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
```

using c:

```
int map[256];
void init()
{
    for (int i = 0; i < 256; i++) map[i] = -1;
}
int max(int a, int b)
{
    if (a > b) return a;
    return b;
}
int lengthOfLongestSubstring(char * s){
    if (s == NULL) return 0;
    init();
    const int n = strlen(s);
    int start = 0;
    int end = 0;
    int ans = 0;
    while (end < n) {
        if (map[ s[end] ] == -1) {
            map[ s[end] ] = end;
        } else {
            ans = max(ans, end - start);
            int index = map[ s[end] ];
            for (int j = start; j < index; j++) {
                map[ s[j] ] = -1;
            }
            start = index + 1;
            map[ s[end] ] = end;
            //ans = max(ans, end - start + 1);
        }
        //printf("%d, %d, %d\n", start, end, ans);
        end++;
    }
    //printf("ans :%d\n", ans);
    ans = max(ans, end - start);
    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。

---

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：[]
```

```c
#define MAX_ANS 1000
#define true 1
#define false 0

char **wordsGlobal = NULL;
int wordsSizeGlobal = 0;
int *wordsMap = NULL; // 保存words中的单词出现的次数
int wordsMapSize = 0; // 记录wordsMap中存有多少个单词

char *substr(char *s, int start, int len)
{
    int end = 0;
    char *newString = (char*)malloc(sizeof(char) * (len + 1));
    for (int i = start; i < start + len; i++) {
        newString[end++] = s[i];
    }
    newString[end] = '\0';
    return newString;
}

// indexOf 返回word 在 words 中的索引，没有返回-1
int indexOf(char *word)
{
    for (int i = 0; i < wordsSizeGlobal; i++) {
        if (strcmp(wordsGlobal[i], word) == 0 && wordsMap[i] == 1) {
            return i;
        }
    }

    return -1;
}

// 是否是一个拼接
// 从 start 开始，每m个字符是一个单词，一共有k个单词
int isSatisfied(char *s, int start, int m, int k)
{
    wordsMapSize = wordsSizeGlobal;
    for (int i = 0; i < wordsSizeGlobal; i++) {
        wordsMap[i] = 1;
    }

    for (int i = 0; i < k; i++) {
        char *word = substr(s, start + m * i, m);
        int index = indexOf(word);
        if (index != -1) {
            wordsMap[index] -= 1;
            wordsMapSize--;
        } else {
            return false;
        }
    }

    if (wordsMapSize == 0) {
        return true;
    } else {
        return false;
    }
}

int* findSubstring(char * s, char ** words, int wordsSize, int* returnSize)
{
    if (s == NULL || words == NULL || wordsSize <= 0) {
        *returnSize = 0;
        return NULL;
    }

    int slen = strlen(s);
    int wlen = strlen(words[0]);
    if (slen < wlen * wordsSize) {
        *returnSize = 0;
        return NULL;
    }

    int *ans = (int*)malloc(sizeof(int) * MAX_ANS);
```

```
    int ansEnd = 0;
    int result;
    int m = strlen(words[0]);
    int n = m * wordsSize;

    wordsGlobal = words;
    wordsSizeGlobal = wordsSize;

    wordsMap = (int*)malloc(sizeof(int) * wordsSize);
    wordsMapSize = wordsSize;

    for (int i = 0; i <= strlen(s) - n; i++) {
        result = isSatisfied(s, i, m, wordsSize);
        if (result) {
            ans[ansEnd++] = i;
        }
    }

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

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]
]
```

```c
#define MAX_ONE_LINE_SIZE 1000
#define MAX_ANS_SIZE 1000

const int *candidateGlobal = NULL; // 指向数组
int nGlobal = 0;
int **lines; // 指向最终的结果
int linesEnd = 0;
int *columnsSize; // 指向一个数组，表示每行元素的个数
int columnsSizeEnd = 0;

// 实现排序比较函数
int compare(const void *p, const void *q)
{
    int a = *(int*)p;
    int b = *(int*)q;
    return a - b;
}

// 深度拷贝一个整数数组，在函数中开辟空间返回
int *deepCopy(int *array, int size)
{
    int *newArray = (int*)malloc(sizeof(int) * size);
    memcpy(newArray, array, sizeof(int) * size);
    return newArray;
}

// 从第i个数字开始深度优先搜索，一趟结果放在line中，最终结果放在lines中
void dfs(int i, int target, int *line, int *lineEnd)
{
    if (target == 0) {
        lines[linesEnd++] = deepCopy(line, *lineEnd);
        columnsSize[columnsSizeEnd++] = *lineEnd;
        return;
    }

    for (int j = i; j < nGlobal; j++) {
        if (candidateGlobal[j] > target) {
            break;
        }
        line[(*lineEnd)++] = candidateGlobal[j];
        dfs(j, target - candidateGlobal[j], line, lineEnd);
        (*lineEnd)--;
    }
}



int** combinationSum(int* candidates, int candidatesSize, int target, int* returnSize, int** returnColumnSizes)
{
    // 存储每一趟扫描的结果
    int *line = (int*)malloc(sizeof(int) * MAX_ONE_LINE_SIZE);
    int lineEnd = 0;

    // 存储最终的结果
    lines = (int**)malloc(sizeof(int*) * MAX_ANS_SIZE);
    linesEnd = 0;

    // 存储每行的大小
    columnsSize = (int*)malloc(sizeof(int) * MAX_ANS_SIZE);
    columnsSizeEnd = 0;

    candidateGlobal = candidates;
    nGlobal = candidatesSize;

    qsort(candidateGlobal, nGlobal, sizeof(int), compare);

    dfs(0, target, line, &lineEnd);

    *returnSize = linesEnd;
    *returnColumnSizes = columnsSize;

    // line 用来存储每趟扫描的结果，是临时变量，不使用应该释放
    free(line);
```

```
    // 其他申请的空间都是最后答案的一部分，无须释放
    return lines;
}
```

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]
]
```

```c
#define MAX_ONE_LINE 1000
#define MAX_ANS_SIZE 1000

int **lines;
int linesEnd = 0;
int *columnsSize;
int columnsSizeEnd = 0;
int *arrayGlobal;
int n;

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

int *deepCopy(int *array, int size)
{
    int *newArray = (int*)malloc(sizeof(int) * size);
    memcpy(newArray, array, sizeof(int) * size);
    return newArray;
}

void dfs(int i, int target, int *line, int *lineEnd)
{
    // 如果得到结果，存储
    if (target == 0) {
        lines[linesEnd++] = deepCopy(line, *lineEnd);
        columnsSize[columnsSizeEnd++] = *lineEnd;
        return;
    }

    for (int j = i + 1; j < n; j++) {
        if (j > i + 1 && arrayGlobal[j] == arrayGlobal[j-1]) {
            continue;
        }
        if (arrayGlobal[j] > target) {
            break;
        }

        line[(*lineEnd)++] = arrayGlobal[j];
        dfs(j, target - arrayGlobal[j], line, lineEnd);
        (*lineEnd)--;
    }

    return;
}

int** combinationSum2(int* candidates, int candidatesSize, int target, int* returnSize, int** returnColumnSizes)
{
    // 存储一趟搜索结果
    int *line = (int*)malloc(sizeof(int) * MAX_ONE_LINE);
    int lineEnd = 0;

    // 存储最终搜索结果
    lines = (int**)malloc(sizeof(int*) * MAX_ANS_SIZE);
    linesEnd = 0;

    // 存储每一行的大小
    columnsSize = (int*)malloc(sizeof(int*) * MAX_ANS_SIZE);
    columnsSizeEnd = 0;

    arrayGlobal = candidates;
    n = candidatesSize;
    qsort(arrayGlobal, n, sizeof(int), compare);

    // 以每一个为起点深度搜索， 跳过重复的开始节点
    for (int i = 0; i < n; i++) {
        if (i > 0 && arrayGlobal[i] == arrayGlobal[i-1]) {
            continue;
        }
        line[lineEnd++] = arrayGlobal[i];
        dfs(i, target - arrayGlobal[i], line, &lineEnd);
```

```
        lineEnd--;
    }

    *returnSize = linesEnd;
    *returnColumnSizes = columnsSize;

    free(line);

    return lines;
}
```

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。
```

```c
#define MAX_ANS_SIZE 1000
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int compare(const void *p, const void *q)
{
    int *a = *(int**)p;
    int *b = *(int**)q;
    return a[0] - b[0];
}
int** merge(int** intervals, int intervalsSize, int* intervalsColSize, int* returnSize, int** returnColumnSizes)
{
    int *columnsSize = (int*)malloc(sizeof(int) * MAX_ANS_SIZE);
    if (intervals == NULL || intervalsColSize == NULL || intervalsSize <= 0) {
        *returnColumnSizes = columnsSize;
        *returnSize = 0;
        return NULL;
    }

    qsort(intervals, intervalsSize, sizeof(int) * 2, compare);

    // 用label标记区间是否留下来
    int start = 0;
    int *label = (int*)malloc(sizeof(int) * intervalsSize);
    for (int i = 0; i < intervalsSize; i++) {
        label[i] = 1;
    }

    // 比较相邻的区间，如果有重叠，就合并区间
    while (start < intervalsSize - 1) {
        if (intervals[start][1] >= intervals[start + 1][0]) {
            intervals[start + 1][0] = intervals[start][0];
            intervals[start + 1][1] = MAX(intervals[start][1], intervals[start + 1][1]);
            label[start] = 0;
        }
        start++;
    }

    // 统计合并之后剩下多少区间
    int count = 0;
    for (int i = 0; i < intervalsSize; i++) {
        if (label[i] == 1) {
            count++;
        }
    }

    // 组装最后的结果
    int **ans = (int**)malloc(sizeof(int*) * count);
    int ansEnd = 0;
    *returnSize = count;
    for (int i = 0; i < count; i++) {
        ans[i] = (int*)malloc(sizeof(int) * 2);
    }
    for (int i = 0; i < intervalsSize; i++) {
        if (label[i] == 1) {
            ans[ansEnd][0] = intervals[i][0];
            ans[ansEnd][1] = intervals[i][1];
            ansEnd++;
        }
    }
    for (int i = 0; i < count; i++) {
        columnsSize[i] = 2;
    }
    *returnColumnSizes = columnsSize;

    // 释放无用的空间
    free(label);

    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
```

```
bool searchMatrix(int** matrix, int matrixSize, int* matrixColSize, int target)
{
    if (matrix == NULL || matrixColSize == NULL || matrixSize <= 0) {
        return false;
    }

    if (matrixSize == 1 && matrixColSize[0] <= 0) {
        return false;
    }

    // 首先使用二分查找查第一列，确定target应该在哪一行
    int left = 0;
    int right = matrixSize - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (target == matrix[mid][0]) {
            return true;
        } else if (target < matrix[mid][0]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    int line = left <= 0 ? 0 : left - 1;
    left = 0, right = matrixColSize[line] - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (target == matrix[line][mid]) {
            return true;
        } else if (target < matrix[line][mid]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return false;
}
```

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;
}
```

使用动态规划

```
int numDecodings(char * s){
    int ans = 0;
    int pre1 = 1; // 当前元素前一个
    int pre2 = 1; // 当前元素前二个
    int count1, count2, number;

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

    // 开头就是0，无法解码
    if (s[0] == '0') {
        return 0;
    }

    for (int i = 1; i < strlen(s); i++) {
        // [1] if s[i] in [1, 9] dp[i-1] * 1 else 0
        // [2] if s[i-1], s[i] int [10,26], dp[i-2] * 1 else 0
        // final = [1] + [2]
        if (s[i] >= '1' && s[i] <= '9') {
            count1 = pre1;
        } else {
            count1 = 0;
        }

        number = (s[i-1] - '0') * 10 + (s[i] - '0');
        if (number >= 10 && number <= 26) {
            count2 = pre2;
        } else {
            count2 = 0;
        }

        pre2 = pre1;
        pre1 = count1 + count2;
    }

    return pre1;
}
```

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]
]
```

使用一个数组模拟双端队列，缺点是浪费点空间，好处是简单清晰。

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


#define MAX_ROWS 900
#define MAX_COLS 900


typedef struct TreeNode TreeNode;
typedef struct TreeNode* TreeNodePtr;

int** levelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes)
{
    int **ans = (int**)malloc(sizeof(int*) * MAX_ROWS);
    int ansEnd = 0;
    int *colsSize = (int*)malloc(sizeof(int) * MAX_ROWS);
    int colsEnd = 0;

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

    int *line;
    TreeNodePtr *deque = (TreeNodePtr*)malloc(sizeof(TreeNode*) * MAX_COLS * MAX_ROWS);
    int dequeStart = 0;
    int dequeEnd = 0;

    // 根节点放入结果中
    line = (int*)malloc(sizeof(int));
    line[0] = root->val;
    ans[ansEnd++] = line;
    colsSize[colsEnd++] = 1;

    // 根节点放入队列中
    deque[dequeEnd++] = root;

    int count = 1;
    while (dequeEnd - dequeStart > 0) {
        TreeNode* topNode = deque[dequeStart++]; // 取出队首节点
        if (topNode->left) {
            deque[dequeEnd++] = topNode->left;
        }
        if (topNode->right) {
            deque[dequeEnd++] = topNode->right;
        }
        count--;
        if (count == 0 && dequeEnd - dequeStart > 0) {
            int lineEnd = 0;
            line = (int*)malloc(sizeof(int) *(dequeEnd - dequeStart));
            colsSize[colsEnd++] = dequeEnd - dequeStart;
            for (int k = dequeStart; k < dequeEnd; k++) {
                line[lineEnd++] = deque[k]->val;
            }
            ans[ansEnd++] = line;
            count = dequeEnd - dequeStart;
        }
    }

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

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"]
]
```

```c
#define MAX_ANS 1000

char ***ans;
int ansEnd;
int *colsSize = NULL;
int colsEnd = 0;
char *sg;
int size;

// 释放字符串数组所占用的空间
void freeStrings(char **strings, int size)
{
    for (int i = 0; i < size; i++) {
        free(strings[i]);
    }
    free(strings);
}

// 深度拷贝 string[start : end]
char *strDeepCopy(char *string, int start, int end)
{
    char *newString = (char*)malloc(sizeof(char) * (end - start + 2));
    int newStringEnd = 0;

    for (int i = start; i <= end; i++) {
        newString[newStringEnd++] = string[i];
    }
    newString[newStringEnd] = '\0';

    return newString;
}

// 深度拷贝一个字符串数组
char **deepCopy(char **strings, int size)
{
    char **ans = (char**)malloc(sizeof(char*) * size);
    for (int i = 0; i < size; i++) {
        ans[i] = strDeepCopy(strings[i], 0, strlen(strings[i]) - 1);
    }
    return ans;
}

int isHuiWen(char *string, int start, int end)
{
    while(start <= end) {
        if (sg[start] != string[end]) {
            return false;
        }
        start++, end--;
    }
    return true;
}

void dfs(int start, int end, char **line, int *lineEnd)
{
    if (end == size) {
        if (start == end) {
            ans[ansEnd++] = deepCopy(line, *lineEnd);
            colsSize[colsEnd++] = *lineEnd;
        }
        return;
    }

    // 如果[start, end] 是回文，从这里切割一下
    if (isHuiWen(sg, start, end)) {
        line[(*lineEnd)++] = strDeepCopy(sg, start, end);
        dfs(end+1, end+1, line, lineEnd);
        (*lineEnd)--;
    }

    // 不切割，继续向下搜索
    dfs(start, end+1, line, lineEnd);
    return;
}
```

```
char *** partition(char * s, int* returnSize, int** returnColumnSizes)
{
    ans = NULL;
    ansEnd = 0;
    colsSize = NULL;
    colsEnd = 0;
    sg = NULL;

    if (s == NULL || strlen(s) == 0) {
        *returnSize = 0;
        *returnColumnSizes = colsSize;
        return ans;
    }

    size = strlen(s);
    char **line = (char**)malloc(sizeof(char*) * size);
    int lineEnd = 0;
    ans = (char***)malloc(sizeof(char**) * MAX_ANS);
    colsSize = (int*)malloc(sizeof(int) * MAX_ANS);

    sg = s;
    dfs(0, 0, line, &lineEnd);

    *returnSize = ansEnd;
    *returnColumnSizes = colsSize;

    freeStrings(line, lineEnd);

    return 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;
}
```

使用库函数少写很多代码

```
#define MAX_N 1000
#define MAX_ANS 10000

// 题目要求如果是0返回单独的一个0，而不是一串0
void adjustResult(char *string, int size)
{
    int zeroFlag = true;
    for (int i = 0; i < size; i++) {
        if (string[i] != '0') {
            zeroFlag = false;
            break;
        }
    }
    if (zeroFlag) {
        string[0] = '0';
        string[1] = '\0';
    }
    return;
}

char *append(const char *s1, const char *s2)
{
    char *newString = (char*)malloc(sizeof(char) * (strlen(s1) + strlen(s2) + 1));
    int newStringEnd = 0;
    for (int i = 0; i < strlen(s1); i++) {
        newString[newStringEnd++] = s1[i];
    }
    for (int i = 0; i < strlen(s2); i++) {
        newString[newStringEnd++] = s2[i];
    }
    newString[newStringEnd] = '\0';
    return newString;
}
char *int2str(int number)
{
    char *string = (char*)malloc(sizeof(char) * MAX_N);
    int end = 0;

    if (number == 0) {
        string[0] = '0';
        string[1] = '\0';
        return string;
    }

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

    int left = 0, right = end - 1;
    while (left <= right) {
        int tmp = string[left];
        string[left] = string[right];
        string[right] = tmp;
        left++, right--;
    }

    string[end] = '\0';
    return string;
}
// 在string的末尾拼接上number
void addNumber(char *string, int *end, int number)
{
    char *numberString = int2str(number);
    for (int i = 0; i < strlen(numberString); i++) {
        string[(*end)++] = numberString[i];
    }
    return;
}

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

```
    char *aString = int2str(a);
    char *bString = int2str(b);
    char *abString = append(aString, bString);
    char *baString = append(bString, aString);
    int n = strlen(abString);

    for (int i = 0; i < n; i++) {
        if (abString[i] > baString[i]) {
            return -1;
        } else if (abString[i] < baString[i]) {
            return 1;
        }
    }

    return 0;
}

char * largestNumber(int* nums, int numsSize){
    char *numString = NULL;
    int end = 0;

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

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

    numString = (char*)malloc(sizeof(char) * MAX_ANS);
    for (int i = 0; i < numsSize; i++) {
        addNumber(numString, &end, nums[i]);
    }

    numString[end] = '\0';

    adjustResult(numString, end);

    return numString;
}
```

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"]
```

```c
#define N 10

struct HashTable {
    char key[N + 1]; // N个字母
    int count; // 出现的次数
    UT_hash_handle hh;
};

typedef struct HashTable Map;

char *deepCopy(char *string, int start, int end)
{
    char *newString = (char*)malloc(sizeof(char) * (end - start + 1));
    int newStringEnd = 0;
    for (int i = start; i < end; i++) {
        newString[newStringEnd++] = string[i];
    }
    newString[newStringEnd] = '\0';
    return newString;
}

char ** findRepeatedDnaSequences(char * s, int* returnSize)
{
    char **ans = NULL;
    *returnSize = 0;

    if (s == NULL || strlen(s) <= N) {
        return ans;
    }

    Map *map = NULL; // 哈希表
    Map *current = NULL; // 哈希表中的当前元素
    for (int i = 0; i <= strlen(s) - N; i++) {
        char *key = deepCopy(s, i, i + N);
        HASH_FIND_STR(map, key, current);
        if (current == NULL) {
            current = (Map*)malloc(sizeof(Map));
            strncpy(current->key, key, N); current->key[N] = '\0';
            current->count = 1;
            HASH_ADD_STR(map, key, current);
        } else {
            current->count++;
        }
    }

    // 统计结果的数量
    int ansCount = 0;
    for(current = map; current != NULL; current = current->hh.next) {
        if (current->count > 1) {
            ansCount++;
        }
    }

    // 根据结果数量开辟空间存储结果
    ans = (char**)malloc(sizeof(char*) * ansCount);
    int ansEnd = 0;
    for(current = map; current != NULL; current = current->hh.next) {
        if (current->count > 1) {
            ans[ansEnd++] = deepCopy(current->key, 0, strlen(current->key));
        }
    }

    // 释放掉map所占用的空间
    Map *tmp;
    HASH_ITER(hh, map, current, tmp) {
      HASH_DEL(map, current);
      free(current);
    }

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

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。

每个非0的数字只移动一次位置

```
void moveZeroes(int* nums, int numsSize)
{
    int zeroCount = 0;
    for (int i = 0; i < numsSize; i++) {
        if (nums[i] == 0) {
            zeroCount++;
        } else {
            nums[i - zeroCount] = nums[i];
        }
    }
    for (int i = numsSize - 1; i >= numsSize - zeroCount; i--) {
        nums[i] = 0;
    }

    return;
}
```

---

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;
}
```

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;
}
```

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`

排序求前K个。

```
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;
}
```

1004. 最大连续1的个数 III

给定一个由若干 `0` 和 `1` 组成的数组 `A`，我们最多可以将 `K` 个值从 0 变成 1 。

返回仅包含 1 的最长（连续）子数组的长度。

示例 1：

输入： `A = [1,1,1,0,0,0,1,1,1,1,0], K = 2`
输出： `6`
解释：
`[1,1,1,0,0,`**`1`**`,1,1,1,1,`**`1`**`]`
粗体数字从 `0` 翻转到 `1`，最长的子数组长度为 `6`。

示例 2：

输入： `A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3`
输出： `10`
解释：
`[0,0,1,1,`**`1`**`,`**`1`**`,1,1,1,`**`1`**`,1,1,0,0,0,1,1,1,1]`
粗体数字从 `0` 翻转到 `1`，最长的子数组长度为 `10`。

提示：

1. `1 <= A.length <= 20000`
2. `0 <= K <= A.length`
3. `A[i]` 为 `0` 或 `1`

```
#define max(a, b) ((a) > (b) ? (a) : (b))
int longestOnes(int* A, int ASize, int K){
    int start = 0;
    int ans = 0;
    int max_count = 0;
    // [start, end] 内有 max_count 个1
    for(int end = 0; end < ASize; end++) {
        if (A[end] == 1) {
            max_count += 1;
        }
        // 当0的个数大于K的个数的时候，向右滑动
        if (end - start + 1 - max_count > K) {
            if (A[start] == 1) {
                max_count -= 1;
            }
            start++;
        }
        ans = max(ans, end - start + 1);
    }
    return ans;
}
```

1313. 解压缩编码列表

给你一个以行程长度编码压缩的整数列表 `nums` 。

考虑每对相邻的两个元素 `freq, val] = [nums[2*i], nums[2*i+1]]` （其中 `i >= 0` ），每一对都表示解压后子列表中有 `freq` 个值为 `val` 的元素，你需要从左到右连接所有子列表以生成解压后的列表。

请你返回解压后的列表。

示例：

输入： nums = [1,2,3,4]
输出： [2,4,4,4]
解释： 第一对 [1,2] 代表着 2 的出现频次为 1，所以生成数组 [2]。
第二对 [3,4] 代表着 4 的出现频次为 3，所以生成数组 [4,4,4]。
最后将它们串联到一起 [2] + [4,4,4] = [2,4,4,4]。

示例 2：

输入： nums = [1,1,2,3]
输出： [1,3,3]

提示：

- 2 <= nums.length <= 100
- nums.length % 2 == 0
- 1 <= nums[i] <= 100

```
#define MAX_ANS 10001

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

    int *ans = (int*)malloc(sizeof(int) * MAX_ANS);
    int ansEnd = 0;
    for (int i = 0; i < numsSize - 1; i = i + 2) {
        for (int j = 0; j < nums[i]; j++) {
            ans[ansEnd++] = nums[i+1];
        }
    }

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

1314. 矩阵区域和

给你一个 m * n 的矩阵 mat 和一个整数 K ，请你返回一个矩阵 answer ，其中每个 answer[i][j] 是所有满足下述条件的元素 mat[r][c] 的和:

- i - K <= r <= i + K, j - K <= c <= j + K
- (r, c) 在矩阵内。

示例 1：

输入： mat = [[1,2,3],[4,5,6],[7,8,9]], K = 1
输出： [[12,21,16],[27,45,33],[24,39,28]]

示例 2：

输入： mat = [[1,2,3],[4,5,6],[7,8,9]], K = 2
输出： [[45,45,45],[45,45,45],[45,45,45]]

提示:

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n, K <= 100`
- `1 <= mat[i][j] <= 100`

```
int compute(int **matrixSums, int x1, int y1, int x2, int y2)
{
    if (x1 == 0 && y1 == 0) return matrixSums[x2][y2];
    if (x1 == 0) {
        return matrixSums[x2][y2] - matrixSums[x2][y1-1];
    }
    if (y1 == 0) {
        return matrixSums[x2][y2] - matrixSums[x1-1][y2];
    }

    int ans = matrixSums[x2][y2] - matrixSums[x2][y1-1] - matrixSums[x1-1][y2] + matrixSums[x1-1][y1-1];
    return ans;
}

/**
 * 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** matrixBlockSum(int** mat, int matSize, int* matColSize, int K, int* returnSize, int** returnColumnSizes)
{
    int **ans;
    int ansEnd = 0;
    int *colsSize;
    int colsSizeEnd = 0;

    if (mat == NULL || matSize == 0 || matColSize == NULL) {
        *returnColumnSizes = &colsSize;
        *returnSize = 0;
        return NULL;
    }

    if (matSize == 1 || matColSize[0] == 0) {
        *returnColumnSizes = &colsSize;
        *returnSize = 0;
        return NULL;
    }

    if (K == 0) {
        *returnColumnSizes = matColSize;
        *returnSize = matSize;
        return mat;
    }

    // 计算矩阵和存储下来
    int **sums = (int**)malloc(sizeof(int*) * matSize);
    for (int i = 0; i < matSize; i++) {
        sums[i] = (int*)malloc(sizeof(int) * matColSize[i]);
    }
    sums[0][0] = mat[0][0];
    for (int i = 1; i < matSize; i++) {
        sums[i][0] = sums[i-1][0] + mat[i][0];
    }
    for (int j = 1; j < matColSize[0]; j++) {
        sums[0][j] = sums[0][j-1] + mat[0][j];
    }
    for (int i = 1; i < matSize; i++) {
        for (int j = 1; j < matColSize[i]; j++) {
            sums[i][j] = sums[i-1][j] + sums[i][j-1] - sums[i-1][j-1] + mat[i][j];
        }
    }

    colsSize = (int*)malloc(sizeof(int) * matSize);
    for (int i = 0; i < matSize; i++) {
        colsSize[i] = matColSize[i];
    }
    ans = (int**)malloc(sizeof(int*) * matSize);
    for (int i = 0; i < matSize; i++) {
        ans[i] = (int*)malloc(sizeof(int) * matColSize[i]);
    }

    // 计算区域和
    int x1, y1, x2, y2;
    for (int i = 0; i < matSize; i++) {
```

```
        for (int j = 0; j < matColSize[i]; j++) {
            x1 = i - K < 0 ? 0 : i - K;
            y1 = j - K < 0 ? 0 : j - K;
            x2 = i + K >= matSize ? matSize - 1 : i + K;
            y2 = j + K >= matColSize[matSize-1] ? matColSize[matSize-1] - 1 : j + K;
            // (x1,y1) -- (x2, y2) 矩阵的和
            ans[i][j] = compute(sums, x1, y1, x2, y2);
        }
    }

    // 释放不使用的内存
    for (int i = 0; i < matSize; i++) {
        free(sums[i]);
    }
    free(sums);

    *returnSize = matSize;
    *returnColumnSizes = colsSize;
    return ans;
}
```

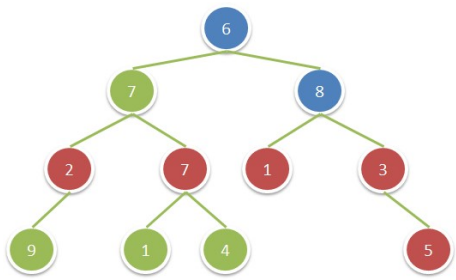
1315. 祖父节点值为偶数的节点和

给你一棵二叉树，请你返回满足以下条件的所有节点的值之和：

- 该节点的祖父节点的值为偶数。（一个节点的祖父节点是指该节点的父节点的父节点。）

如果不存在祖父节点值为偶数的节点，那么返回 `0` 。

示例：



输入： root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]
输出： 18
解释： 图中红色节点的祖父节点的值为偶数，蓝色节点为这些红色节点的祖父节点。

提示：

- 树中节点的数目在 `1` 到 `10^4` 之间。
- 每个节点的值在 `1` 到 `100` 之间。

```
int computeSum(struct TreeNode* root)
{
    if (root == NULL) return 0;

    int ans = 0;
    if (root->left) {
        struct TreeNode* left = root->left;
        if (left->left) ans += left->left->val;
        if (left->right) ans += left->right->val;
    }
    if (root->right) {
        struct TreeNode* right = root->right;
        if (right->left) ans += right->left->val;
        if (right->right) ans += right->right->val;
    }
    return ans;
}

void dfs(struct TreeNode* root, int *ans)
{
    if (root->val % 2 == 0) {
        *ans += computeSum(root);
    }
    if (root->left) dfs(root->left, ans);
    if (root->right) dfs(root->right, ans);
    return;
}

int sumEvenGrandparent(struct TreeNode* root){
    if (root == NULL) return 0;
    if (root->left == NULL && root->right == NULL) return 0;

    int ans = 0;
    dfs(root, &ans);
    return ans;
}
```

1360. 日期之间隔几天

请你编写一个程序来计算两个日期之间隔了多少天。

日期以字符串形式给出，格式为 `YYYY-MM-DD`，如示例所示。

示例 1：

```
输入：date1 = "2019-06-29", date2 = "2019-06-30"
输出：1
```

示例 2：

```
输入：date1 = "2020-01-15", date2 = "2019-12-31"
输出：15
```

提示：

- 给定的日期是 `1971` 年到 `2100` 年之间的有效日期。

```
void parse(char *string, int *year, int *month, int *day)
{
    int yyyy = (string[0] - '0') * 1000 + (string[1] - '0') * 100 + (string[2] - '0') * 10 + (string[3] - '0');
    int mm = (string[5] - '0') * 10 + (string[6] - '0');
    int dd = (string[8] - '0') * 10 + (string[9] - '0');

    *year = yyyy, *month = mm, *day = dd;

    return;
}
int getMonthDay(int year, int m)
{
    int ans = 0;
    switch(m){
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            ans = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            ans = 30;
            break;
        case 2:
            if (year % 400 == 0 || (year % 100 != 0 && year % 4 == 0)) ans = 29;
            else ans = 28;
            break;
    }
    return ans;
}
int days(int year, int month, int day)
{
    int ans = day;
    for (int i = 1; i < month; i++) {
        ans += getMonthDay(year, i);
    }
    printf("%d-%d-%d : %d\n", year, month, day, ans);
    return ans;
}
int daysBetweenDates(char * date1, char * date2){
    int y1, m1, d1; int y2, m2, d2;
    int ans;

    parse(date1, &y1, &m1, &d1); parse(date2, &y2,&m2,&d2);

    if (y1 >= y2) {
        int dd1 = 0;
        dd1 = days(y1, m1, d1);
        for (int yy = y1 - 1; yy >= y2; yy--) {
            dd1 += days(yy,12,31);
        }

        int dd2 = days(y2, m2, d2);
        if (dd1 > dd2) ans = dd1 - dd2;
        else ans = dd2 - dd1;
        return ans;
    }
    return daysBetweenDates(date2, date1);
}
```

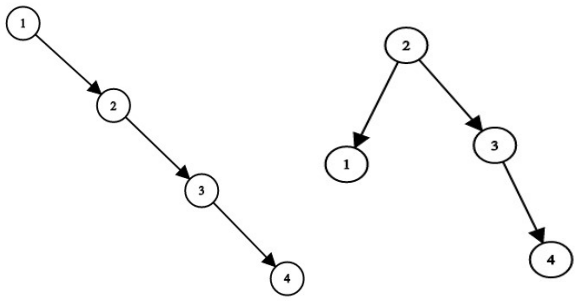
1382. 将二叉搜索树变平衡

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。

```
#define N 10002

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
void middle(struct TreeNode* root, int *arr, int *arrEnd)
{
    if (root == NULL) return;
    if (root->left) {
        middle(root->left, arr, arrEnd);
    }
    arr[(*arrEnd)++] = root->val;
    if (root->right) {
        middle(root->right, arr, arrEnd);
    }
    return;
}

struct TreeNode* balanceDFS(int *arr, int start, int end)
{
    if (start > end) return NULL;

    if (start == end) {
        struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
        newNode->val = arr[start];
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
    }

    int mid = (end + start) / 2;
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->val = arr[mid];
    newNode->left = balanceDFS(arr, start, mid - 1);
    newNode->right = balanceDFS(arr, mid + 1, end);
    return newNode;
}

struct TreeNode* balanceBST(struct TreeNode* root){
    if (root == NULL) return root;

    int *array = (int*)malloc(sizeof(int) * N);
    int arrayEnd = 0;

    middle(root, array, &arrayEnd); // 中序遍历得到排序好的数组
    for (int i = 0; i < arrayEnd; i++) {
        printf("%d ", array[i]);
    }

    int mid = (arrayEnd - 1) / 2;
    root = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    root->val = array[mid];
    root->left = balanceDFS(array, 0, mid - 1);
    root->right = balanceDFS(array, mid + 1, arrayEnd - 1);

    free(array);

    return root;
}
```

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
```

1311. 获取你好友已观看的视频

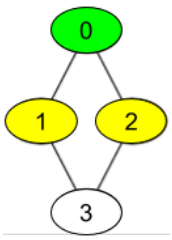
有 n 个人，每个人都有一个 0 到 n-1 的唯一 *id* 。

给你数组 watchedVideos 和 friends ，其中 watchedVideos[i] 和 friends[i] 分别表示 id = i 的人观看过的视频列表和他的好友列表。

Level **1** 的视频包含所有你好友观看过的视频，level **2** 的视频包含所有你好友的好友观看过的视频，以此类推。一般的，Level 为 **k** 的视频包含所有从你出发，最短距离为 **k** 的好友观看过的视频。

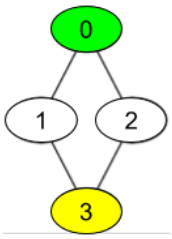
给定你的 id 和一个 level 值，请你找出所有指定 level 的视频，并将它们按观看频率升序返回。如果有频率相同的视频，请将它们按字母顺序从小到大排列。

示例 1：



输入： watchedVideos = [["A","B"],["C"],["B","C"],["D"]], friends = [[1,2],[0,3],[0,3],[1,2]], id = 0, level = 1
输出： ["B","C"]
解释：
你的 id 为 0（绿色），你的朋友包括（黄色）：
id 为 1 -> watchedVideos = ["C"]
id 为 2 -> watchedVideos = ["B","C"]
你朋友观看过视频的频率为：
B -> 1
C -> 2

示例 2：



输入： watchedVideos = [["A","B"],["C"],["B","C"],["D"]], friends = [[1,2],[0,3],[0,3],[1,2]], id = 0, level = 2
输出： ["D"]
解释：
你的 id 为 0（绿色），你朋友的朋友只有一个人，他的 id 为 3（黄色）。

提示：

- n == watchedVideos.length == friends.length
- 2 <= n <= 100
- 1 <= watchedVideos[i].length <= 100
- 1 <= watchedVideos[i][j].length <= 8
- 0 <= friends[i].length < n
- 0 <= friends[i][j] < n
- 0 <= id < n
- 1 <= level < n
- 如果 friends[i] 包含 j ，那么 friends[j] 包含 i

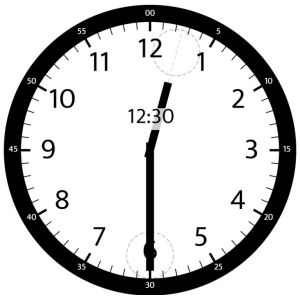
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "uthash.h"

#define MAXN 1000
#define N 100

struct HashTable {
    char key[N + 1]; // N个字母
    int count; // 出现的次数
    UT_hash_handle hh;
};
typedef struct HashTable Map;

/* 拷贝一个字符串 */
char *deepCopy(char *string)
{
    if (string == NULL) return NULL;

    const int n = strlen(string);
    char *newString = (char*)malloc(sizeof(char) * (n + 1));

    for (int i = 0; i < n; i++) {
        newString[i] = string[i];
    }
    newString[n] = '\0';
    return newString;
}

/*
 * 频率不同按照频率从小到大排列
 * 频率相同按照字典序排列
 */
int compare_function(Map *a, Map *b)
{
    if (a->count != b->count) {
        return a->count - b->count;
    }

    return strcmp(a->key, b->key);
}

/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
char ** watchedVideosByFriends(char *** watchedVideos, int watchedVideosSize, int* watchedVideosColSize,
    int** friends, int friendsSize, int* friendsColSize, int id, int level, int* returnSize)
{
    if (watchedVideos == NULL || watchedVideosColSize == NULL || watchedVideosSize <= 0) {
        *returnSize = 0;
        return NULL;
    }

    if (friends == NULL || friendsColSize == NULL || friendsSize <= 0) {
        *returnSize = 0;
        return NULL;
    }

    int *curr = (int*)malloc(sizeof(int*) * MAXN); // 当前的人
    int currEnd = 0;
    int *next = (int*)malloc(sizeof(int) * MAXN); // 当前人的所有朋友
    int nextEnd = 0;
    int *unique = (int*)malloc(sizeof(int) * MAXN);


    curr[0] = id;
    currEnd = 1;
    while (level) {
        // 把 curr 中人的朋友放入 next 中
        for (int i = 0; i < MAXN; i++) unique[i] = 0;
        for (int i = 0; i < currEnd; i++) {
            for (int j = 0; j < friendsColSize[i]; j++) {
                if (unique[ friends[i][j] ] == 0) {
```

```
                    next[nextEnd++] = friends[i][j];
                    unique[ friends[i][j] ] = 1;
                }
            }
        }
        // 把 next 放入 curr 中， 然后把 next 清空，继续下一次循环
        currEnd = 0;
        for (int i = 0; i < nextEnd; i++) {
            curr[currEnd++] = next[i];
        }
        nextEnd = 0;
        level--;
    }

    // curr 中保存的是id, level层级的所有朋友
    Map *map = NULL; // 保存每个视频出现的次数
    Map *current = NULL;
    for (int i = 0; i < currEnd; i++) {
        for (int j = 0; j < watchedVideosColSize[i]; j++) {
            char *key = watchedVideos[i][j];
            HASH_FIND_STR(map, key, current);
            if (current == NULL) {
                current = (Map*)malloc(sizeof(Map));
                int len = strlen(watchedVideos[i][j]);
                strncpy(current->key, string, len); current->key[len+1] = '\0';
                current->count = 1;
                HASH_ADD_STR(map, key, current);
            } else {
                current->count++;
            }
        }
    }

    // 遍历map, 按照出现频率从低到高排列
    HASH_SORT(map, compare_function);

    int ansCount = HASH_COUNT(map);
    char **ans = (char**)malloc(sizeof(char*) * ansCount);
    int ansEnd = 0;

    Map *tmp = NULL;
    for(tmp = map; tmp != NULL; tmp = tmp->hh.next) {
        ans[ansEnd++] = deepCopy(tmp->key);
    }

    // 释放用不到的空间
    Map *current = NULL;
    Map *tmp = NULL;
    HASH_ITER(hh, map, current, tmp) {
        HASH_DEL(map, current);
        free(current);
    }
    free(curr);
    free(next);

    return ans;
}
```

1344. 时钟指针的夹角

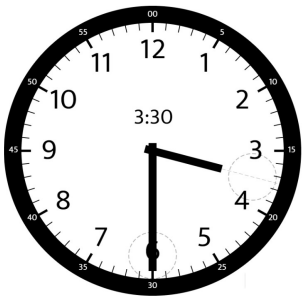
给你两个数 hour 和 minutes 。请你返回在时钟上，由给定时间的时针和分针组成的较小角的角度（60 单位制）。

示例 1：



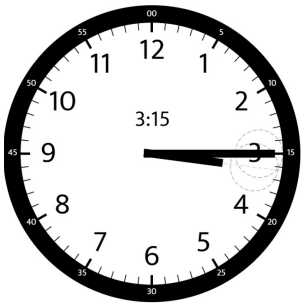
输入：hour = 12, minutes = 30
输出：165

示例 2：



输入：hour = 3, minutes = 30
输出：75

示例 3：



输入：hour = 3, minutes = 15
输出：7.5

示例 4：

输入：hour = 4, minutes = 50
输出：155

示例 5：

输入：hour = 12, minutes = 0
输出：0

提示：

- `1 <= hour <= 12`
- `0 <= minutes <= 59`
- 与标准答案误差在 `10^-5` 以内的结果都被视为正确结果。

```
double angleClock(int hour, int minutes){
    hour = (hour == 12) ? 0 : hour;
    double hrand = hour * (360.0 / 12);
    double mrand = minutes * (360.0 / 60);
    hrand += minutes * (360.0 / (12 * 60));
    printf("%f, %f\n", hrand, mrand);
    double ans = hrand > mrand ? (hrand - mrand) : (mrand - hrand);
    if (ans > 180) return 360.0 - ans;
    return ans;
}
```

1356. 根据数字二进制下 1 的数目排序

给你一个整数数组 `arr` 。请你将数组中的元素按照其二进制表示中数字 **1** 的数目升序排序。

如果存在多个数字二进制中 **1** 的数目相同，则必须将它们按照数值大小升序排列。

请你返回排序后的数组。

示例 1：

```
输入：arr = [0,1,2,3,4,5,6,7,8]
输出：[0,1,2,4,8,3,5,6,7]
解释：[0] 是唯一一个有 0 个 1 的数。
[1,2,4,8] 都有 1 个 1 。
[3,5,6] 有 2 个 1 。
[7] 有 3 个 1 。
按照 1 的个数排序得到的结果数组为 [0,1,2,4,8,3,5,6,7]
```

示例 2：

```
输入：arr = [1024,512,256,128,64,32,16,8,4,2,1]
输出：[1,2,4,8,16,32,64,128,256,512,1024]
解释：数组中所有整数二进制下都只有 1 个 1 ，所以你需要按照数值大小将它们排序。
```

示例 3：

```
输入：arr = [10000,10000]
输出：[10000,10000]
```

示例 4：

```
输入：arr = [2,3,5,7,11,13,17,19]
输出：[2,3,5,17,7,11,13,19]
```

示例 5：

```
输入：arr = [10,100,1000,10000]
输出：[10,100,10000,1000]
```

提示：

- `1 <= arr.length <= 500`
- `0 <= arr[i] <= 10^4`

```
int Bits(int n)
{
    int count = 0;
    while (n > 0) {
        count++;
        n = n & (n - 1);
    }
    //printf("%d : %d\n", n, count);
    return count;
}
int compare(const void *p, const void *q)
{
    int a = *(int*)p;
    int b = *(int*)q;
    int aBits = Bits(a);
    int bBits = Bits(b);
    if (aBits == bBits) return a - b;
    return aBits - bBits;
}
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* sortByBits(int* arr, int arrSize, int* returnSize){
    qsort(arr, arrSize, sizeof(int), compare);
    *returnSize = arrSize;
    return arr;
}
```

1370. 上升下降字符串

给你一个字符串 `s` ，请你根据下面的算法重新构造字符串：

1. 从 `s` 中选出 **最小** 的字符，将它 **接在** 结果字符串的后面。
2. 从 `s` 剩余字符中选出 **最小** 的字符，且该字符比上一个添加的字符大，将它 **接在** 结果字符串后面。
3. 重复步骤 2 ，直到你没法从 `s` 中选择字符。
4. 从 `s` 中选出 **最大** 的字符，将它 **接在** 结果字符串的后面。
5. 从 `s` 剩余字符中选出 **最大** 的字符，且该字符比上一个添加的字符小，将它 **接在** 结果字符串后面。
6. 重复步骤 5 ，直到你没法从 `s` 中选择字符。
7. 重复步骤 1 到 6 ，直到 `s` 中所有字符都已经被选过。

在任何一步中，如果最小或者最大字符不止一个 ，你可以选择其中任意一个，并将其添加到结果字符串。

请你返回将 `s` 中字符重新排序后的 **结果字符串** 。

示例 1：

```
输入：s = "aaaabbbbcccc"
输出："abccbaabccba"
解释：第一轮的步骤 1，2，3 后，结果字符串为 result = "abc"
第一轮的步骤 4，5，6 后，结果字符串为 result = "abccba"
第一轮结束，现在 s = "aabbcc" ，我们再次回到步骤 1
第二轮的步骤 1，2，3 后，结果字符串为 result = "abccbaabc"
第二轮的步骤 4，5，6 后，结果字符串为 result = "abccbaabccba"
```

示例 2：

```
输入：s = "rat"
输出："art"
解释：单词 "rat" 在上述算法重排序以后变成 "art"
```

示例 3：

```
输入：s = "leetcode"
输出："cdelotee"
```

示例 4：

输入： s = "ggggggg"
输出： "ggggggg"

示例 5:

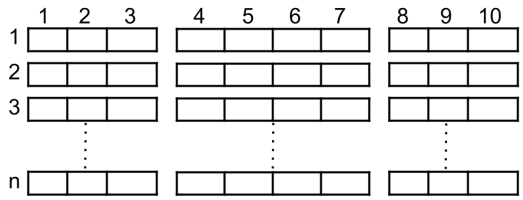
输入： s = "spo"
输出： "ops"

提示:

- 1 <= s.length <= 500
- s 只包含小写英文字母。

```
#define MAX_CHAR 128
#define MAX_ANS 501

int map[MAX_CHAR];
void init()
{
    for (int i = 0; i < MAX_CHAR; i++) {
        map[i] = 0;
    }
}

char * sortString(char * s){
    if (s == NULL) return s;

    int n = strlen(s);
    int i;

    for (i = 0; i < n; i++) {
        map[ s[i] ] ++;
    }

    char *ans = (char*)malloc(sizeof(char) * MAX_ANS);
    int ansEnd = 0;
    while (n > 0) {
        for (i = 0; i < MAX_CHAR; i++) {
            if (map[i] > 0) {
                ans[ansEnd++] = (char)i;
                n--;
                map[i]--;
            }
        }
        for (int i = MAX_CHAR - 1; i >= 0; i--) {
            if (map[i] > 0) {
                ans[ansEnd++] = (char)i;
                n--;
                map[i]--;
            }
        }
    }
    ans[ansEnd++] = '\0';
    return ans;
}
```

1386. 安排电影院座位

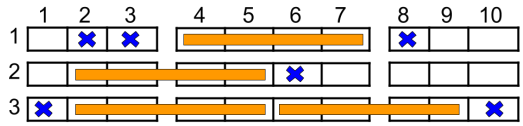


如上图所示，电影院的观影厅中有 n 行座位，行编号从 1 到 n ，且每一行内总共有 10 个座位，列编号从 1 到 10 。

给你数组 `reservedSeats` ，包含所有已经被预约了的座位。比如说， `researvedSeats[i]=[3,8]` ，它表示第 **3** 行第 **8** 个座位被预约了。

请你返回 **最多能安排多少个 4 人家庭** 。4 人家庭要占据 **同一行内连续** 的 4 个座位。隔着过道的座位（比方说 [3,3] 和 [3,4]）不是连续的座位，但是如果你可以将 4 人家庭拆成过道两边各坐 2 人，这样子是允许的。

示例 1：




```
输入：n = 3, reservedSeats = [[1,2],[1,3],[1,8],[2,6],[3,1],[3,10]]
输出：4
解释：上图所示是最优的安排方案，总共可以安排 4 个家庭。蓝色的叉表示被预约的座位，橙色的连续座位表示一个 4 人家庭。
```

示例 2：

```
输入：n = 2, reservedSeats = [[2,1],[1,8],[2,6]]
输出：2
```

示例 3：

```
输入：n = 4, reservedSeats = [[4,3],[1,4],[4,6],[1,7]]
输出：4
```

提示：

- `1 <= n <= 10^9`
- `1 <= reservedSeats.length <= min(10*n, 10^4)`
- `reservedSeats[i].length == 2`
- `1 <= reservedSeats[i][0] <= n`
- `1 <= reservedSeats[i][1] <= 10`
- 所有 `reservedSeats[i]` 都是互不相同的。

```
int familysInRow(int *seated)
{
    if (seated[1] == 0 && seated[2] == 0 && seated[3] == 0 && seated[4] == 0 &&
        seated[5] == 0 && seated[6] == 0 && seated[7] == 0 && seated[8] == 0) {
        return 2;
    }
    if (seated[1] == 0 && seated[2] == 0 && seated[3] == 0 && seated[4] == 0) {
        return 1;
    }
    if (seated[3] == 0 && seated[4] == 0 && seated[5] == 0 && seated[6] == 0) {
        return 1;
    }
    if (seated[5] == 0 && seated[6] == 0 && seated[7] == 0 && seated[8] == 0) {
        return 1;
    }
    return 0;
}

int compare(const void *p, const void *q)
{
    int *a = *(int**)p;
    int *b = *(int**)q;
    printf("pp:%d, %d\n", a[0],b[0]);
    return a[0] - b[0];
}
int maxNumberOfFamilies(int n, int** reservedSeats, int reservedSeatsSize, int* reservedSeatsColSize){
    int *row = (int*)malloc(sizeof(int) * 10);
    int rowEnd = 0;
    int ans = 0;

    qsort(reservedSeats, reservedSeatsSize, sizeof(int) * 2, compare);

    int line = 1;
    int iter = 0;
    int count = 0;
    while (iter < reservedSeatsSize) {
        for (int i = 0; i < 10; i++) row[i] = 0;
        if (count > 1) {
            ans += 2 * (count - 1);
        }
        count = 0;
        while (iter < reservedSeatsSize && reservedSeats[iter][0] == line) {
            printf("\n%d,%d\n", reservedSeats[iter][0], iter);
            row[ reservedSeats[iter][1]-1 ] = 1;
            iter++;
        }
        ans += familysInRow(row);
        while (iter < reservedSeatsSize && reservedSeats[iter][0] > line) {
            line++;
            count++;
        }
    }
    if (n - line > 0) {
        ans += 2 * (n - line);
    }
    free(row);

    return ans;
}
```

1376. 通知所有员工所需的时间

公司里有 `n` 名员工，每个员工的 ID 都是独一无二的，编号从 `0` 到 `n - 1`。公司的总负责人通过 `headID` 进行标识。

在 `manager` 数组中，每个员工都有一个直属负责人，其中 `manager[i]` 是第 `i` 名员工的直属负责人。对于总负责人，`manager[headID] = -1`。题目保证从属关系可以用树结构显示。

公司总负责人想要向公司所有员工通告一条紧急消息。他将会首先通知他的直属下属们，然后由这些下属通知他们的下属，直到所有的员工都得知这条紧急消息。

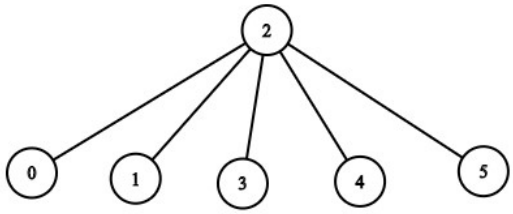
第 i 名员工需要 informTime[i] 分钟来通知它的所有直属下属（也就是说在 informTime[i] 分钟后，他的所有直属下属都可以开始传播这一消息）。

返回通知所有员工这一紧急消息所需要的 **分钟数** 。

示例 1：

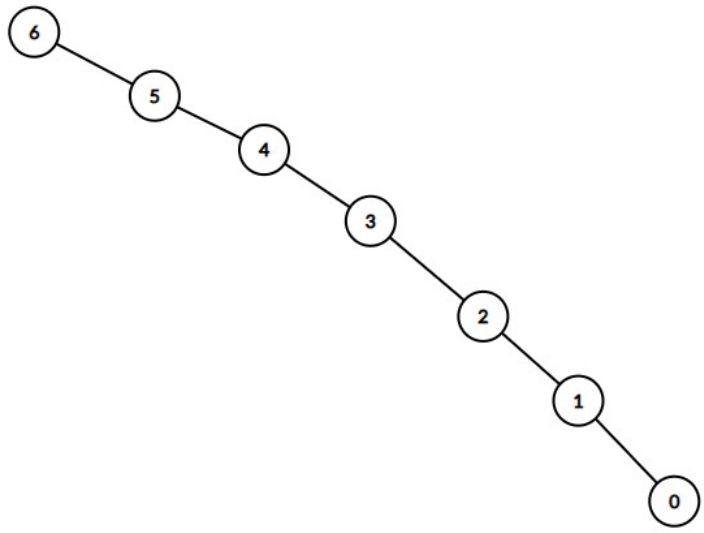
```
输入：n = 1, headID = 0, manager = [-1], informTime = [0]
输出：0
解释：公司总负责人是该公司的唯一一名员工。
```

示例 2：




```
输入：n = 6, headID = 2, manager = [2,2,-1,2,2,2], informTime = [0,0,1,0,0,0]
输出：1
解释：id = 2 的员工是公司的总负责人，也是其他所有员工的直属负责人，他需要 1 分钟来通知所有员工。
上图显示了公司员工的树结构。
```

示例 3：




```
输入：n = 7, headID = 6, manager = [1,2,3,4,5,6,-1], informTime = [0,6,5,4,3,2,1]
输出：21
解释：总负责人 id = 6。他将在 1 分钟内通知 id = 5 的员工。
id = 5 的员工将在 2 分钟内通知 id = 4 的员工。
id = 4 的员工将在 3 分钟内通知 id = 3 的员工。
id = 3 的员工将在 4 分钟内通知 id = 2 的员工。
id = 2 的员工将在 5 分钟内通知 id = 1 的员工。
id = 1 的员工将在 6 分钟内通知 id = 0 的员工。
所需时间 = 1 + 2 + 3 + 4 + 5 + 6 = 21 。
```

示例 4：

```
输入：n = 15, headID = 0, manager = [-1,0,0,1,1,2,2,3,3,4,4,5,5,6,6], informTime = [1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]
输出：3
解释：第一分钟总负责人通知员工 1 和 2 。
第二分钟他们将会通知员工 3, 4, 5 和 6 。
第三分钟他们将会通知剩下的员工。
```

示例 5：

输入： n = 4, headID = 2, manager = [3,3,-1,2], informTime = [0,0,162,914]
输出： 1076

提示：

- 1 <= n <= 10^5
- 0 <= headID < n
- manager.length == n
- 0 <= manager[i] < n
- manager[headID] == -1
- informTime.length == n
- 0 <= informTime[i] <= 1000
- 如果员工 i 没有下属，informTime[i] == 0 。
- 题目 **保证** 所有员工都可以收到通知。

```
#define MAXN 100000


int *managerG;
int *informTimeG;
int managerSizeG;
int informTimeSizeG;
int max(int a, int b)
{
    if (a > b) return a;
    return b;
}

int dfs(int i, int cost)
{
    if (informTimeG[i] == 0) {
        return cost;
    }
    int ans = 0;
    for (int j = 0; j < managerSizeG; j++) {
        if (managerG[j] == i) {
            int tmp = dfs(j, cost + informTimeG[i]);
            ans = max(ans, tmp);
        }
    }
    return ans;
}
int numOfMinutes(int n, int headID, int* manager, int managerSize, int* informTime, int informTimeSize){
    if (n <= 0 || manager == NULL || managerSize <= 0 || informTime == NULL || informTimeSize <= 0) return 0;

    int ans = 0;
    managerG = manager;
    managerSizeG = managerSize;
    informTimeG = informTime;
    informTimeSizeG = informTimeSize;

    ans = dfs(headID, 0);

    return ans;
}
```

1380. 矩阵中的幸运数

给你一个 m * n 的矩阵，矩阵中的数字 **各不相同** 。请你按 **任意** 顺序返回矩阵中的所有幸运数。

幸运数是指矩阵中满足同时下列两个条件的元素：

- 在同一行的所有元素中最小
- 在同一列的所有元素中最大

示例 1：

```
输入：matrix = [[3,7,8],[9,11,13],[15,16,17]]
输出：[15]
解释：15 是唯一的幸运数，因为它是其所在行中的最小值，也是所在列中的最大值。
```

示例 2：

```
输入：matrix = [[1,10,4,2],[9,3,8,7],[15,16,17,12]]
输出：[12]
解释：12 是唯一的幸运数，因为它是其所在行中的最小值，也是所在列中的最大值。
```

示例 3：

```
输入：matrix = [[7,8],[1,2]]
输出：[7]
```

提示：

- `m == mat.length`
- `n == mat[i].length`
- `1 <= n, m <= 50`
- `1 <= matrix[i][j] <= 10^5`
- 矩阵中的所有元素都是不同的

```c
#define MAX_ANS 1000

int minRows(int **matrix, int row, int rowSize)
{
    int minValue = INT_MAX;
    for (int i = 0; i < rowSize; i++) {
        if (matrix[row][i] < minValue) {
            minValue = matrix[row][i];
        }
    }
    return minValue;
}

int maxCols(int **matrix, int col, int colSize)
{
    int maxValue = INT_MIN;
    for (int i = 0; i < colSize; i++) {
        if (matrix[i][col] > maxValue) {
            maxValue = matrix[i][col];
        }
    }
    return maxValue;
}

int* luckyNumbers (int** matrix, int matrixSize, int* matrixColSize, int* returnSize)
{
    int *ans = NULL;
    int ansEnd = 0;

    if (matrix == NULL || matrixSize == 0) {
        *returnSize = 0;
        return ans;
    }

    if (matrixSize == 1 && matrixColSize[0] == 0) {
        *returnSize = 0;
        return ans;
    }

    ans = (int*)malloc(sizeof(int) * MAX_ANS);
    for (int i = 0; i < matrixSize; i++) {
        int minValue = minRows(matrix, i, matrixColSize[i]);
        for (int j = 0; j < matrixColSize[i]; j++) {
            if (matrix[i][j] == minValue) {
                int maxValue = maxCols(matrix, j, matrixSize);
                if (maxValue == minValue) {
                    ans[ansEnd++] = minValue;
                }
            }
        }
    }

    return ans;
}
```

1381. 设计一个支持增量操作的栈

请你设计一个支持下述操作的栈。

实现自定义栈类 `CustomStack` ：

- `CustomStack(int maxSize)`：用 `maxSize` 初始化对象， `maxSize` 是栈中最多能容纳的元素数量，栈在增长到 `maxSize` 之后则不支持 `push` 操作。
- `void push(int x)`：如果栈还未增长到 `maxSize` ，就将 `x` 添加到栈顶。
- `int pop()`：弹出栈顶元素，并返回栈顶的值，或栈为空时返回 **-1** 。
- `void inc(int k, int val)`：栈底的 `k` 个元素的值都增加 `val` 。如果栈中元素总数小于 `k` ，则栈中的所有元素都增加 `val` 。

示例：

输入：
```
["CustomStack","push","push","pop","push","push","push","increment","increment","pop","pop","pop","pop"]
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]
```
输出：
```
[null,null,null,2,null,null,null,null,null,103,202,201,-1]
```
解释：
```
CustomStack customStack = new CustomStack(3); // 栈是空的 []
customStack.push(1);                          // 栈变为 [1]
customStack.push(2);                          // 栈变为 [1, 2]
customStack.pop();                            // 返回 2 --> 返回栈顶值 2，栈变为 [1]
customStack.push(2);                          // 栈变为 [1, 2]
customStack.push(3);                          // 栈变为 [1, 2, 3]
customStack.push(4);                          // 栈仍然是 [1, 2, 3]，不能添加其他元素使栈大小变为 4
customStack.increment(5, 100);                // 栈变为 [101, 102, 103]
customStack.increment(2, 100);                // 栈变为 [201, 202, 103]
customStack.pop();                            // 返回 103 --> 返回栈顶值 103，栈变为 [201, 202]
customStack.pop();                            // 返回 202 --> 返回栈顶值 202，栈变为 [201]
customStack.pop();                            // 返回 201 --> 返回栈顶值 201，栈变为 []
customStack.pop();                            // 返回 -1 --> 栈为空，返回 -1
```

提示：

- `1 <= maxSize <= 1000`
- `1 <= x <= 1000`
- `1 <= k <= 1000`
- `0 <= val <= 100`
- 每种方法 `increment`， `push` 以及 `pop` 分别最多调用 `1000` 次

```
#define N 1001
#define min(a,b) ((a) > (b) ? (b) : (a))

typedef struct {
    int array[N];
    int end;
    int maxSize;
} CustomStack;


CustomStack* customStackCreate(int maxSize) {
    CustomStack *new = (CustomStack*)malloc(sizeof(CustomStack));
    new->end = 0;
    new->maxSize = maxSize;
    return new;
}

void customStackPush(CustomStack* obj, int x) {
    if (obj->end < obj->maxSize) {
        obj->array[obj->end++] = x;
    }
}

int customStackPop(CustomStack* obj) {
    int ans = -1;
    if (obj->end > 0) {
        ans = obj->array[(obj->end)--];
    }
    return ans;
}

void customStackIncrement(CustomStack* obj, int k, int val) {
    for (int i = 0; i < min(k, obj->end); i++) {
        obj->array[i] += val;
    }
}

void customStackFree(CustomStack* obj) {
    free(obj);
}

/**
 * Your CustomStack struct will be instantiated and called as such:
 * CustomStack* obj = customStackCreate(maxSize);
 * customStackPush(obj, x);

 * int param_2 = customStackPop(obj);

 * customStackIncrement(obj, k, val);

 * customStackFree(obj);
*/
```

LCP 3. 机器人大冒险

力扣团队买了一个可编程机器人，机器人初始位置在原点 (0, 0) 。小伙伴事先给机器人输入一串指令 command ，机器人就会**无限循环**这条指令的步骤进行移动。指令有两种：

1. U：向 y 轴正方向移动一格
2. R：向 x 轴正方向移动一格。

不幸的是，在 xy 平面上还有一些障碍物，他们的坐标用 obstacles 表示。机器人一旦碰到障碍物就会被**损毁**。

给定终点坐标 (x, y)，返回机器人能否**完好**地到达终点。如果能，返回 true； 否则返回 false 。

示例 1：

输入： command = "URR", obstacles = [], x = 3, y = 2
输出： true
解释： U(0, 1) -> R(1, 1) -> R(2, 1) -> U(2, 2) -> R(3, 2)。

示例 2：

输入： command = "URR", obstacles = [[2, 2]], x = 3, y = 2
输出： false
解释： 机器人在到达终点前会碰到(2, 2)的障碍物。

示例 3：

输入： command = "URR", obstacles = [[4, 2]], x = 3, y = 2
输出： true
解释： 到达终点后，再碰到障碍物也不影响返回结果。

限制：

1. 2 <= command的长度 <= 1000
2. command 由 U，R 构成，且至少有一个 U ，至少有一个 R
3. 0 <= x <= 1e9, 0 <= y <= 1e9
4. 0 <= obstacles的长度 <= 1000
5. obstacles[i] 不为原点或者终点

暴力方法，超出时间限制

48 / 53 个通过测试用例

```
class Solution(object):
    def robot(self, command, obstacles, x, y):
        """
        :type command: str
        :type obstacles: List[List[int]]
        :type x: int
        :type y: int
        :rtype: bool
        """
        n = len(command)

        i, j, index = 0, 0, 0
        while (i <= x and j <= y):
            if command[index] == 'U':
                if [i, j+1] in obstacles:
                    return False
                else:
                    j += 1
            if command[index] == 'R':
                if [i+1, j] in obstacles:
                    return False
                else:
                    i += 1
            if i == x and j == y:
                return True
            index = (index + 1) % n

        return False
```

只处理一个小矩阵的数据，其他的都可以复用，通过。

```
class Solution(object):
    def robot(self, command, obstacles, x, y):
        """
        :type command: str
        :type obstacles: List[List[int]]
        :type x: int
        :type y: int
        :rtype: bool
        """
        ucount = command.count("U")
        rcount = command.count("R")

        # 首先判断如果没有障碍物，能否到达终点
        minc = min(x // rcount, y // ucount)
        xleave = x - rcount * minc
        yleave = y - ucount * minc
        # 如果剩下的大于command的长度,说明x,y长度不匹配，直接返回失败
        if xleave + yleave > len(command):
            return False
        xyleave = command[:xleave + yleave]
        # 如果剩下的不能用command的开头部分元素走完，直接返回失败
        if xyleave.count("R") != xleave or xyleave.count("U") != yleave:
            return False

        # 将障碍物缩放到最小的方格中
        subobstacles = []
        for obstacle in obstacles:
            # 位于最小矩形之内的首先放入结果
            if obstacle[0] < rcount and obstacle[1] < ucount:
                subobstacles.append((obstacle[0], obstacle[1]))
            # 位于终点之外的不会影响最终结果，不处理
            elif obstacle[0] <= x and obstacle[1] <= y:
                minc = min(obstacle[0] // rcount, obstacle[1] // ucount)
                (tmpx, tmpy) = (obstacle[0] - rcount * minc, obstacle[1] - ucount * minc)
                if (tmpx, tmpy) not in subobstacles:
                    subobstacles.append((tmpx, tmpy))

        # 只判断最小的矩阵中是否阻挡了，其余的都是重复的
        i, j = 0, 0
        for com in command:
            # 当前位于(i,j) 如果当前是障碍，返回false
            if (i, j) in subobstacles:
                return False
            # 是否可以向上走
            if com == "U":
                if (i, j+1) in subobstacles:
                    return False
                else:
                    j += 1
            # 是否可以向右走
            if com == "R":
                if (i+1, j) in subobstacles:
                    return False
                else:
                    i += 1

        return True
```