != j and bits[i] & bits[j] == 0:
                    ab = len(words[i]) * len(words[j])
                    ans = max([ans, ab])

        return ans
```

322. 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

示例 1:

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

示例 2:

```
输入: coins = [2], amount = 3
输出: -1
```

说明:
你可以认为每种硬币的数量是无限的。

```c
#define MIN(a,b) ((a) > (b) ? (b) : (a))

int compare(const void *p, const void *q)
{
    int a = *(int *)p;
    int b = *(int *)q;
    return a - b;
}
int coinChange(int* coins, int coinsSize, int amount){
    int **dp;
    int i, j;
    int ans1, ans2, ans;

    qsort(coins, coinsSize, sizeof(int), compare);

    dp = (int**)malloc(sizeof(int*) * coinsSize);
    for (i = 0; i < coinsSize; i++) {
        dp[i] = (int*)malloc(sizeof(int) * (amount + 1));
        memset(dp[i], 0, sizeof(int) * (amount + 1));
    }

    // 填充第一行，只使用第一个硬币
    for (j = 0; j <= amount; j++) {
        if (j % coins[0] == 0) {
            dp[0][j] = j / coins[0];
        }
    }

    for (i = 1; i < coinsSize; i++) {
        for (j = 1; j <= amount; j++) {
            // 无法使用第j个硬币，因为它本身就比目标大
            if (coins[i] > j) {
                dp[i][j] = dp[i-1][j];
            } else {
                ans1 = dp[i-1][j]; // 使用前i-1个硬币组成j
                ans2 = dp[i][j - coins[i]] + 1; // 使用前i个组成j-coins[j] + 一个当前硬币
                dp[i][j] = MIN(ans1, ans2);
            }
        }
    }

    for (i = 0; i < coinsSize; i++) {
        for (j = 0; j <= amount; j++) {
            printf("%d ", dp[i][j]);
        }
        printf("\n");
    }

    if (amount == 0) {
        ans = 0;
    } else if (dp[coinsSize-1][amount] == 0) {
       ans = -1;
    } else {
        ans = dp[coinsSize-1][amount];
    }

    // 释放申请的空间
    for (i = 0; i < coinsSize; i++) {
        free(dp[i]);
    }
    free(dp);

    return ans;
}
```

342. 4的幂

给定一个整数 (32 位有符号整数)，请编写一个函数来判断它是否是 4 的幂次方。

示例 1:

输入： 16
输出： true

示例 2:

输入： 5
输出： false

进阶：
你能不使用循环或者递归来完成本题吗?

```
bool isPowerOfFour(int num){
    if (num == 0) {
        return false;
    }
    while (num % 4 == 0) {
        num = num / 4;
    }
    if (num == 1) {
        return true;
    }
    return false;
}
```

---

378. 有序矩阵中第K小的元素

给定一个 $n \times n$ 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第k小的元素。
请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素。

示例:

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

返回 13。
```

提示：
你可以假设 k 的值永远是有效的, $1 \le k \le n^2$ 。

```
from heapq import heappush, heappop
class Solution(object):
    def kthSmallest(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        minheap = []
        n = len(matrix)
        for i in range(min(k, n)):  # 当k小于n时，只需遍历前k行
            heappush(minheap, (matrix[i][0], i, 0))  # 堆里的每个元素是（matrix[i][j], i, j）

        counter = 0
        x, i, j = 0, 0, 0
        while counter < k:
            counter += 1
            x, i, j = heappop(minheap)
            if j < n-1:
                heappush(minheap, (matrix[i][j+1], i, j+1))  # 向堆里加入该元素所在行的下一个元素
        return x
```

414. 第三大的数

给定一个非空数组，返回此数组中第三大的数。如果不存在，则返回数组中最大的数。要求算法时间复杂度必须是O(n)。

示例 1:

输入: [3, 2, 1]

输出: 1

解释: 第三大的数是 1.

示例 2:

输入: [1, 2]

输出: 2

解释: 第三大的数不存在, 所以返回最大的数 2 .

示例 3:

输入: [2, 2, 3, 1]

输出: 1

解释: 注意，要求返回第三大的数，是指第三大且唯一出现的数。
存在两个值为2的数，它们都排第二。

```
int thirdMax(int* nums, int numsSize){
    long long max1 = LLONG_MIN;
    long long max2 = LLONG_MIN;
    long long max3 = LLONG_MIN;
    int i;

    for (i = 0; i < numsSize; i++) {
        if (nums[i] > max1) {
            max3 = max2;
            max2 = max1;
            max1 = nums[i];
        }
        if (nums[i] < max1 && nums[i] > max2) {
            max3 = max2;
            max2 = nums[i];
        }
        if (nums[i] < max2 && nums[i] > max3) {
            max3 = nums[i];
        }
    }
    if (max3 == LLONG_MIN) { return max1; }
    return max3;
}
```

415. 字符串相加

给定两个字符串形式的非负整数 `num1` 和 `num2` ，计算它们的和。

注意：

1. `num1` 和 `num2` 的长度都小于 5100.
2. `num1` 和 `num2` 都只包含数字 `0-9` .
3. `num1` 和 `num2` 都不包含任何前导零。
4. **你不能使用任何内建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式。**

```
char *reversed(char *string)
{
    int i, j;
    int len = strlen(string);
    int mid = len / 2;
    char c;
    char *ans;
    ans = (char*)malloc(sizeof(char) * (len + 1));
    strcpy(ans, string);
    i = 0;
    j = len - 1;
    while (i < j) {
        c = ans[i];
        ans[i] = ans[j];
        ans[j] = c;
        i++;
        j--;
    }
    ans[len] = '\0';
    return ans;
}
char * addStrings(char * num1, char * num2){
    int n1, n2, i;
    int carry = 0;
    int sum = 0;
    char *reversedNum1;
    char *reversedNum2;
    char *reversedAns;
    char *ans;

    n1 = strlen(num1);
    n2 = strlen(num2);
    if (n1 < n2) {
        return addStrings(num2, num1);
    }
    if (n1 == 0 && n2 == 0) {
        return num1;
    }
    if (n1 == 0) {
        return num2;
    }
    if (n2 == 0) {
        return num1;
    }

    reversedAns = (char*)malloc(sizeof(char) * (strlen(num1) + 2));
    reversedNum1 = reversed(num1);
    reversedNum2 = reversed(num2);

    for (i = 0; i < n2; i++) {
        sum = reversedNum1[i] - '0' + reversedNum2[i] - '0' + carry;
        reversedAns[i] = sum % 10 + '0';
        carry = sum / 10;
    }
    for (;i < n1; i++) {
        sum = reversedNum1[i] - '0' + carry;
        reversedAns[i] = sum % 10 + '0';
        carry = sum / 10;
    }
    if (carry > 0) {
        reversedAns[i] = carry + '0';
        i++;
    }
    reversedAns[i] = '\0';
    ans = reversed(reversedAns);
    free(reversedNum1);
    free(reversedNum2);
    free(reversedAns);
    return ans;
}
```

454. 四数相加 II

给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 `(i, j, k, l)` ，使得 `A[i] + B[j] + C[k] + D[l] = 0` 。

为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 $0 \le N \le 500$ 。所有整数的范围在 -2^{28} 到 $2^{28} - 1$ 之间，最终结果不会超过 $2^{31} - 1$ 。

例如:

```
输入:
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]

输出:
2

解释:
两个元组如下:
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
```

先求两个数组所有可能的和，存入dic 两个dic计算相反数的个数，相乘再加总即可。

```
class Solution(object):
    def fourSumCount(self, A, B, C, D):
        """
        :type A: List[int]
        :type B: List[int]
        :type C: List[int]
        :type D: List[int]
        :rtype: int
        """
        def getSum(a, b):
            dic = {}
            n = len(a)
            for i in range(n):
                for j in range(n):
                    if a[i] + b[j] in dic:
                        dic[ a[i] + b[j] ] += 1
                    else:
                        dic[ a[i] + b[j] ] = 1
            return dic

        dicab = getSum(A, B)
        diccd = getSum(C, D)
        ans = 0
        flag = True
        for key, value in diccd.iteritems():
            if -key in dicab:
                ans += dicab[-key] * value
        return ans
```

463. 岛屿的周长

给定一个包含 0 和 1 的二维网格地图，其中 1 表示陆地 0 表示水域。

网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（或者说，一个或多个表示陆地的格子相连组成的岛屿）。

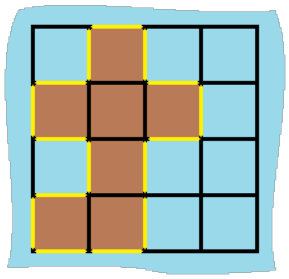
岛屿中没有“湖”（“湖” 指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。网格为长方形，且宽度和高度均不超过 100 。计算这个岛屿的周长。

示例 :

输入：
```
[[0,1,0,0],
 [1,1,1,0],
 [0,1,0,0],
 [1,1,0,0]]
```

输出： 16

解释： 它的周长是下面图片中的 16 个黄色的边：



```
int get(int **grid, int gridSize, int col, int i, int j)
{
    int count = 4;
    if (i - 1 >= 0 && grid[i-1][j] == 1) {
        count--;
    }
    if (i + 1 < gridSize && grid[i+1][j] == 1) {
        count--;
    }
    if (j - 1 >= 0 && grid[i][j-1] == 1) {
        count--;
    }
    if (j + 1 < col && grid[i][j+1] == 1) {
        count--;
    }
    //printf("%d ", count);
    return count;
}
int islandPerimeter(int** grid, int gridSize, int* gridColSize){
    int i, j;
    int ans = 0;
    for (i = 0; i < gridSize; i++) {
        for (j = 0; j < gridColSize[i]; j++) {
            if (grid[i][j] == 1) {
                ans += get(grid, gridSize, gridColSize[i], i, j);
            }
        }
    }
    return ans;
}
```

468. 验证IP地址

编写一个函数来验证输入的字符串是否是有效的 IPv4 或 IPv6 地址。

IPv4 地址由十进制数和点来表示，每个地址包含4个十进制数，其范围为 0 - 255， 用(".")分割。比如， `172.16.254.1` ；

同时，IPv4 地址内的数不会以 0 开头。比如，地址 `172.16.254.01` 是不合法的。

IPv6 地址由8组16进制的数字来表示，每组表示 16 比特。这些组数字通过 (":")分割。比如， `2001:0db8:85a3:0000:0000:8a2e:0370:7334` 是一个有效的地址。而且，我们可以加入一些以 0 开头的数字，字母可以使用大写，也可以是小写。所以， `2001:db8:85a3:0:0:8A2E:0370:7334` 也是一个有效的 IPv6 address地址 (即，忽略 0 开头，忽略大小写)。

然而，我们不能因为某个组的值为 0，而使用一个空的组，以至于出现 (::) 的情况。 比如， `2001:0db8:85a3::8A2E:0370:7334` 是无效的 IPv6 地址。

同时，在 IPv6 地址中，多余的 0 也是不被允许的。比如， `02001:0db8:85a3:0000:0000:8a2e:0370:7334` 是无效的。

说明: 你可以认为给定的字符串里没有空格或者其他特殊字符。

示例 1:

输入: "172.16.254.1"

输出: "IPv4"

解释: 这是一个有效的 IPv4 地址, 所以返回 "IPv4"。

示例 2:

输入: "2001:0db8:85a3:0:0:8A2E:0370:7334"

输出: "IPv6"

解释: 这是一个有效的 IPv6 地址, 所以返回 "IPv6"。

示例 3:

输入: "256.256.256.256"

输出: "Neither"

解释: 这个地址既不是 IPv4 也不是 IPv6 地址。

```
#define N 20
void setAns(char *ans, int ipType)
{
    if (ipType == 1) {
        ans[0] = 'I';
        ans[1] = 'P';
        ans[2] = 'v';
        ans[3] = '4';
        ans[4] = '\0';
    } else if (ipType == 2) {
        ans[0] = 'I';
        ans[1] = 'P';
        ans[2] = 'v';
        ans[3] = '6';
        ans[4] = '\0';
    } else {
        ans[0] = 'N';
        ans[1] = 'e';
        ans[2] = 'i';
        ans[3] = 't';
        ans[4] = 'h';
        ans[5] = 'e';
        ans[6] = 'r';
        ans[7] = '\0';
    }
}
int find(char *IP, char c)
{
    const int n = strlen(IP);
    int i;
    for (i = 0; i < n; i++) {
        if (IP[i] == c) {
            return 1;
        }
    }
    return 0;
}
// 是 ipv4地址返回 1
int isIpv4(char *IP)
{
    // 按照 . 分割成4个数字，检查4个数组是否合法
    const int n = strlen(IP);
    int i = 0;
    int num = 0;
    int count = 0;
    while (i < n) {
        if (IP[i] >= '0' && IP[i] <= '9'){
            if (num == 0 && IP[i] == '0') {
                // 有前导0不合法
                if (i+1 < n && IP[i+1] == '.') {
                    i++;
                    continue;
                }
                if (i+1 == n) {
                    i++;
                    continue;
                }
                return 10;
            }
            if (num >= 256) {
                return 10;
            }
            num = num * 10 + IP[i] - '0';
        } else if (IP[i] == '.') {
            count++;
            if (num < 0 || num >= 256) {
                // 超出数据范围不合法
                return 10;
            }
            if (i == n - 1) {
                // 最后一个不能是 .
                return 10;
            }
            if (i + 1 < n && IP[i + 1] == '.') {
```

```
                // . 不能直接相连
                return 10;
            }
            num = 0;
        } else {
            return 10;
        }
        i++;
    }
    // 最后一组的数字也要合法
    if (num < 0 || num >= 256) {
        return 10;
    }
    // 应该恰好有 3 个点号，分成4组
    if (count != 3) {
        return 10;
    }
    return 1;
}
// 是 ipv6 地址返回 2
int isIpv6(char *IP)
{
    const int n = strlen(IP);
    int i, j;
    int count = 0; // : 的个数
    int every_count = 0; // 每个组内部不应该超过4位
    i = 0;
    while (i < n) {
        if ( (IP[i] >= '0' && IP[i] <= '9')  ||
              (IP[i] >= 'a' && IP[i] <= 'f') ||
              (IP[i] >= 'A' && IP[i] <= 'F') ) {
                  every_count++;
        } else if (IP[i] == ':') {
            count++;
            if (i == n - 1) {
                return 10;
            } else if (i + 1 < n && IP[i+1] == ':') {
                return 10;
            }
            if (every_count > 4) {
                return 10;
            }
            every_count = 0;
        } else {
            return 10;
        }
        i++;
    }
    if (every_count > 4) {
        return 10;
    }
    if (count >= 8) {
        return 10;
    }
    return 2;
}
char * validIPAddress(char * IP){
    char *ans;
    int ipType;
    ans = (char*)malloc(sizeof(char) * N);
    if (find(IP, '.')) {
        ipType = isIpv4(IP);
    } else if (find(IP, ':')) {
        ipType = isIpv6(IP);
    } else {
        ipType = 3;
    }

    setAns(ans, ipType);
    return ans;
}
```

495. 提莫攻击

在《英雄联盟》的世界中，有一个叫 “提莫” 的英雄，他的攻击可以让敌方英雄艾希（编者注：寒冰射手）进入中毒状态。现在，给出提莫对艾希的攻击时间序列和提莫攻击的中毒持续时间，你需要输出艾希的中毒状态总时长。

你可以认为提莫在给定的时间点进行攻击，并立即使艾希处于中毒状态。

示例1:

```
输入: [1,4], 2
输出: 4
原因: 在第 1 秒开始时，提莫开始对艾希进行攻击并使其立即中毒。中毒状态会维持 2 秒钟，直到第 2 秒钟结束。
在第 4 秒开始时，提莫再次攻击艾希，使得艾希获得另外 2 秒的中毒时间。
所以最终输出 4 秒。
```

示例2:

```
输入: [1,2], 2
输出: 3
原因: 在第 1 秒开始时，提莫开始对艾希进行攻击并使其立即中毒。中毒状态会维持 2 秒钟，直到第 2 秒钟结束。
但是在第 2 秒开始时，提莫再次攻击了已经处于中毒状态的艾希。
由于中毒状态不可叠加，提莫在第 2 秒开始时的这次攻击会在第 3 秒钟结束。
所以最终输出 3。
```

注意:

1. 你可以假定时间序列数组的总长度不超过 10000。
2. 你可以假定提莫攻击时间序列中的数字和提莫攻击的中毒持续时间都是非负整数，并且不超过 10,000,000。

维护一个当前中毒持续截至时间的变量end, 扫描每个时间点，如果当前时间点大于等于end, 持续时间+duration 如果当前时间点小于end, 说明中毒时间还没有结束，中毒时间有损失，为 time + duration - end. 扫描一遍即可得到结果，时间复杂度O(n).

```
class Solution(object):
    def findPoisonedDuration(self, timeSeries, duration):
        """
        :type timeSeries: List[int]
        :type duration: int
        :rtype: int
        """
        dead = 0
        end = 0
        for time in timeSeries:
            if time >= end:
                dead += duration
            else:
                dead += time + duration - end
            end = time + duration
        return dead
```

---

504. 七进制数

给定一个整数，将其转化为7进制，并以字符串形式输出。

示例 1:

```
输入: 100
输出: "202"
```

示例 2:

```
输入: -7
输出: "-10"
```

注意: 输入范围是 [-1e7, 1e7] 。

```
#define N 10000

char * convertToBase7(int num){
    bool negtive = false;
    int i;
    char *ans;
    int end = 0;
    int tmp;
    char t;
    int left;
    int right;

    ans = (char*)malloc(sizeof(char) * N);

    if (num == 0) {
        ans[0] = '0';
        ans[1] = '\0';
        return ans;
    }
    if (num < 0) {
        num = -num;
        negtive = true;
        ans[end++] = '-';
    }

    while (num > 0) {
        tmp = num % 7;
        num = num / 7;
        ans[end++] = tmp + '0';
    }

    left = negtive ? 1 : 0;
    right = end - 1;
    while (left < right) {
        t = ans[left];
        ans[left] = ans[right];
        ans[right] = t;
        left++;
        right--;
    }
    ans[end] = '\0';
    return ans;
}
```

515. 在每个树行中找最大值

您需要在二叉树的每一行中找到最大的值。

示例：

输入：

```
          1
         / \
        3   2
       / \   \
      5   3   9
```

输出： [1, 3, 9]

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

class Solution(object):
    def largestValues(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        ans = []
        if root is None:
            return ans
        count = 1
        deque = []

        deque.append(root)
        ans.append(root.val)
        while len(deque) > 0:
            cur = deque.pop(0)
            if cur.left:
                deque.append(cur.left)
            if cur.right:
                deque.append(cur.right)
            count -= 1
            if count == 0 and len(deque) > 0:
                ans.append( max([node.val for node in deque]) )
                count = len(deque)
        return ans
```

541. 反转字符串 II

给定一个字符串和一个整数 k，你需要对从字符串开头算起的每个 2k 个字符的前k个字符进行反转。如果剩余少于 k 个字符，则将剩余的所有全部反转。如果有小于 2k 但大于或等于 k 个字符，则反转前 k 个字符，并将剩余的字符保持原样。

示例:

```
输入: s = "abcdefg", k = 2
输出: "bacdfeg"
```

要求:

1. 该字符串只包含小写的英文字母。
2. 给定字符串的长度和 k 在[1, 10000]范围内。

```
void reverse(char *s, int start, int end)
{
    char tmp;
    while (start < end) {
        tmp = s[start];
        s[start] = s[end];
        s[end] = tmp;
        start++;
        end--;
    }
}
char * reverseStr(char * s, int k){
    int n = strlen(s);
    int count2k = n / (2 * k);
    int last = n - count2k * k * 2;
    int start = -1;
    int end = -1;
    int i;
    for (i = 0; i < count2k; i++) {
        start = i * 2 * k;
        end = start + k - 1;
        reverse(s, start, end);
    }
    start = count2k * 2 * k;
    if (last >= k) {
        end = start + k - 1;
        reverse(s, start, end);
    } else {
        end = n - 1;
        reverse(s, start, end);
    }

    return s;
}
```

605. 种花问题

假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花），和一个数 **n** 。能否在不打破种植规则的情况下种入 **n** 朵花？能则返回True，不能则返回False。

示例 1:

```
输入: flowerbed = [1,0,0,0,1], n = 1
输出: True
```

示例 2:

```
输入: flowerbed = [1,0,0,0,1], n = 2
输出: False
```

注意:

1. 数组内已种好的花不会违反种植规则。
2. 输入的数组长度范围为 [1, 20000]。
3. **n** 是非负整数，且不会超过输入数组的大小。

先处理只有一个元素的情况 对于两个及以上的元素，判断每个位置是否能种花，如果能种， n - 1 同时将该位置种上花。 最后判断N是否为0.

```
bool canPlaceFlowers(int* flowerbed, int flowerbedSize, int n){
    int i;
    // 只有一个元素的单独处理
    if (flowerbedSize == 1 && flowerbed[0] == 0 && n == 1) {
        return true;
    }
    for (i = 0; i < flowerbedSize; i++) {
        if (flowerbed[i] == 1) {
            continue;
        }
        // 如果位于起始位置，只要后面一个不是1，就可以种花
        if (i == 0 && i + 1 < flowerbedSize && flowerbed[i + 1] == 0) {
            n = n - 1;
            flowerbed[i] = 1;
        }
        // 如果位于最后，只要前一个位置不是1，就可以种花
        if (i == flowerbedSize - 1 && i - 1 >= 0 && flowerbed[i - 1] == 0) {
            n = n - 1;
            flowerbed[i] = 1;
        }
        if (i - 1 >= 0 && i + 1 < flowerbedSize && flowerbed[i - 1] == 0 && flowerbed[i + 1] == 0) {
            n = n - 1;
            flowerbed[i] = 1;
        }
    }

    if (n > 0) {
        return false;
    }

    return true;
}
```

649. Dota2 参议院

Dota2 的世界里有两个阵营： `Radiant` (天辉)和 `Dire` (夜魇)

Dota2 参议院由来自两派的参议员组成。现在参议院希望对一个 Dota2 游戏里的改变作出决定。他们以一个基于轮为过程的投票进行。在每一轮中，每一位参议员都可以行使两项权利中的 一 项：

1. 禁止一名参议员的权利 ：

   参议员可以让另一位参议员在这一轮和随后的几轮中丧失**所有的权利**。

2. 宣布胜利 ：

   如果参议员发现有权利投票的参议员都是**同一个阵营的**，他可以宣布胜利并决定在游戏中的有关变化。

给定一个字符串代表每个参议员的阵营。字母 “R” 和 “D” 分别代表了 `Radiant` （天辉）和 `Dire` （夜魇）。然后，如果有 `n` 个参议员，给定字符串的大小将是 `n` 。

以轮为基础的过程从给定顺序的第一个参议员开始到最后一个参议员结束。这一过程将持续到投票结束。所有失去权利的参议员将在过程中被跳过。

假设每一位参议员都足够聪明，会为自己的政党做出最好的策略，你需要预测哪一方最终会宣布胜利并在 Dota2 游戏中决定改变。输出应该是 `Radiant` 或 `Dire` 。

示例 1:

```
输入: "RD"
输出: "Radiant"
解释:  第一个参议员来自  Radiant 阵营并且他可以使用第一项权利让第二个参议员失去权力，因此第二个参议员将被跳过因为他没有任何权利。然后在第二轮的时候，第一个参议员可以宣布胜利，因为他是唯一一个有投票权的人
```

示例 2:

输入： "RDD"
输出： "Dire"
解释：
第一轮中,第一个来自 Radiant 阵营的参议员可以使用第一项权利禁止第二个参议员的权利
第二个来自 Dire 阵营的参议员会被跳过因为他的权利被禁止
第三个来自 Dire 阵营的参议员可以使用他的第一项权利禁止第一个参议员的权利
因此在第二轮只剩下第三个参议员拥有投票的权利,于是他可以宣布胜利

注意:

1. 给定字符串的长度在 [1, 10,000] 之间.

有几个用例不通过，需要调试一下，看看算法不对还是程序写的有问题

76 / 81 个通过测试用例

```
class Solution(object):
    def predictPartyVictory(self, senate):
        """
        :type senate: str
        :rtype: str
        """
        def find(l, i):
            """
            返回 i 之后的第一个索引，如果没有返回第一个
            """
            for index, p in enumerate(l):
                if p > i:
                    return (p, index)
            return (0,0)

        dset = []
        rset = []
        senate = list(senate)
        label = [1 for _ in range(len(senate))]
        for i in range(len(senate)):
            if senate[i] == "D":
                dset.append(i)
            if senate[i] == "R":
                rset.append(i)
        while dset and rset:
            for i in range(len(senate)):
                if label[i] == 1 and senate[i] == "D":
                    if not rset:
                        return "Dire"
                    # 选择后面第一个不是同一阵营的删除
                    del_index, index = find(rset, i)
                    label[ del_index ] = 0
                    del rset[index]
                if label[i] == 1 and senate[i] == "R":
                    if not dset:
                        return "Radiant"
                    del_index, index = find(dset, i)
                    label[del_index] = 0
                    del dset[index]
        if dset:
            return "Dire"
        if rset:
            return "Radiant"
```

658. 找到 K 个最接近的元素

给定一个排序好的数组，两个整数 k 和 x，从数组中找到最靠近 x （两数之差最小）的 k 个数。返回的结果必须要是按升序排好的。如果有两个数与 x 的差值一样，优先选择数值较小的那个数。

示例 1:

```
输入: [1,2,3,4,5], k=4, x=3
输出: [1,2,3,4]
```

示例 2:

```
输入: [1,2,3,4,5], k=4, x=-1
输出: [1,2,3,4]
```

说明:

1. k 的值为正数，且总是小于给定排序数组的长度。
2. 数组不为空，且长度不超过 10^4
3. 数组里的每个元素与 x 的绝对值不超过 10^4

更新(2017/9/19):
这个参数 *arr* 已经被改变为一个**整数数组**（而不是整数列表）。 ***请重新加载代码定义以获取最新更改。***

---

1. 使用二分查找找到 x 应该插入的位置
2. 使用双指针搜索附近元素，找到距离最近的K个
3. 排序输出

```c
#define ABS(a) ((a) >= 0 ? (a) : (-(a)))
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
void swap(int *arr, int index1, int index2)
{
    int tmp;

    if (index1 == index2) {
        return;
    }

    tmp = arr[index1];
    arr[index1] = arr[index2];
    arr[index2] = tmp;
    return;
}

int partition(int *arr, int start, int end)
{
    int x, i, j;

    x = arr[end];
    j = start - 1;
    for (i = start; i < end; i++) {
        if (arr[i] <= x) {
            j = j + 1;
            swap(arr, j, i);
        }
    }
    swap(arr, j + 1, end);
    return j + 1;
}
void sorted(int *arr, int start, int end)
{
    int mid;

    if (start >= end) {
        return;
    }

    mid = partition(arr, start, end);
    sorted(arr, start, mid - 1);
    sorted(arr, mid + 1, end);
    return;
}
int BinaryFind(int *arr, int arrSize, int target)
{
    int left = 0;
    int right = arrSize - 1;
    int mid;

    while (left < right) {
        mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
int* findClosestElements(int* arr, int arrSize, int k, int x, int* returnSize){
    int *ans;
    int ansEnd = 0;
    int left, right, pos;

    if (arr == NULL || arrSize == 0) {
        return NULL;
    }

    // 二分查找找到 x 应该插入 arr 的位置
```

```
    pos = BinaryFind(arr, arrSize, x);

    // 从 pos 开始双指针左右搜索， 找与 x 差值最小的添加到结果中
    ans = (int*)malloc(sizeof(int) * k);
    left = pos - 1;
    right = pos;
    while (ansEnd < k) {
        if (left >= 0 && right < arrSize) {
            if (ABS(arr[left] - x) <= ABS(arr[right] - x)) {
                ans[ansEnd++] = arr[left];
                left -= 1;
            } else {
                ans[ansEnd++] = arr[right];
                right += 1;
            }
        } else if (left >= 0) {
            ans[ansEnd++] = arr[left];
            left -= 1;
        } else {
            ans[ansEnd++] = arr[right];
            right += 1;
        }
    }

    sorted(ans, 0, ansEnd - 1);

    *returnSize = ansEnd;
    return ans;
}
```

---

714. 买卖股票的最佳时机含手续费

给定一个整数数组 `prices`，其中第 `i` 个元素代表了第 `i` 天的股票价格；非负整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每次交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

示例 1:

```
输入: prices = [1, 3, 2, 8, 4, 9], fee = 2
输出: 8
解释: 能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

注意:

- `0 < prices.length <= 50000`.
- `0 < prices[i] < 50000`.
- `0 <= fee < 50000`.

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
int maxProfit(int* prices, int pricesSize, int fee){
    int unhold = 0;
    int hold = -prices[0];
    int i;
    for (i = 1; i < pricesSize; i++) {
        unhold = max(unhold, hold + prices[i] - fee);
        hold = max(hold, unhold - prices[i]);
    }
    return unhold;
}
```

739. 每日温度

根据每日 气温 列表，请重新生成一个列表，对应位置的输出是需要再等待多久温度才会升高超过该日的天数。如果之后都不会升高，请在该位置用 0 来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是 [1, 1, 4, 2, 1, 1, 0, 0]。

提示： 气温 列表长度的范围是 [1, 30000]。每个气温的值的均为华氏度，都是在 [30, 100] 范围内的整数。

暴力方法，两层循环，最后两个用例无法通过。 35 / 37 个通过测试用例

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* dailyTemperatures(int* T, int TSize, int* returnSize){
    int *ans;
    int i, j;

    ans = (int*)malloc(sizeof(int) * TSize);
    memset(ans, 0, sizeof(int) * TSize);
    for (i = 0; i < TSize - 1; i++) {
        for (j = i + 1; j < TSize; j++) {
            if (T[j] > T[i]) {
                ans[i] = j - i;
                break;
            }
        }
    }
    *returnSize = TSize;
    return ans;
}
```

暴力方法每次都到扫描到最后，无法利用前一次扫描的信息，所以时间复杂度很高。 我们采取从后道前的方法扫描每一个数字， 当 nums[i] < nums[i+1] 时， ans[i] = 1 当nums[i] >= nums[i+1] 时， 需要向后找到第一个大于nums[i]的数字 在向后找的时候，因为已经知道了任意一个数字第一个大于它的数字出现的位置，所以可以不必每次移动一个位置，而是跳跃着找，这样明显可以快很多。

```
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* dailyTemperatures(int* T, int TSize, int* returnSize){
    int *ans;
    int i, j;

    ans = (int*)malloc(sizeof(int) * TSize);
    memset(ans, 0, sizeof(int) * TSize);
    for (i = TSize - 2; i >= 0; i--) {
        if (T[i] < T[i + 1]) {
            ans[i] = 1;
        } else {
            j = i + 1;
            while (j < TSize && T[i] >= T[j]) {
                if (ans[j] == 0) {
                    break;
                }
                j = j + ans[j];
            }
            if (j < TSize && j > i && T[j] > T[i]) {
                ans[i] = j - i;
            }
        }
    }
    *returnSize = TSize;
    return ans;
}
```

---

767. 重构字符串

给定一个字符串 S，检查是否能重新排布其中的字母，使得两相邻的字符不同。

若可行，输出任意可行的结果。若不可行，返回空字符串。

示例 1:

```
输入: S = "aab"
输出: "aba"
```

示例 2:

```
输入: S = "aaab"
输出: ""
```

注意:

- `S` 只包含小写字母并且长度在 `[1, 500]` 区间内。

```
N = 600
class Solution(object):
    def reorganizeString(self, S):
        """
        :type S: str
        :rtype: str
        """
        dic = {}
        for c in S:
            if c not in dic:
                dic[c] = 1
            else:
                dic[c] += 1
        maps = []
        for key, value in dic.iteritems():
            maps.append( [N-value, value, key] )
        heapq.heapify(maps)
        hp = maps
        ans = ""
        while (len(hp) > 0):
            index, value, key = heapq.heappop(hp)
            if len(ans) == 0 or ans[-1] != key:
                ans += key
                value = value - 1
                if value > 0:
                    heapq.heappush(hp, [N-value, value, key])
            else:
                if len(hp) == 0:
                    return ""
                index, value2, key2 = heapq.heappop(hp)
                ans += key2
                value2 = value2 - 1
                if value2 > 0:
                    heapq.heappush(hp, [N-value2, value2, key2] )
                heapq.heappush(hp, [N-value, value, key])
        return ans
```

---

768. 最多能完成排序的块 II

这个问题和“最多能完成排序的块”相似，但给定数组中的元素可以重复，输入数组最大长度为 `2000` ，其中的元素最大为 `10**8` 。

`arr` 是一个可能包含**重复元素**的整数数组，我们将这个数组分割成几个“块”，并将这些块分别进行排序。之后再连接起来，使得连接的结果和按升序排序后的原数组相同。

我们最多能将数组分成多少块?

示例 1:

```
输入: arr = [5,4,3,2,1]
输出: 1
解释:
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [5, 4], [3, 2, 1] 的结果是 [4, 5, 1, 2, 3]，这不是有序的数组。
```

示例 2:

```
输入: arr = [2,1,3,4,4]
输出: 4
解释:
我们可以把它分成两块，例如 [2, 1], [3, 4, 4]。
然而，分成 [2, 1], [3], [4], [4] 可以得到最多的块数。
```

注意:

- `arr` 的长度在 `[1, 2000]` 之间。
- `arr[i]` 的大小在 `[0, 10**8]` 之间。

先排序，这样可以快速获取任何区间最大值和最小值 扫描一趟，找到排序之后的数组和未排序的数组最大值和最小值及其个数都相同的点 一旦找到，结果就增加1，更新循环变量，接着往下找

```
class Solution(object):
    def maxChunksToSorted(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        def getMaxMin(arr):
            minv = min(arr)
            minc = arr.count(minv)
            maxv = max(arr)
            maxc = arr.count(maxv)
            return (minv, minc), (maxv, maxc)

        ans = 0
        i = 0
        n = len(arr)
        start = i
        sorted_arr = sorted(arr)
        while i < n:
            minv, maxv = getMaxMin(sorted_arr[start : i + 1])
            arrMin, arrMax = getMaxMin(arr[start : i + 1])
            if minv == arrMin and maxv == arrMax:
                ans += 1
                start = i + 1
            i = i + 1
        return ans
```

---

769. 最多能完成排序的块

数组 `arr` 是 `[0, 1, ..., arr.length - 1]` 的一种排列，我们将这个数组分割成几个“块”，并将这些块分别进行排序。之后再连接起来，使得连接的结果和按升序排序后的原数组相同。

我们最多能将数组分成多少块?

示例 1:

```
输入: arr = [4,3,2,1,0]
输出: 1
解释:
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [4, 3], [2, 1, 0] 的结果是 [3, 4, 0, 1, 2]，这不是有序的数组。
```

示例 2:

```
输入: arr = [1,0,2,3,4]
输出: 4
解释:
我们可以把它分成两块，例如 [1, 0], [2, 3, 4]。
然而，分成 [1, 0], [2], [3], [4] 可以得到最多的块数。
```

注意:

- `arr` 的长度在 `[1, 10]` 之间。

- arr[i] 是 [0, 1, ..., arr.length - 1] 的一种排列。

相当于找出局部乱序的字数组，只需要按照最大值，最小值判断 最大值和最小值符合原始数组的最大值最小值，就可以重排

```
class Solution(object):
    def maxChunksToSorted(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        def getMaxMin(arr, start, end):
            maxv = max(arr[start : end + 1])
            minv = min(arr[start : end + 1])
            return minv, maxv

        ans = 0
        i = 0
        n = len(arr)
        start = i
        end = -1
        while i < n:
            minv, maxv = start, i
            arrMin, arrMax = getMaxMin(arr, start, i)
            if minv == arrMin and maxv == arrMax:
                ans += 1
                start = i + 1
            i = i + 1
        return ans
```

---

779. 第K个语法符号

在第一行我们写上一个 `0`。接下来的每一行，将前一行中的 `0` 替换为 `01`， `1` 替换为 `10`。

给定行数 `N` 和序数 `K`，返回第 `N` 行中第 `K` 个字符。（`K` 从1开始）

例子:

```
输入: N = 1, K = 1
输出: 0

输入: N = 2, K = 1
输出: 0

输入: N = 2, K = 2
输出: 1

输入: N = 4, K = 5
输出: 1

解释:
第一行: 0
第二行: 01
第三行: 0110
第四行: 01101001
```

注意:

1. `N` 的范围 `[1, 30]`.
2. `K` 的范围 `[1, 2^(N-1)]`.

```
int kthGrammar(int N, int K){
    int count = 0;
    int number;
    int i = N - 2;

    if (N == 1) return 0;
    if (N == 2) return K - 1;

    while (i) {
        number = 1 << i;
        count += K / number;
        K = K % number;
        i--;
    }

    if (count % 2) {
        return K;
    }

    return 1 - K;
}
```

792. 匹配子序列的单词数

给定字符串 S 和单词字典 words，求 words[i] 中是 S 的子序列的单词个数。

示例：
输入：
S = "abcde"
words = ["a", "bb", "acd", "ace"]
输出： 3
解释： 有三个是 S 的子序列的单词：`"a"`, `"acd"`, `"ace"`。

注意：

- 所有在 words 和 S 里的单词都只由小写字母组成。
- S 的长度在 [1, 50000]。
- words 的长度在 [1, 5000]。
- words[i] 的长度在 [1, 50]。

```python
class Solution(object):
    def match(self, dic, word):

        cur = -1
        for c in word:
            if c in dic:
                indexs = dic[c]
                flag = True
                for index in indexs:
                    if index > cur:
                        cur = index
                        flag = False
                        break
                if flag:
                    return False
            else:
                return False
        return True
    def numMatchingSubseq(self, S, words):
        """
        :type S: str
        :type words: List[str]
        :rtype: int
        """
        ans = 0
        dic = {}
        for i, c in enumerate(S):
            if c not in dic:
                dic[c] = [i]
            else:
                dic[c].append(i)
        for word in words:
            if self.match(dic, word):
                ans += 1
        return ans
```

797. 所有可能的路径

给一个有 `n` 个结点的有向无环图，找到所有从 `0` 到 `n-1` 的路径并输出（不要求按顺序）

二维数组的第 i 个数组中的单元都表示有向图中 i 号结点所能到达的下一些结点（译者注：有向图是有方向的，即规定了a→b你就不能从b→a）空就是没有下一个结点了。

示例：
输入： `[[1,2], [3], [3], []]`
输出： `[[0,1,3],[0,2,3]]`
解释： 图是这样的：
```
0--->1
|    |
v    v
2--->3
```
这有两条路：`0 -> 1 -> 3` 和 `0 -> 2 -> 3`.

提示：

- 结点的数量会在范围 `[2, 15]` 内。
- 你可以把路径以任意顺序输出，但在路径内的结点的顺序必须保证。

```python
class Solution(object):
    def dfs(self, graph, start, end, color, road, roads):
        color[start] = 1
        road.append(start)
        if start == end:
            roads.append(road[:])
        else:
            for i in graph[start]:
                if color[i] == 0:
                    self.dfs(graph, i, end, color, road, roads)
        color[start] = 0
        road.pop()
    def allPathsSourceTarget(self, graph):
        """
        :type graph: List[List[int]]
        :rtype: List[List[int]]
        """
        start = 0
        end = len(graph) - 1
        road = []
        color = [0] * len(graph)
        roads = []
        self.dfs(graph, start, end, color, road, roads)
        return roads
```

848. 字母移位

有一个由小写字母组成的字符串 `S`，和一个整数数组 `shifts`。

我们将字母表中的下一个字母称为原字母的 *移位*（由于字母表是环绕的， `'z'` 将会变成 `'a'`）。

例如·， `shift('a') = 'b'`， `shift('t') = 'u'`，以及 `shift('z') = 'a'`。

对于每个 `shifts[i] = x`， 我们会将 `S` 中的前 `i+1` 个字母移位 `x` 次。

返回将所有这些移位都应用到 `S` 后最终得到的字符串。

示例：

```
输入：S = "abc", shifts = [3,5,9]
输出："rpl"
解释：
我们以 "abc" 开始。
将 S 中的第 1 个字母移位 3 次后，我们得到 "dbc"。
再将 S 中的前 2 个字母移位 5 次后，我们得到 "igc"。
最后将 S 中的这 3 个字母移位 9 次后，我们得到答案 "rpl"。
```

提示：

1. `1 <= S.length = shifts.length <= 20000`
2. `0 <= shifts[i] <= 10 ^ 9`

```c
void shift(char *S, int i, int count)
{
    int num;
    num = S[i] - 'a';
    num = (num + count % 26 ) % 26;
    S[i] = num + 'a';
}
char * shiftingLetters(char * S, int* shifts, int shiftsSize){
    int i;
    for (i = shiftsSize - 2; i >= 0; i--) {
        shifts[i] = (shifts[i+1] % 26 + shifts[i] % 26) % 26;
    }
    for (i = 0; i < shiftsSize; i++) {
        shift(S, i, shifts[i]);
    }
    return S;
}
```

870. 优势洗牌

给定两个大小相等的数组 `A` 和 `B`，`A` 相对于 `B` 的*优势*可以用满足 `A[i] > B[i]` 的索引 `i` 的数目来描述。

返回 `A` 的**任意**排列，使其相对于 `B` 的优势最大化。

示例 1：

```
输入：A = [2,7,11,15], B = [1,10,4,11]
输出：[2,11,7,15]
```

示例 2：

```
输入：A = [12,24,8,32], B = [13,25,32,11]
输出：[24,32,8,12]
```

提示：

1. `1 <= A.length = B.length <= 10000`
2. `0 <= A[i] <= 10^9`
3. `0 <= B[i] <= 10^9`

贪心法解题，先把A排序，遍历B中每个元素，从A中找到刚好大于它的最小元素放入结果中，如果没有大于的元素，就选择一个最小的元素放进去。

两层循环，时间复杂度较高，无法通过所有的用例。(59 / 67 个通过测试用例)

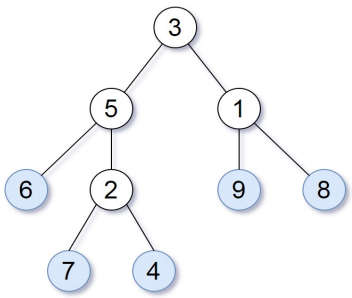
```
class Solution(object):
    def advantageCount(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: List[int]
        """
        A = sorted(A)

        ans = []
        for i in range(len(B)):
            find = False
            for j in range(len(A)):
                if A[j] > B[i]:
                    ans.append(A[j])
                    del A[j]
                    find = True
                    break
            if not find:
                ans.append(A[0])
                del A[0]
        return ans
```

优化查找过程，使用二分查找加快速度，可以通过所有的用例

```
class Solution(object):
    def advantageCount(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: List[int]
        """
        def find(number):
            """
            找到A中第一个大于number的数字的下标
            如果没有找到，返回A中最小数字的下标
            """
            n = len(A)
            left = 0
            right = n - 1
            while left <= right:
                mid = (left + right) // 2
                if A[mid] > number:
                    right = mid - 1
                else:
                    left = mid + 1
            if left < n and A[left] > number:
                return left
            else:
                return 0

        A = sorted(A)

        ans = []
        for i in range(len(B)):
            index = find(B[i])
            #print("{0} {1} {2}\n".format(A[index], B[i], index))
            ans.append(A[index])
            del A[index]

        return ans
```

872. 叶子相似的树

请考虑一颗二叉树上所有的叶子，这些叶子的值按从左到右的顺序排列形成一个 *叶值序列* 。



举个例子，如上图所示，给定一颗叶值序列为 `(6, 7, 4, 9, 8)` 的树。

如果有两颗二叉树的叶值序列是相同，那么我们就认为它们是 *叶相似* 的。

如果给定的两个头结点分别为 `root1` 和 `root2` 的树是叶相似的，则返回 `true`；否则返回 `false` 。

提示:

- 给定的两颗树可能会有 `1` 到 `200` 个结点。
- 给定的两颗树上的值介于 `0` 到 `200` 之间。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

#define N 100

// 返回叶子节点序列
void dfs(struct TreeNode *root, int *leaves, int *leavesEnd)
{
    if (root == NULL) {
        return;
    }
    if (root->left == NULL && root->right == NULL) {
        leaves[(*leavesEnd)++] = root->val;
    }
    if (root->left) {
        dfs(root->left, leaves, leavesEnd);
    }
    if (root->right) {
        dfs(root->right, leaves, leavesEnd);
    }
    return;
}
bool leafSimilar(struct TreeNode* root1, struct TreeNode* root2){
     int *l1;
     int *l2;
     int l1End = 0;
     int l2End = 0;
     int i;

     l1 = (int*)malloc(sizeof(int) * N);
     l2 = (int*)malloc(sizeof(int) * N);
     dfs(root1, l1, &l1End);
     dfs(root2, l2, &l2End);
     if (l1End != l2End) {
         return false;
     }
     //printf("%d ", l1End);
     for (i = 0; i < l1End; i++) {
         if (l1[i] != l2[i]) {
             return false;
         }
     }
     return true;
}
```

884. 两句话中的不常见单词

给定两个句子 `A` 和 `B` 。（*句子*是一串由空格分隔的单词。每个*单词*仅由小写字母组成。）

如果一个单词在其中一个句子中只出现一次，在另一个句子中却没有出现，那么这个单词就是*不常见的*。

返回所有不常用单词的列表。

您可以按任何顺序返回列表。

示例 1：

```
输入：A = "this apple is sweet", B = "this apple is sour"
输出：["sweet","sour"]
```

示例 2：

输入： A = "apple apple", B = "banana"
输出： ["banana"]

提示：

1. 0 <= A.length <= 200
2. 0 <= B.length <= 200
3. A 和 B 都只包含空格和小写字母。

```
class Solution(object):
    def uncommonFromSentences(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: List[str]
        """
        alist = A.split(' ')
        blist = B.split(' ')
        alist.extend(blist)
        dic = {}
        for word in alist:
            if word not in dic:
                dic[word] = 1
            else:
                dic[word] += 1
        ans = []
        for key, value in dic.iteritems():
            if value == 1:
                ans.append(key)
        return ans;
```

---

890. 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

```
#define N 26

int map[N];
int unique[N];
void init(int *array, int value)
{
    int i;
    for (i = 0; i < N; i++) {
        array[i] = value;
    }
}
int isMatch(char *str, char *pattern)
{
    int i;
    char a, b;
    int aa, bb;

    int n1 = strlen(str);
    int n2 = strlen(pattern);
    if (n1 != n2) {
        return false;
    }

    for (i = 0; i < n1; i++) {
        a = str[i];
        b = pattern[i];
        aa = a - 'a';
        bb = b - 'a';
        if (map[aa] == -1) {
            map[aa] = bb;
        } else if (map[aa] != bb) {
            return false;
        }
        if (unique[bb] == -1) {
            unique[bb] = aa;
        } else if (unique[bb] != aa) {
            return false;
        }
    }
    return true;
}
char *copystr(char *str)
{
    int n = strlen(str);
    int i;
    char *newstr;
    newstr = (char*)malloc(sizeof(char) * (n + 1));
    for (i = 0; i < n; i++) {
        newstr[i] = str[i];
    }
    newstr[i] = '\0';
    return newstr;
}
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
char ** findAndReplacePattern(char ** words, int wordsSize, char * pattern, int* returnSize){
    int i;
    char **ans;
    int ansEnd = 0;
    ans = (char**)malloc(sizeof(char*) * wordsSize);
    for (i = 0; i < wordsSize; i++) {
        init(map, -1);
        init(unique, -1);
        if (isMatch(words[i], pattern)) {
            ans[ansEnd++] = copystr(words[i]);
        }
    }
    *returnSize = ansEnd;
    return ans;
}
```

915. 分割数组

给定一个数组 A，将其划分为两个不相交（没有公共元素）的连续子数组 left 和 right， 使得：

- left 中的每个元素都小于或等于 right 中的每个元素。
- left 和 right 都是非空的。
- left 要尽可能小。

在完成这样的分组后返回 left 的**长度**。可以保证存在这样的划分方法。

示例 1：

```
输入：[5,0,3,8,6]
输出：3
解释：left = [5,0,3]，right = [8,6]
```

示例 2：

```
输入：[1,1,1,0,6,12]
输出：4
解释：left = [1,1,1,0]，right = [6,12]
```

提示：

1. 2 <= A.length <= 30000
2. 0 <= A[i] <= 10^6
3. 可以保证至少有一种方法能够按题目所描述的那样对 A 进行划分。

```
class Solution:
     def partitionDisjoint(self, A):
        n = len(A)
        left_max = A[0]
        right_min = min(A[1:])
        right_count = A[1:].count(right_min)

        for i in range(1, n):
            if left_max <= right_min:
                return i
            # A[i] 加入左侧
            if A[i] > left_max:
                left_max = A[i]
            # 右侧排除 A[i]
            if A[i] == right_min:
                if right_count > 1:
                    right_count -= 1
                else:
                    right_min = min(A[i+1:])
                    right_count = A[i+1:].count(right_min)

        return -1
```

927. 三等分

给定一个由 0 和 1 组成的数组 A，将数组分成 3 个非空的部分，使得所有这些部分表示相同的二进制值。

如果可以做到，请返回**任何** [i, j]，其中 i+1 < j，这样一来：

- A[0], A[1], ..., A[i] 组成第一部分；
- A[i+1], A[i+2], ..., A[j-1] 作为第二部分；
- A[j], A[j+1], ..., A[A.length - 1] 是第三部分。
- 这三个部分所表示的二进制值相等。

如果无法做到，就返回 [-1, -1]。

注意，在考虑每个部分所表示的二进制时，应当将其看作一个整体。例如，`[1,1,0]` 表示十进制中的 `6`，而不会是 `3`。此外，前导零也是被允许的，所以 `[0,1,1]` 和 `[1,1]` 表示相同的值。

示例 1：

输入：`[1,0,1,0,1]`
输出：`[0,3]`

示例 2：

输出：`[1,1,0,1,1]`
输出：`[-1,-1]`

提示：

1. `3 <= A.length <= 30000`
2. `A[i] == 0` 或 `A[i] == 1`

```
int* threeEqualParts(int* A, int ASize, int* returnSize){
    int count = 0;
    int c1;
    int i, j, right;
    int *ans;
    ans = (int*)malloc(sizeof(int) * 2);
    ans[0] = -1;
    ans[1] = -1;
    *returnSize = 2;
    for (i = 0; i < ASize; i++) {
        if (A[i] == 1) {
            count++;
        }
    }
    if (count == 0) {
        ans[0] = 0;
        ans[1] = 2;
        return ans;
    }
    if (count % 3 != 0) {
        return ans;
    }
    c1 = count / 3; // 每个值应该包含c1个1
    for (i = ASize - 1; i >= 0; i--) {
        if (A[i] == 1) {
            c1--;
        }
        if (c1 == 0) {
            right = i;
            break;
        }
    }
    // right ... ASize-1 组成的值是最终三个组都应该达到的值
    // 但是right可能不是第三组的开始，如果有0可以提前

    // 从开始找到第一个为1的位置，是否完全能和 right...ASize-1 匹配
    // 如果不能匹配，就返回false
    // 如果可以匹配，就确定了第一组的位置
    i = 0;
    while (i < ASize && A[i] == 0) {
        i++;
    }
    j = right;
    while (i < ASize && j < ASize && A[i] == A[j]) {
        i++;
        j++;
    }
    if (j < ASize) {
        ans[0] = -1;
        ans[1] = -1;
        return ans;
    }
    // 第一组匹配上了， i 的位置就是第二组开始的位置
    ans[0] = i - 1;
    while (i < ASize && A[i] == 0) {
        i++;
    }
    j = right;
    while (i < ASize && j < ASize && A[i] == A[j]) {
        i++;
        j++;
    }
    if (j < ASize) {
        ans[0] = -1;
        ans[1] = -1;
        return ans;
    }
    if (i == right) {
        ans[1] = i;
    } else {
        ans[1] = i;
    }
    return ans;
}
```

944. 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"]，

a	b	c	d	e	f
u	v	w	x	y	z

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"]。

a	b	c	d	e	f
u	v	w	x	y	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。

示例 1：

```
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。
```

示例 2：

```
输入：["a", "b"]
输出：0
解释：D = {}
```

示例 3：

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

提示：

1. 1 <= A.length <= 100
2. 1 <= A[i].length <= 1000

```
int minDeletionSize(char ** A, int ASize){
    int ans = 0;
    int i;
    int j;
    int n = strlen(A[0]);

    for (i = 0; i < n; i++) {
        bool flag = false;
        for (j = 1; j < ASize; j++) {
            if (A[j][i] < A[j - 1][i]) {
                flag = true;
                break;
            }
        }
        if (flag) {
            ans++;
        }
    }
    return ans;
}
```

946. 验证栈序列

给定 pushed 和 popped 两个序列，每个序列中的 **值都不重复**，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时，返回 true；否则，返回 false 。

示例 1：

```
输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

示例 2：

```
输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。
```

提示：

1. 0 <= pushed.length == popped.length <= 1000
2. 0 <= pushed[i], popped[i] < 1000
3. pushed 是 popped 的排列。

```
class Solution(object):
    def validateStackSequences(self, pushed, popped):
        """
        :type pushed: List[int]
        :type popped: List[int]
        :rtype: bool
        """
        n = len(pushed)
        n2 = len(popped)
        if n != n2:
            return False

        i = 0
        j = 0
        stack = []
        while j < n2:
            if len(stack) > 0 and stack[-1] == popped[j]:
                stack.pop()
                j += 1
                continue
            while i < n and pushed[i] != popped[j]:
                stack.append(pushed[i])
                i += 1
            if i >= n:
                return False
            i = i + 1
            j = j + 1
        if len(stack) == 0:
            return True
        else:
            return False
```

973. 最接近原点的 K 个点

我们有一个由平面上的点组成的列表 points 。需要从中找出 K 个距离原点 (0, 0) 最近的点。

（这里，平面上两点之间的距离是欧几里德距离。）

你可以按任何顺序返回答案。除了点坐标的顺序之外，答案确保是唯一的。

示例 1：

```
输入：points = [[1,3],[-2,2]], K = 1
输出：[[-2,2]]
解释：
(1, 3) 和原点之间的距离为 sqrt(10)，
(-2, 2) 和原点之间的距离为 sqrt(8)，
由于 sqrt(8) < sqrt(10)，(-2, 2) 离原点更近。
我们只需要距离原点最近的 K = 1 个点，所以答案就是 [[-2,2]]。
```

示例 2：

```
输入：points = [[3,3],[5,-1],[-2,4]], K = 2
输出：[[3,3],[-2,4]]
（答案 [[-2,4],[3,3]] 也会被接受。）
```

提示：

1. `1 <= K <= points.length <= 10000`
2. `-10000 < points[i][0] < 10000`
3. `-10000 < points[i][1] < 10000`

```
void swap(int **points, int index1, int index2)
{
    int tmp1, tmp2;

    tmp1 = points[index1][0];
    tmp2 = points[index1][1];

    points[index1][0] = points[index2][0];
    points[index1][1] = points[index2][1];

    points[index2][0] = tmp1;
    points[index2][1] = tmp2;

    return;
}
int less(int *a, int *b)
{
    if ((a[0] * a[0] + a[1] * a[1]) <= (b[0] * b[0] + b[1] * b[1])) {
        return 1;
    }
    return 0;
}

int partition(int **points, int start, int end)
{
    int *x = points[end];
    int i;
    int j = start - 1;

    for (i = start; i < end; i++) {
        if (less(points[i], x) == 1) {
            j = j + 1;
            swap(points, i, j);
        }
    }
    swap(points, j + 1, end);
    return j + 1;
}
void sorted(int **points, int start, int end)
{
    int mid;
    if (start >= end) {
        return;
    }
    mid = partition(points, start, end);
    sorted(points, start, mid - 1);
    sorted(points, mid + 1, end);
    return;
}
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller calls free().
 */
int** kClosest(int** points, int pointsSize, int* pointsColSize, int K, int* returnSize, int** returnColumnSizes){
    int **ans;
    int *cols;
    int i, j;

    // 为答案开辟空间
    ans = (int**)malloc(sizeof(int*) * K);
    for (i = 0; i < K; i++) {
        ans[i] = (int*)malloc(sizeof(int) * 2);
    }
    cols = (int*)malloc(sizeof(int) * K);

    // 按照距离原点的距离排序
    sorted(points, 0, pointsSize - 1);

    // 取前K个放入结果中
    for (i = 0; i < K; i++) {
        ans[i][0] = points[i][0];
        ans[i][1] = points[i][1];
        cols[i] = 2;
```

```
    }

    *returnSize = K;
    *returnColumnSizes = cols;
    return ans;
}
```

974. 和可被 K 整除的子数组

给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。

示例：

输入： A = [4,5,0,-2,-3,1], K = 5
输出： 7
解释：
有 7 个子数组满足其元素之和可被 K = 5 整除：
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

提示：

1. 1 <= A.length <= 30000
2. -10000 <= A[i] <= 10000
3. 2 <= K <= 10000

```
#define N 10001
int hashMap[N];
void init()
{
    int i;
    for (i = 0; i < N; i++) {
        hashMap[i] = 0;
    }
}
int subarraysDivByK(int* A, int ASize, int K){
    int *sums;
    int i;
    int ans = 0;

    sums = (int*)malloc(sizeof(int) * ASize);
    sums[0] = A[0];
    for (i = 1; i < ASize; i++) {
        sums[i] = sums[i - 1] + A[i];
    }
    init();
    for (i = 0; i < ASize; i++) {
        sums[i] = (sums[i] % K + K) % K;
        hashMap[sums[i]]++;
        if (sums[i] == 0) {
            ans++;
        }
    }
    for (i = 0; i < N; i++) {
        if (hashMap[i] != 0) {
            ans += hashMap[i] * (hashMap[i] - 1) / 2;
        }
    }
    return ans;
}
```

991. 坏了的计算器

在显示着数字的坏计算器上，我们可以执行以下两种操作：

- **双倍（Double）**：将显示屏上的数字乘 2；
- **递减（Decrement）**：将显示屏上的数字减 1 。

最初，计算器显示数字 `X` 。

返回显示数字 `Y` 所需的最小操作数。

示例 1：

```
输入：X = 2, Y = 3
输出：2
解释：先进行双倍运算，然后再进行递减运算 {2 -> 4 -> 3}.
```

示例 2：

```
输入：X = 5, Y = 8
输出：2
解释：先递减，再双倍 {5 -> 4 -> 8}.
```

示例 3：

```
输入：X = 3, Y = 10
输出：3
解释：先双倍，然后递减，再双倍 {3 -> 6 -> 5 -> 10}.
```

示例 4：

```
输入：X = 1024, Y = 1
输出：1023
解释：执行递减运算 1023 次
```

提示：

1. `1 <= X <= 10^9`
2. `1 <= Y <= 10^9`

考虑从Y得到X比较容易，可以使用贪心法。 如果Y是奇数，先加1再除2 如果Y是偶数，直接除2 知道Y小于X的时候， 再执行X-Y次加法操作

因为除2可以一次操作减少一半，可以更快的使Y逼近X附近

```
class Solution(object):
    def brokenCalc(self, X, Y):
        """
        :type X: int
        :type Y: int
        :rtype: int
        """
        ans = 0
        while Y > X:
            ans += 1
            if Y % 2:
                Y += 1
            else:
                Y = Y // 2
        return ans + X - Y
```

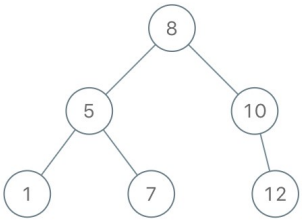
1008. 先序遍历构造二叉树

返回与给定先序遍历 `preorder` 相匹配的二叉搜索树（binary **search** tree）的根结点。

(回想一下，二叉搜索树是二叉树的一种，其每个节点都满足以下规则，对于 `node.left` 的任何后代，值总 `<` `node.val` ，而 `node.right` 的任何后代，值总 `>` `node.val` 。此外，先序遍历首先显示节点的值，然后遍历 `node.left` ，接着遍历 `node.right` 。)

示例：

输入：[8,5,1,7,10,12]
输出：[8,5,10,1,7,null,12]



提示：

1. 1 <= preorder.length <= 100
2. 先序 preorder 中的值是不同的。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

struct TreeNode* bstFromPreorder2(int* preorder, int start, int end){
    int i;
    int left_start = -1;
    int left_end = -2;
    int right_start = -1;
    int right_end = -2;

    // 空数组返回
    if (preorder == NULL) {
        return NULL;
    }
    // 长度为0返回
    if (end - start + 1 <= 0) {
        return NULL;
    }
    // 第一个结点作为跟结点，所有比它小的为做子树，所有比它大的为右子树
    struct TreeNode* root;
    root = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    root->val = preorder[start];

    i = start + 1;
    left_start = start + 1;
    while (i <= end) {
        if (preorder[i] > preorder[start]) {
            left_end = i - 1;
            right_start = i;
            right_end = end;
            break;
        }
        // 如果没有右子树
        if (i == end) {
            left_end = end;
            break;
        }
        i++;
    }
    root->left = bstFromPreorder2(preorder, left_start, left_end);
    root->right = bstFromPreorder2(preorder, right_start, right_end);
    return root;
}
struct TreeNode* bstFromPreorder(int* preorder, int preorderSize){
    return bstFromPreorder2(preorder, 0, preorderSize - 1);
}
```

1009. 十进制整数的反码

每个非负整数 `N` 都有其二进制表示。例如， `5` 可以被表示为二进制 `"101"`， `11` 可以用二进制 `"1011"` 表示，依此类推。注意，除 `N = 0` 外，任何二进制表示中都不含前导零。

二进制的反码表示是将每个 `1` 改为 `0` 且每个 `0` 变为 `1`。例如，二进制数 `"101"` 的二进制反码为 `"010"`。

给你一个十进制数 `N`，请你返回其二进制表示的反码所对应的十进制整数。

示例 1：

```
输入：5
输出：2
解释：5 的二进制表示为 "101"，其二进制反码为 "010"，也就是十进制中的 2 。
```

示例 2：

```
输入：7
输出：0
解释：7 的二进制表示为 "111"，其二进制反码为 "000"，也就是十进制中的 0 。
```

示例 3：

```
输入：10
输出：5
解释：10 的二进制表示为 "1010"，其二进制反码为 "0101"，也就是十进制中的 5 。
```

提示：

1. `0 <= N < 10^9`
2. 本题与 476：https://leetcode-cn.com/problems/number-complement/ (https://leetcode-cn.com/problems/number-complement/) 相同

```
class Solution(object):
    def bitwiseComplement(self, N):
        """
        :type N: int
        :rtype: int
        """
        s = list( bin(N)[2:] )
        for i in range( len(s) ):
            if s[i] == '0':
                s[i] = '1'
            elif s[i] == '1':
                s[i] = '0'
        ss = ''.join(s)
        ans = int(ss, 2)
        return ans
```

1011. 在 D 天内送达包裹的能力

传送带上的包裹必须在 D 天内从一个港口运送到另一个港口。

传送带上的第 `i` 个包裹的重量为 `weights[i]`。每一天，我们都会按给出重量的顺序往传送带上装载包裹。我们装载的重量不会超过船的最大运载重量。

返回能在 `D` 天内将传送带上的所有包裹送达的船的最低运载能力。

示例 1：

输入：weights = [1,2,3,4,5,6,7,8,9,10], D = 5
输出：15
解释：
船舶最低载重 15 就能够在 5 天内送达所有包裹，如下所示：
第 1 天：1, 2, 3, 4, 5
第 2 天：6, 7
第 3 天：8
第 4 天：9
第 5 天：10

请注意，货物必须按照给定的顺序装运，因此使用载重能力为 14 的船舶并将包装分成 (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) 是不允

示例 2：

输入：weights = [3,2,2,4,1,4], D = 3
输出：6
解释：
船舶最低载重 6 就能够在 3 天内送达所有包裹，如下所示：
第 1 天：3, 2
第 2 天：2, 4
第 3 天：1, 4

示例 3：

输入：weights = [1,2,3,1,1], D = 4
输出：3
解释：
第 1 天：1
第 2 天：2
第 3 天：3
第 4 天：1, 1

提示：

1. 1 <= D <= weights.length <= 50000
2. 1 <= weights[i] <= 500

```
// 运载量为K的船是否可以在D天之内运送完货物
int canShip(int *weights, int weightsSize, int D, int k)
{
    int i = 0;
    int sum = 0;
    while (i < weightsSize) {
        sum += weights[i];
        if (sum > k) {
            D--;
            sum = 0;
            i = i - 1;
        }
        i++;
    }
    if (sum > 0) {
        D--;
    }
    if (D >= 0) return 1;
    return 0;
}
int shipWithinDays(int* weights, int weightsSize, int D){
    int mink, maxk;
    int maxw = weights[0];
    int sumw = 0;
    int i;
    int k;
    for (i = 0; i < weightsSize; i++) {
        if (weights[i] > maxw) {
            maxw = weights[i];
        }
        sumw += weights[i];
    }
    mink = maxw;
    maxk = sumw;
    while (mink < maxk) {
        k = (mink + maxk) / 2;
        if (canShip(weights, weightsSize, D, k)) {
            maxk = k;
        } else {
            mink = k + 1;
        }
    }
    return mink;
}
```

1016. 子串能表示从 1 到 N 数字的二进制串

给定一个二进制字符串 `S` （一个仅由若干 '0' 和 '1' 构成的字符串）和一个正整数 `N`，如果对于从 `1` 到 `N` 的每个整数 `X`，其二进制表示都是 `S` 的子串，就返回 `true`，否则返回 `false`。

示例 1：

输入： `S = "0110", N = 3`
输出： `true`

示例 2：

输入： `S = "0110", N = 4`
输出： `false`

提示：

1. `1 <= S.length <= 1000`
2. `1 <= N <= 10^9`

```
#define PN 100
int next[PN];
int nextEnd = 0;

// 计算模式串的 next 数组
void computeNext(const char *pattern)
{
    int i;
    int j = -1;
    const int n = strlen(pattern);
    next[0] = j;
    for (i = 1; i < n; i++) {
        while (j > -1 && pattern[j+1] != pattern[i]) {
            j = next[j];
        }
        if (pattern[i] == pattern[j + 1]) {
            j++;
        }
        next[i] = j;
    }
    nextEnd = n;
}

int kmp(const char *string, const char *pattern)
{
    int i;
    int j = -1;
    const int n = strlen(string);
    const int m = strlen(pattern);
    if (m == 0 && n == 0) return 0;
    if (m == 0) return 0;
    computeNext(pattern);
    for (i = 0; i < n; i++) {
        while (j > -1 && pattern[j+1] != string[i]) {
            j = next[j];
        }
        if (string[i] == pattern[j + 1]) {
            j++;
        }
        if (j == m - 1) {
            return i - j;
        }
    }
    return -1;
}
bool find(char *S, int i)
{
    const int n = strlen(S);
    char num[32];
    char tmp;
    int k = 0;
    while (i) {
        num[k++] = (char)(i & 1) + '0';
        i = i >> 1;
    }
    num[k] = '\0';
    int a = 0;
    int b = k - 1;
    while (a < b) {
        tmp = num[a];
        num[a] = num[b];
        num[b] = tmp;
        a++;
        b--;
    }
    int result;
    result = kmp(S, num);
    if (result == -1) {
        return false;
    }
    return true;
}
bool queryString(char * S, int N){
    int i;
```

```
    for (i = 1; i <= N; i++) {
        if (!find(S, i)) {
            return false;
        }
    }
    return true;
}
```

1023. 驼峰式匹配

如果我们可以将**小写字母**插入模式串 pattern 得到待查询项 query，那么待查询项与给定模式串匹配。（我们可以在任何位置插入每个字符，也可以插入 0 个字符。）

给定待查询列表 queries，和模式串 pattern，返回由布尔值组成的答案列表 answer。只有在待查项 queries[i] 与模式串 pattern 匹配时， answer[i] 才为 true，否则为 false。

示例 1：

```
输入：queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FB"
输出：[true,false,true,true,false]
示例：
"FooBar" 可以这样生成："F" + "oo" + "B" + "ar"。
"FootBall" 可以这样生成："F" + "oot" + "B" + "all".
"FrameBuffer" 可以这样生成："F" + "rame" + "B" + "uffer".
```

示例 2：

```
输入：queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FoBa"
输出：[true,false,true,false,false]
解释：
"FooBar" 可以这样生成："Fo" + "o" + "Ba" + "r".
"FootBall" 可以这样生成："Fo" + "ot" + "Ba" + "ll".
```

示例 3：

```
输出：queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FoBaT"
输入：[false,true,false,false,false]
解释：
"FooBarTest" 可以这样生成："Fo" + "o" + "Ba" + "r" + "T" + "est".
```

提示：

1. 1 <= queries.length <= 100
2. 1 <= queries[i].length <= 100
3. 1 <= pattern.length <= 100
4. 所有字符串都仅由大写和小写英文字母组成。

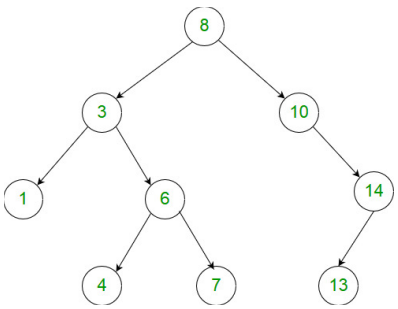
```
#define N 1001
bool isMatch(char *s, char *pattern)
{
    int n = strlen(s);
    int m = strlen(pattern);
    int i = 0, j = 0;
    while (i < n && j < m) {
        if (s[i] == pattern[j]) {
            i++;
            j++;
        } else if (s[i] >= 'a' && s[i] <= 'z') {
            i++;
        } else {
            return false;
        }
    }
    if (j < m) {
        return false;
    }
    if (i < n) {
        while (i < n) {
            if (s[i] >= 'A' && s[i] <= 'Z') {
                return false;
            }
            i++;
        }
    }
    return true;
}
bool* camelMatch(char ** queries, int queriesSize, char * pattern, int* returnSize){
    bool *ans;
    bool ansOne;
    int i;
    int ansEnd = 0;
    ans = (bool*)malloc(sizeof(bool) * N);
    for (i = 0; i< queriesSize; i++) {
        if (isMatch(queries[i], pattern)) {
            ans[ansEnd++] = true;
        } else {
            ans[ansEnd++] = false;
        }
    }
    (*returnSize) = ansEnd;
    return ans;
}
```

1026. 节点与其祖先之间的最大差值

给定二叉树的根节点 `root`，找出存在于不同节点 `A` 和 `B` 之间的最大值 `V`，其中 `V = |A.val - B.val|`，且 `A` 是 `B` 的祖先。

（如果 A 的任何子节点之一为 B，或者 A 的任何子节点是 B 的祖先，那么我们认为 A 是 B 的祖先）

示例：



输入：[8,3,10,1,6,null,14,null,null,4,7,13]
输出：7
解释：
我们有大量的节点与其祖先的差值，其中一些如下：
|8 - 3| = 5
|3 - 7| = 4
|8 - 1| = 7
|10 - 13| = 3
在所有可能的差值中，最大值 7 由 |8 - 1| = 7 得出。

提示：

1. 树中的节点数在 2 到 5000 之间。
2. 每个节点的值介于 0 到 100000 之间。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

#define N 5000

int road[N];
int roadEnd = 0;
int ans = 0;

int abs2(int a, int b)
{
    if (a > b) {
        return a - b;
    } else {
        return b - a;
    }
}
// 计算新入栈的元素和栈内其他元素的差的绝对值
// 并更新 ans
void compute(int *road, int roadEnd)
{
    int i, diff;
    for (i = 0; i < roadEnd - 1; i++) {
        diff = abs2(road[i], road[roadEnd - 1]);
        if (diff > ans) {
            ans = diff;
        }
    }
    printf("%d ", ans);
}
void dfs(struct TreeNode* root, int *road, int *roadEnd)
{
    road[(*roadEnd)++] = root->val;
    compute(road, *roadEnd);
    if (root->left) {
        dfs(root->left, road, roadEnd);
    }
    if (root->right) {
        dfs(root->right, road, roadEnd);
    }
    (*roadEnd)--;
    return;
}
int maxAncestorDiff(struct TreeNode* root){
    if (root == NULL) {
        return 0;
    }
    roadEnd = 0;
    ans = 0;
    dfs(root, road, &roadEnd);
    return ans;
}
```

1028. 从先序遍历还原二叉树

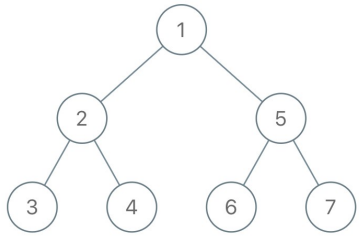
我们从二叉树的根节点 `root` 开始进行深度优先搜索。

在遍历中的每个节点处，我们输出 `D` 条短划线（其中 `D` 是该节点的深度），然后输出该节点的值。（*如果节点的深度为 `D`，则其直接子节点的深度为 `D + 1`。根节点的深度为 `0`*）。

如果节点只有一个子节点，那么保证该子节点为左子节点。

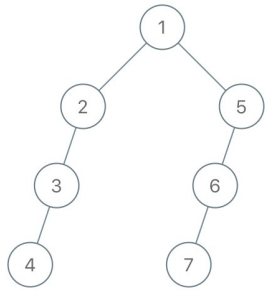
给出遍历输出 `S`，还原树并返回其根节点 `root`。

示例 1：



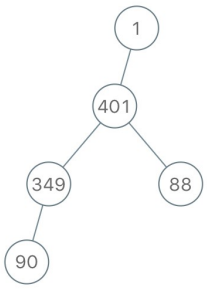

```
输入："1-2--3--4-5--6--7"
输出：[1,2,5,3,4,6,7]
```

示例 2：




```
输入："1-2--3---4-5--6---7"
输出：[1,2,5,3,null,6,null,4,null,7]
```

示例 3：




```
输入："1-401--349---90--88"
输出：[1,401,null,349,88,90]
```

提示：

- 原始树中的节点数介于 1 和 1000 之间。
- 每个节点的值介于 1 和 10 ^ 9 之间。

```
#define N 1002

struct TreeNode *nodeStack[N];
int nodeStackEnd = 0;
int levelStack[N];
int levelStackEnd = 0;

// 返回解析出来的 number, level, 和下一次开始的位置
int parse(char *S, int i, int *level, int *next)
{
    int l = 0;
    int num = 0;
    int j;
    int n = strlen(S);
    j = i;
    while (j < n && S[j] == '-') {
        l++;
        j++;
    }
    while (j < n && S[j] >= '0' && S[j] <= '9') {
        num = num * 10 + S[j] - '0';
        j++;
    }
    (*next) = j;
    (*level) = l;
    return num;
}


struct TreeNode* recoverFromPreorder(char * S){
    int i = 0;
    int j;
    int d; // 短横线的数量
    int next;
    int num;
    struct TreeNode *root;
    struct TreeNode *newNode;
    struct TreeNode *curNode;
    int number;
    int level = 0;
    int curLevel = 0;
    int n = strlen(S);
    if (S == NULL || n == 0) {
        return NULL;
    }
    j = 0;
    num = 0;
    while (j < n && S[j] >= '0' && S[j] <= '9') {
        num = num * 10 + S[j] - '0';
        j++;
    }
    root = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    root->left = NULL;
    root->right = NULL;
    root->val = num;
    if (n == j) {
       return root;
    }
    nodeStack[nodeStackEnd++] = root;
    levelStack[levelStackEnd++] = 0;
    curLevel = 0;
    for (i = j; i < strlen(S); i++) {
        level = 0;
        next = 0;
        number = parse(S, i, &level, &next);
        if (level == curLevel + 1) {
            // 说明是栈顶元素的左子树
            newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
            newNode->val = number;
            newNode->left = NULL;
            newNode->right = NULL;
            nodeStack[nodeStackEnd - 1]->left = newNode;
            nodeStack[nodeStackEnd++] = newNode;
            levelStack[levelStackEnd++] = level;
```

```
        } else if (level < curLevel + 1) {
            while (levelStackEnd > 0 && levelStack[levelStackEnd - 1] >= level) {
                levelStackEnd--;
                nodeStackEnd--;
            }
            curNode = nodeStack[nodeStackEnd - 1];
            newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
            newNode->left = NULL;
            newNode->right = NULL;
            newNode->val = number;
            curNode->right = newNode;
            nodeStack[nodeStackEnd++] = newNode;
            levelStack[levelStackEnd++] = level;
        } else {

        }
        i = next - 1;
        curLevel = level;
    }
    return root;
}
```

1031. 两个非重叠子数组的最大和

给出非负整数数组 `A` ，返回两个非重叠（连续）子数组中元素的最大和，子数组的长度分别为 `L` 和 `M`。（这里需要澄清的是，长为 L 的子数组可以出现在长为 M 的子数组之前或之后。）

从形式上看，返回最大的 `V`，而 `V = (A[i] + A[i+1] + ... + A[i+L-1]) + (A[j] + A[j+1] + ... + A[j+M-1])` 并满足下列条件之一：

- `0 <= i < i + L - 1 < j < j + M - 1 < A.length`，**或**
- `0 <= j < j + M - 1 < i < i + L - 1 < A.length`.

示例 1：

```
输入：A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2
输出：20
解释：子数组的一种选择中，[9] 长度为 1，[6,5] 长度为 2。
```

示例 2：

```
输入：A = [3,8,1,3,2,1,8,9,0], L = 3, M = 2
输出：29
解释：子数组的一种选择中，[3,8,1] 长度为 3，[8,9] 长度为 2。
```

示例 3：

```
输入：A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3
输出：31
解释：子数组的一种选择中，[5,6,0,9] 长度为 4，[0,3,8] 长度为 3。
```

提示：

1. `L >= 1`
2. `M >= 1`
3. `L + M <= A.length <= 1000`
4. `0 <= A[i] <= 1000`

先求出数组的前缀和，然后固定一个数组， 求另一个数组的最大和， 遍历每一个固定的数组，即可找出所有最大的解，最后取最大的那个。 注意边界条件的确定。

```
#define MAX(a, b) ((a) > (b)) ? (a) : (b)
int getMSum(int *A, int *aSum, int ASize, int i, int L, int M)
{
    int lefts = 0;
    int lefte = i - 1;
    int rights = i + L;
    int righte = ASize - 1;
    int maxv = -1;
    int j, tmp;

    if (lefte - lefts + 1 >= M) {
        for (j = lefts; j <= lefte - M + 1; j++) {
            tmp = aSum[j + M] - aSum[j];
            maxv = MAX(maxv, tmp);
        }
    }

    if (righte - rights + 1 >= M) {
        for (j = rights; j <= righte - M + 1; j++) {
            tmp = aSum[j + M] - aSum[j];
            maxv = MAX(maxv, tmp);
        }
    }
    return maxv;
}
int maxSumTwoNoOverlap(int* A, int ASize, int L, int M){
    int *aSum;
    int lSum = 0;
    int mSum = 0;
    int sum = 0;
    int ans = 0;
    int i, j;

    if (A == NULL || ASize == 0) {
        return 0;
    }

    /* 求前缀和 */
    aSum = (int*)malloc(sizeof(int) * (ASize + 1));
    aSum[0] = 0;
    for (i = 0; i < ASize; i++) {
        aSum[i + 1] = aSum[i] + A[i];
    }

    /* 确定L，寻找可能的最大的M */
    for (i = 0; i <= ASize - L; i++) {
        lSum = aSum[i + L] - aSum[i];
        mSum = getMSum(A, aSum, ASize, i, L, M);
        sum = lSum + mSum;
        ans = MAX(ans, sum);
    }
    return ans;
}
```

1169. 查询无效交易

如果出现下述两种情况，交易 **可能无效**：

- 交易金额超过 ¥1000
- 或者，它和另一个城市中同名的另一笔交易相隔不超过 60 分钟（包含 60 分钟整）

每个交易字符串 `transactions[i]` 由一些用逗号分隔的值组成，这些值分别表示交易的名称，时间（以分钟计），金额以及城市。

给你一份交易清单 `transactions`，返回可能无效的交易列表。你可以按任何顺序返回答案。

示例 1：

输入： transactions = ["alice,20,800,mtv","alice,50,100,beijing"]
输出： ["alice,20,800,mtv","alice,50,100,beijing"]
解释： 第一笔交易是无效的，因为第二笔交易和它间隔不超过 60 分钟、名称相同且发生在不同的城市。同样，第二笔交易也是无效的。

示例 2：

输入： transactions = ["alice,20,800,mtv","alice,50,1200,mtv"]
输出： ["alice,50,1200,mtv"]

示例 3：

输入： transactions = ["alice,20,800,mtv","bob,50,1200,mtv"]
输出： ["bob,50,1200,mtv"]

提示：

- transactions.length <= 1000
- 每笔交易 transactions[i] 按 "{name},{time},{amount},{city}" 的格式进行记录
- 每个交易名称 {name} 和城市 {city} 都由小写英文字母组成，长度在 1 到 10 之间
- 每个交易时间 {time} 由一些数字组成，表示一个 0 到 1000 之间的整数
- 每笔交易金额 {amount} 由一些数字组成，表示一个 0 到 2000 之间的整数

```
class Solution:
    def findInvalid(self, trans):
        n = len(trans)
        ans = set()
        for i in range(n-1):
            for j in range(i+1, n):
                a1, b1, c1, d1 = trans[i].split(",", 3)
                a2, b2, c2, d2 = trans[j].split(",", 3)
                if abs(int(b1) - int(b2)) <= 60 and d1 != d2:
                    if trans[i] not in ans:
                        ans.add(trans[i])
                    if trans[j] not in ans:
                        ans.add(trans[j])
        return ans

    def invalidTransactions(self, transactions):
        ans = set()
        dic = {}
        for line in transactions:
            a, b, c, d = line.split(",", 3)
            if a in dic:
                dic[a].append(line)
            else:
                dic[a] = [line]
            if int(c) > 1000:
                ans.add(line)
        #print(dic)
        for key, value in dic.items():
            invilds = self.findInvalid(value)
            if invilds:
                ans = ans | invilds
        return ans
```

1170. 比较字符串最小字母出现频次

我们来定义一个函数 f(s)，其中传入参数 s 是一个非空字符串；该函数的功能是统计 s 中（按字典序比较）最小字母的出现频次。

例如，若 s = "dcce"，那么 f(s) = 2，因为最小的字母是 "c"，它出现了 2 次。

现在，给你两个字符串数组待查表 queries 和词汇表 words，请你返回一个整数数组 answer 作为答案，其中每个 answer[i] 是满足 f(queries[i]) < f(W) 的词的数目，W 是词汇表 words 中的词。

示例 1：

输入： queries = ["cbd"], words = ["zaaaz"]
输出： [1]
解释： 查询 f("cbd") = 1，而 f("zaaaz") = 3 所以 f("cbd") < f("zaaaz")。

示例 2：

输入： queries = ["bbb","cc"], words = ["a","aa","aaa","aaaa"]
输出： [1,2]
解释： 第一个查询 f("bbb") < f("aaaa")，第二个查询 f("aaa") 和 f("aaaa") 都 > f("cc")。

提示：

- 1 <= queries.length <= 2000
- 1 <= words.length <= 2000
- 1 <= queries[i].length, words[i].length <= 10
- queries[i][j]，words[i][j] 都是小写英文字母

```
int f(char *s)
{
    int i;
    int n = strlen(s);
    int count = 1;
    char c = s[0];
    for (i = 1; i < n; i++) {
        if (s[i] < c) {
            c = s[i];
            count = 1;
        } else if (c == s[i]) {
            count++;
        }
    }
    return count;
}
int* numSmallerByFrequency(char ** queries, int queriesSize, char ** words, int wordsSize, int* returnSize){
    int i, j;
    int count;
    int n;
    int *wordCount;
    int ans;
    int *ansArray;
    wordCount = (int*)malloc(sizeof(int) * wordsSize);
    ansArray = (int*)malloc(sizeof(int) * queriesSize);
    for (i = 0; i < wordsSize; i++) {
        wordCount[i] = f(words[i]);
    }
    for (i = 0; i < queriesSize; i++) {
        count = f(queries[i]);
        ans = 0;
        for (j = 0; j < wordsSize; j++) {
            if (count < wordCount[j]) {
                ans++;
            }
        }
        ansArray[i] = ans;
    }
    (*returnSize) = queriesSize;
    free(wordCount);
    return ansArray;
}
```

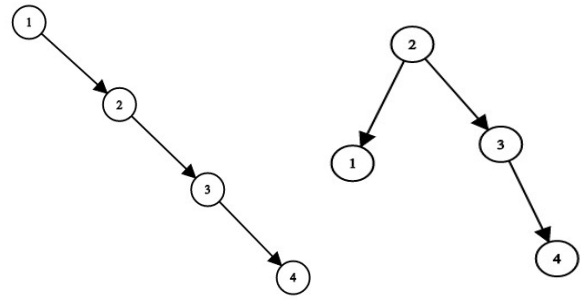
1382. 将二叉搜索树变平衡

给你一棵二叉搜索树，请你返回一棵 **平衡后** 的二叉搜索树，新生成的树应该与原来的树有着相同的节点值。

如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过 1 ，我们就称这棵二叉搜索树是 **平衡的** 。

如果有多种构造方法，请你返回任意一种。

示例：




```
输入：root = [1,null,2,null,3,null,4,null,null]
输出：[2,1,3,null,null,null,4]
解释：这不是唯一的正确答案，[3,1,4,null,2,null,null] 也是一个可行的构造方案。
```

提示：

- 树节点的数目在 1 到 10^4 之间。
- 树节点的值互不相同，且在 1 到 10^5 之间。

1. 先中序遍历，得到排序数组。
2. 依据排序数组，构造平衡二叉树

1224. 最大相等频率

给出一个正整数数组 nums，请你帮忙从该数组中找出能满足下面要求的 **最长** 前缀，并返回其长度：

- 从前缀中 **删除一个** 元素后，使得所剩下的每个数字的出现次数相同。

如果删除这个元素后没有剩余元素存在，仍可认为每个数字都具有相同的出现次数（也就是 0 次）。

示例 1：

```
输入：nums = [2,2,1,1,5,3,3,5]
输出：7
解释：对于长度为 7 的子数组 [2,2,1,1,5,3,3]，如果我们从中删去 nums[4]=5，就可以得到 [2,2,1,1,3,3]，里面每个数字都出现了两次。
```

示例 2：

```
输入：nums = [1,1,1,2,2,2,3,3,3,4,4,4,5]
输出：13
```

示例 3：

```
输入：nums = [1,1,1,2,2,2]
输出：5
```

示例 4：

```
输入：nums = [10,2,8,9,3,8,1,5,2,3,7,6]
输出：8
```

提示：

- 2 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^5

思路：

1. 维护两个map, 一个是num --> count, 记录每种数字出现的次数，一个是count --> num, 表示出现次数为count的数字都有哪些。
2. 遍历一遍数组，计算出两个map, 然后从后向前遍历一遍。
3. 对于每一个i, 检查nums[0:i+1]是否满足要求，如果满足，返回 i + 1, 如果不满足，删除nums[i], 更新两个map.

该思路有两个关键逻辑：

1. 如何通过两个map判断是否满足题目要求
2. 删除一个数字之后，如何更新两个map

```
class Solution(object):
    def maxEqualFreq(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        def check(num2count, count2num, current):
            """
            检查是否满足删除一个数字之后，剩下的数字出现次数都相同
            """
            # count 出现3种以上的情况，删除一个至少还有2种情况
            if len(count2num) > 2:
                return False
            # 如果所有的数字都出现一次，可以满足删除一个数字之后出现的次数相等
            if len(count2num) == 1 and count2num.keys()[0] == 1:
                return True
            # 如果所有数字都出现了相同次数，且只有一种数字，可以满足删除一个之后剩下的出现次数还一样
            if len(count2num) == 1 and len(num2count) == 1:
                return True
            # 否则，所有的数字出现次数相等且不是都出现了一次，无法满足删除一个数字之后出现的次数都相等
            if len(count2num) <= 1:
                return False
            key1, key2 = count2num.keys()
            if key1 > key2:
                key1, key2 = key2, key1
            # 如果只有两种出现频率，其中出现一次的只有一个数字，去掉该数字之后，剩下的出现频率都相等
            if len(count2num[key1]) == 1 and key1 == 1:
                return True
            # 如果只有两种出现频率且只差1， 出现频率大的只有一种数字，那么可以删除一个该数字，这样剩下的所有数字出现次数相同
            if key2-key1==1 and len(count2num[key2]) == 1:
                return True
            return False

        num2count = {}
        count2num = {}

        # 统计每种数字的出现次数
        for i, num in enumerate(nums):
            if num not in num2count:
                num2count[num] = 1
            else:
                num2count[num] += 1

        # 记录每种出现次数都有哪些数字
        for key, value in num2count.iteritems():
            if value not in count2num:
                count2num[value] = [key]
            else:
                count2num[value].append(key)

        # 检查是否符合要求，然后去掉 i
        for i in reversed(range(len(nums))):
            if check(num2count, count2num, nums[i]) == True:
                return i + 1
            # 删除 nums[i]，更新两个 map
            if nums[i] in num2count:
                count = num2count[ nums[i] ]
                if count > 1:
                    num2count[ nums[i] ] -= 1
                else:
                    del num2count[ nums[i] ]

                if len(count2num[count]) == 1:
                    del count2num[count]
                else:
                    count2num[count].remove(nums[i])
                if count > 1:
                    count = count - 1
                    if count not in count2num:
                        count2num[count] = [nums[i]]
                    else:
                        count2num[count].append(nums[i])
        return 0
```

1239. 串联字符串的最大长度

给定一个字符串数组 `arr`，字符串 `s` 是将 `arr` 某一子序列字符串连接所得的字符串，如果 `s` 中的每一个字符都只出现过一次，那么它就是一个可行解。

请返回所有可行解 `s` 中最长长度。

示例 1：

```
输入：arr = ["un","iq","ue"]
输出：4
解释：所有可能的串联组合是 "","un","iq","ue","uniq" 和 "ique"，最大长度为 4。
```

示例 2：

```
输入：arr = ["cha","r","act","ers"]
输出：6
解释：可能的解答有 "chaers" 和 "acters"。
```

示例 3：

```
输入：arr = ["abcdefghijklmnopqrstuvwxyz"]
输出：26
```

提示：

- `1 <= arr.length <= 16`
- `1 <= arr[i].length <= 26`
- `arr[i]` 中只含有小写英文字母

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
int size = 0;

// 字符串转换成数字
int char2int(char *string)
{
    int ans = 0;
    while (*string) {
        ans += 1 << (*string - 'a');
    }
    return ans;
}
// 返回把string中字母加入集合set中的结果
int add(char *string, int set)
{
    int ans = char2int(string);
    ans = ans | set;
    return ans;
}

// 返回string中的任意一个字母是否在set中出现过
int notCanPut(char* string, int set)
{
    int ans = char2int(string);
    ans = ans & set;
    return ans;
}
/*
 * arr[i] 当前检查的单词
 * currentSet : 当前加入的集合
 * oneAns : 深度搜索到底一趟的结果
 * ans : 保存最终的结果
 */
void dfs(char** arr, int i, int currentSet, int *oneAns, int *ans)
{
    // 如果到达了末尾，比较一趟的结果和最终的结果大小
    if (i == size) {
        *ans = MAX(*ans, *oneAns);
        return;
    }

    // 如果arr[i]无法放入，则搜索下一个单词
    if (notCanPut(arr[i], currentSet)) {
        dfs(arr, i+1, currentSet, oneAns, ans);
        return;
    }

    // 如果arr[i]可以放入，那么放入和不放入两种情况都需要递归
    int tmpOneAns = *oneAns + strlen(arr[i]);
    dfs(arr, i + 1, add(arr[i], currentSet), &tmpOneAns, ans);

    dfs(arr, i + 1, currentSet, oneAns, ans);

    return;
}
int maxLength(char ** arr, int arrSize){
    int ans = 0;
    int oneAns = 0;
    int currentSet = 0;

    size = arrSize;
    dfs(arr, 0, &oneAns, &ans);
    return ans;
}
```

面试题 01.01. 判定字符是否唯一

实现一个算法，确定一个字符串 s 的所有字符是否全都不同。

示例 1：

输入： s = "leetcode"
输出： false

示例 2：

输入： s = "abc"
输出： true

限制：

- 0 <= len(s) <= 100
- 如果你不使用额外的数据结构，会很加分。

使用哈希表记录出现的字符

```
#define MAX_CHAR 256
int map[MAX_CHAR];

bool isUnique(char* astr){
    memset(map, -1, sizeof(int) * MAX_CHAR);
    for (int i = 0; i < strlen(astr); i++) {
        if (map[astr[i]] != -1) {
            return false;
        }
        map[astr[i]] += 1;
    }
    return true;
}
```

如果不使用额外的数据结构，实际上使用bit位表示字符是否存在。

面试题 01.02. 判定是否互为字符重排

给定两个字符串 s1 和 s2，请编写一个程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。

示例 1：

输入： s1 = "abc", s2 = "bca"
输出： true

示例 2：

输入： s1 = "abc", s2 = "bad"
输出： false

说明：

- 0 <= len(s1) <= 100
- 0 <= len(s2) <= 100

第一种思路，排序之后逐个字符比较， 如果完全一样，则可以。

```
int compare(const void *p, const void *q)
{
    char a = *(char*)p;
    char b = *(char*)q;
    return a - b;
}
bool CheckPermutation(char* s1, char* s2){
    if (s1 == NULL && s2 == NULL) {
        return true;
    }
    if (s1 == NULL || s2 == NULL) {
        return false;
    }
    if (strlen(s1) != strlen(s2)) {
        return false;
    }
    qsort(s1, strlen(s1), sizeof(char), compare);
    qsort(s2, strlen(s2), sizeof(char), compare);
    for (int i = 0; i < strlen(s1); i++) {
        if (s1[i] != s2[i]) {
            return false;
        }
    }
    return true;
}
```

第二种思路，使用哈希表存储每个字符的出现次数

```
#define MAX_CHAR 256
int map[MAX_CHAR];

bool CheckPermutation(char* s1, char* s2){
    if (s1 == NULL && s2 == NULL) {
        return true;
    }
    if (s1 == NULL || s2 == NULL) {
        return false;
    }

    int i;
    memset(map, 0, sizeof(int) * MAX_CHAR);

    for (i = 0; i < strlen(s1); i++) {
        map[ s1[i] ] += 1;
    }
    for (i = 0; i < strlen(s2); i++) {
        map[ s2[i] ] -= 1;
    }
    for (i = 0; i < MAX_CHAR; i++) {
        if (map[i] != 0) {
            return false;
        }
    }
    return true;
}
```

面试题 01.03. URL化

URL化。编写一种方法，将字符串中的空格全部替换为 `%20` 。假定该字符串尾部有足够的空间存放新增字符，并且知道字符串的“真实”长度。（注：用 Java 实现的话，请使用字符数组实现，以便直接在数组上操作。）

示例1:

```
输入："Mr John Smith    ", 13
输出："Mr%20John%20Smith"
```

示例2:

输入："   ", 5
输出："%20%20%20%20%20"

提示：

1. 字符串长度在[0, 500000]范围内。

先统计出空格的数量，计算好移动的位置，从后向前移动，避免重复移动。

```
char* replaceSpaces(char* S, int length){
    int space = 0;
    int i, j, oldENd, end;

    // 统计空格数量
    for (i = length - 1; i >= 0; i--) {
        if (S[i] == ' ') {
            space++;
        }
    }

    // 计算新字符串的结尾
    end = length + space * 2;
    S[end] = '\0';
    i = length - 1;
    oldENd = length - 1;
    while (i >= 0) {
        if (S[i] == ' ') {
            //一步移动到位，避免重复移动
            for (j = oldENd; j > i; j--) {
                S[--end] = S[j];
            }
            S[--end] = '0';
            S[--end] = '2';
            S[--end] = '%';
            oldENd = i - 1;
        }
        i--;
    }
    return S;
}
```

优化逻辑，更易理解，只用一层循环

```
char* replaceSpaces(char* S, int length){
    int space = 0;
    int i, end;

    // 统计空格数量
    for (i = length - 1; i >= 0; i--) {
        if (S[i] == ' ') {
            space++;
        }
    }

    // 计算新字符串的结尾
    end = length + space * 2;
    S[end--] = '\0';
    for(i = length - 1; i >= 0; i--) {
        if (S[i] != ' ') {
            S[end--] = S[i];
        } else {
            S[end--] = '0';
            S[end--] = '2';
            S[end--] = '%';
        }
    }
    return S;
}
```

面试题 02.08. 环路检测

给定一个有环链表，实现一个算法返回环路的开头节点。
有环链表的定义：在链表中某个节点的next元素指向在它前面出现过的节点，则表明该链表存在环路。

示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```

示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```

示例 3：

```
输入：head = [1], pos = -1
输出：no cycle
解释：链表中没有环。
```

进阶：
你是否可以不用额外空间解决此题？

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode *detectCycle(struct ListNode *head) {
    if (head == NULL || head->next == NULL || head->next->next == NULL) {
        return NULL;
    }

    struct ListNode *slow = head->next;
    struct ListNode *quick = head->next->next;

    // 快慢指针直到两个指针相遇
    while (slow != NULL && quick != NULL && slow != quick) {
        if (slow->next == NULL) return NULL;
        if (quick->next == NULL) return NULL;
        if (quick->next->next == NULL) return NULL;
        slow = slow->next;
        quick = quick->next->next;
    }

    // 找到入口的位置
    slow = head;
    while (slow != quick) {
        slow = slow->next;
        quick = quick->next;
    }
    return slow;
}
```

面试题 01.08. 零矩阵

编写一种算法，若M × N矩阵中某个元素为0，则将其所在的行与列清零。

示例 1：

```
输入：
[
  [1,1,1],
  [1,0,1],
  [1,1,1]
]
输出：
[
  [1,0,1],
  [0,0,0],
  [1,0,1]
]
```

示例 2：

```
输入：
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
输出：
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]
```

先记录需要清零的行和列，再清除

```
#define M 1000
#define N 1000

void setZeroes(int** matrix, int matrixSize, int* matrixColSize){
    int rows[M];
    int rowsEnd = 0;
    int cols[N];
    int colsEnd = 0;
    int i, j;

    // 统计需要清零的行号和列号
    for (i = 0; i < matrixSize; i++) {
        for (j = 0; j < matrixColSize[i]; j++) {
            if (matrix[i][j] == 0) {
                rows[rowsEnd++] = i;
                cols[colsEnd++] = j;
            }
        }
    }

    // 根据统计结果执行清零操作
    for (i = 0; i < rowsEnd; i++) {
        for (j = 0; j < matrixColSize[ rows[i] ]; j++) {
            matrix[ rows[i] ][j] = 0;
        }
    }
    for (i = 0; i < colsEnd; i++) {
        for (j = 0; j < matrixSize; j++) {
            matrix[j][ cols[i] ] = 0;
        }
    }
    return;
}
```

不使用额外的空间存储行和列的信息

```
// 不使用额外的空间
void setZeroes(int** matrix, int matrixSize, int* matrixColSize){
    int i, j;
    int delFirstRow = 0; // 标记第一行是否应该清0
    int delRow = 0; // 标记该行是否应该清0

    // 第一行是否应该清0
    for (j = 0; j < matrixColSize[0]; j++) {
        if (matrix[0][j] == 0) {
            delFirstRow = 1;
            break;
        }
    }

    // 统计每一行需要清0的列，记录在第一行，然后清零该行
    for (i = 1; i < matrixSize; i++) {
        for (j = 0; j < matrixColSize[i]; j++) {
            if (matrix[i][j] == 0) {
                delRow = 1;
                matrix[0][j] = 0;
            }
        }
        if (delRow) {
            memset(matrix[i], 0, sizeof(int) * matrixColSize[i]);
            delRow = 0;
        }
    }

    // 根据第一行清0列
    for (j = 0; j < matrixColSize[0]; j++) {
        if (matrix[0][j] == 0) {
            for (i = 1; i < matrixSize; i++) {
                matrix[i][j] = 0;
            }
        }
    }

    // 根据最开始统计的标记决定第一行是否清0
    if (delFirstRow) {
        memset(matrix[0], 0, sizeof(int) * matrixColSize[0]);
    }

    return;
}
```

面试题 01.05. 一次编辑

字符串有三种编辑操作:插入一个字符、删除一个字符或者替换一个字符。 给定两个字符串，编写一个函数判定它们是否只需要一次(或者零次)编辑。

示例 1:

```
输入:
first = "pale"
second = "ple"
输出: True
```

示例 2:

```
输入:
first = "pales"
second = "pal"
输出: False
```

按照两个字符串的长度分3中情况讨论

```
bool canReach(char* first, char *second)
{
    int i = 0;
    int j = 0;
    while (i < strlen(first) && j < strlen(second) && first[i] == second[j]) {
        i++;
        j++;
    }
    i = i + 1; //跳过一个字符
    while (i < strlen(first) && j < strlen(second)) {
        if (first[i] != second[j]) {
            return false;
        }
        i++;
        j++;
    }
    return true;
}
bool oneEditAway(char* first, char* second){
    if (first == NULL || second == NULL) return true;

    int n1 = strlen(first);
    int n2 = strlen(second);
    int i,j,num;

    if (n1 == n2) {
        num = 0;
        for (i = 0; i < n1; i++) {
            if (first[i] != second[i]) {
                num++;
            }
        }
        if (num <= 1) {
            return true;
        } else {
            return false;
        }
    } else if (n1 - n2 == 1) {
        return canReach(first, second);
    } else if (n2 - n1 == 1) {
        return canReach(second, first);
    } else {
        return false;
    }
}
```