152. Maximum Product Subarray

Given an integer array `nums`, find the contiguous subarray within an array (containing at least one number) which has the largest product.

Example 1:

```
Input: [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
```

Example 2:

```
Input: [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.
```

这个求最大子数组乘积问题是由最大子数组之和 Maximum Subarray 演变而来，但是却比求最大子数组之和要复杂，因为在求和的时候，遇到0，不会改变最大值，遇到负数，也只是会减小最大值而已。而在求最大子数组乘积的问题中，遇到0会使整个乘积为0，而遇到负数，则会使最大乘积变成最小乘积，正因为有负数和0的存在，使问题变得复杂了不少。比如，现在有一个数组 [2, 3, -2, 4]，可以很容易的找出所有的连续子数组，[2]，[3]，[-2]，[4]，[2, 3]，[3, -2]，[-2, 4]，[2, 3, -2]，[3, -2, 4]，[2, 3, -2, 4]，然后可以很轻松的算出最大的子数组乘积为6，来自子数组 [2, 3]。但如何写代码来实现自动找出最大子数组乘积呢，博主最先想到的方比较简单粗暴，就是找出所有的子数组，然后算出每一个子数组的乘积，然后比较找出最大的一个，需要两个 for 循环，第一个 for 遍历整个数组，第二个 for 遍历含有当前数字的子数组，就是按以下顺序找出子数组: [2]，[2, 3]，[2, 3, -2]，[2, 3, -2, 4]，[3]，[3, -2]，[3, -2, 4]，[-2]，[-2, 4]，[4]，在本地测试的一些数组全部通过，于是兴高采烈的拿到 OJ 上测试，结果丧心病狂的 OJ 用一个有 15000 个数字的数组来测试，然后说程序的运行时间超过了要求值，一看代码，果然如此，时间复杂度 O(n2), 得想办法只用一次循环搞定。想来想去想不出好方法，于是到网上搜各位大神的解决方法。其实这道题最直接的方法就是用 DP 来做，而且要用两个 dp 数组，其中 f[i] 表示子数组 [0, i] 范围内并且一定包含 nums[i] 数字的最大子数组乘积，g[i] 表示子数组 [0, i] 范围内并且一定包含 nums[i] 数字的最小子数组乘积，初始化时 f[0] 和 g[0] 都初始化为 nums[0]，其余都初始化为0。那么从数组的第二个数字开始遍历，那么此时的最大值和最小值只会在这三个数字之间产生，即 f[i-1]*nums[i]，g[i-1]*nums[i]，和 nums[i]。所以用三者中的最大值来更新 f[i]，用最小值来更新 g[i]，然后用 f[i] 来更新结果 res 即可，由于最终的结果不一定会包括 nums[n-1] 这个数字，所以 f[n-1] 不一定是最终解，不断更新的结果 res 才是，参见代码如下：

解法一：

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int res = nums[0], n = nums.size();
        vector<int> f(n, 0), g(n, 0);
        f[0] = nums[0];
        g[0] = nums[0];
        for (int i = 1; i < n; ++i) {
            f[i] = max(max(f[i - 1] * nums[i], g[i - 1] * nums[i]),
nums[i]);
            g[i] = min(min(f[i - 1] * nums[i], g[i - 1] * nums[i]),
nums[i]);
            res = max(res, f[i]);
        }
        return res;
    }
};
```

我们可以对上面的解法进行空间上的优化，以下摘自 OJ 官方解答，大体思路相同，写法更加简洁：

Besides keeping track of the largest product, we also need to keep track of the smallest product. Why? The smallest product, which is the largest in the negative sense could become the maximum when being multiplied by a negative number.

Let us denote that:

```
f(k) = Largest product subarray, from index 0 up to k.
```

Similarly,

```
g(k) = Smallest product subarray, from index 0 up to k.
```

Then,

```
f(k) = max( f(k-1) * A[k], A[k], g(k-1) * A[k] )
g(k) = min( g(k-1) * A[k], A[k], f(k-1) * A[k] )
```

There we have a dynamic programming formula. Using two arrays of size *n* , we could deduce the final answer as f(*n* -1). Since we only need to access its previous elements at each step, two variables are sufficient.

```
public int maxProduct(int[] A) {
   assert A.length > 0;
   int max = A[0], min = A[0], maxAns = A[0];
   for (int i = 1; i < A.length; i++) {
      int mx = max, mn = min;
      max = Math.max(Math.max(A[i], mx * A[i]), mn * A[i]);
      min = Math.min(Math.min(A[i], mx * A[i]), mn * A[i]);
      maxAns = Math.max(max, maxAns);
   }
   return maxAns;
}
```

根据上述描述可以写出代码如下：

解法二：

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        if (nums.empty()) return 0;
        int res = nums[0], mn = nums[0], mx = nums[0];
        for (int i = 1; i < nums.size(); ++i) {
            int tmax = mx, tmin = mn;
            mx = max(max(nums[i], tmax * nums[i]), tmin * nums[i]);
            mn = min(min(nums[i], tmax * nums[i]), tmin * nums[i]);
            res = max(res, mx);
        }
        return res;
    }
};
```

下面这种方法也是用两个变量来表示当前最大值和最小值的，但是没有无脑比较三个数，而是对于当前的 nums[i] 值进行了正负情况的讨论：

\1. 当遍历到一个正数时，此时的最大值等于之前的最大值乘以这个正数和当前正数中的较大值，此时的最小值等于之前的最小值乘以这个正数和当前正数中的较小值。

\2. 当遍历到一个负数时，先用一个变量t保存之前的最大值 mx，然后此时的最大值等于之前最小值乘以这个负数和当前负数中的较大值，此时的最小值等于之前保存的最大值t乘以这个负数和当前负数中的较小值。

\3. 在每遍历完一个数时，都要更新最终的最大值。

P.S. 如果这里改成求最小值的话，就是求最小子数组乘积，并且时间复杂度是醉人的 O(n)，是不是很强大呢，参见代码如下：

解法三：

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int res = nums[0], mx = res, mn = res;
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] > 0) {
                mx = max(mx * nums[i], nums[i]);
                mn = min(mn * nums[i], nums[i]);
            } else {
                int t = mx;
                mx = max(mn * nums[i], nums[i]);
                mn = min(t * nums[i], nums[i]);
            }
            res = max(res, mx);
        }
        return res;
    }
};
```

下面这道题使用了一个 trick 来将上面解法的分情况讨论合成了一种，在上面的解法中分析了当 nums[i] 为正数时，最大值和最小值的更新情况，为负数时，稍有不同的就是最小值更新时要用到之前的最大值，而不是更新后的最大值，所以才要用变量t来保存之前的结果。而下面这种方法的巧妙处在于先判断一个当前数字是否是负数，是的话就交换最大值和最小值。那么此时的 mx 就是之前的 mn，所以 mx 的更新还是跟上面的方法是统一的，而在在更新 mn 的时候，之前的 mx 已经保存到 mn 中了，而且并没有改变，所以可以直接拿来用，不得不说，确实叼啊，参见代码如下：

解法四：

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int res = nums[0], mx = res, mn = res;
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] < 0) swap(mx, mn);
            mx = max(nums[i], mx * nums[i]);
            mn = min(nums[i], mn * nums[i]);
            res = max(res, mx);
        }
        return res;
    }
};
```

再来看一种画风不太一样的解法，这种解法遍历了两次，一次是正向遍历，一次是反向遍历，相当于正向建立一个累加积数组，每次用出现的最大值更新结果 res，然后再反响建立一个累加积数组，再用出现的最大值更新结果 res，注意当遇到0的时候，prod 要重置为1。至于为啥正反两次遍历就可以得到正确的结果了呢？主要还是由于负数个数的关系，因为负数可能会把最大值和最小值翻转，那么当有奇数个负数时，如果只是正向遍历的话，可能会出错，比如 [-1, -2, -3]，累加积会得到 -1，2，-6，看起来最大值只能为2，其实不对，而如果我们再反向来一遍，累加积为 -3，6，-6，就可以得到6了。所以当负数个数为奇数时，首次出现和末尾出现的负数就很重要，有可能会是最大积的组成数字，所以遍历两次就不会漏掉组成最大值的机会，参见代码如下：

解法五：

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int res = nums[0], prod = 1, n = nums.size();
        for (int i = 0; i < n; ++i) {
            res = max(res, prod *= nums[i]);
            if (nums[i] == 0) prod = 1;
        }
        prod = 1;
        for (int i = n - 1; i >= 0; --i) {
            res = max(res, prod *= nums[i]);
            if (nums[i] == 0) prod = 1;
        }
        return res;
    }
};
```

153. Find Minimum in Rotated Sorted Array

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., `[0,1,2,4,5,6,7]` might become `[4,5,6,7,0,1,2]`).

Find the minimum element.

You may assume no duplicate exists in the array.

Example 1:

```
Input: [3,4,5,1,2]
Output: 1
```

Example 2:

```
Input: [4,5,6,7,0,1,2]
Output: 0
```

这道寻找旋转有序数组的最小值肯定不能通过直接遍历整个数组来寻找，这个方法过于简单粗暴，这样的话，旋不旋转就没有意义。应该考虑将时间复杂度从简单粗暴的 O(n) 缩小到 O(lgn)，这时候二分查找法就浮现在脑海。这里的二分法属于博主之前的总结帖 LeetCode Binary Search Summary 二分搜索法小结 中的第五类，也是比较难的那一类，没有固定的 target 值比较，而是要跟数组中某个特定位置上的数字比较，决定接下来去哪一边继续搜索。这里用中间的值 nums[mid] 和右边界值 nums[right] 进行比较，若数组没有旋转或者旋转点在左半段的时候，中间值是一定小于右边界值的，所以要去左半边继续搜索，反之则去右半段查找，最终返回 nums[right] 即可，参见代码如下：

解法一：

```cpp
class Solution {
public:
    int findMin(vector<int>& nums) {
        int left = 0, right = (int)nums.size() - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) left = mid + 1;
            else right = mid;
        }
        return nums[right];
    }
};
```

下面这种分治法 Divide and Conquer 的解法，由热心网友 howard144 提供，这里每次将区间 [start, end] 从中间 mid 位置分为两段，分别调用递归函数，并比较返回值，每次取返回值较小的那个即可，参见代码如下：

解法二：

```cpp
class Solution {
public:
    int findMin(vector<int>& nums) {
        return helper(nums, 0, (int)nums.size() - 1);
    }
    int helper(vector<int>& nums, int start, int end) {
        if (nums[start] <= nums[end]) return nums[start];
        int mid = (start + end) / 2;
        return min(helper(nums, start, mid), helper(nums, mid + 1, end));
    }
};
```

讨论：对于数组中有重复数字的情况，请参见博主的另一篇博文 Find Minimum in Rotated Sorted Array II。

154. Find Minimum in Rotated Sorted Array II

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., `[0,1,2,4,5,6,7]` might become `[4,5,6,7,0,1,2]`).

Find the minimum element.

The array may contain duplicates.

Example 1:

```
Input: [1,3,5]
Output: 1
```

Example 2:

```
Input: [2,2,2,0,1]
Output: 0
```

Note:

- This is a follow up problem to Find Minimum in Rotated Sorted Array.
- Would allow duplicates affect the run-time complexity? How and why?

这道寻找旋转有序重复数组的最小值是之前那道 Find Minimum in Rotated Sorted Array 的拓展，当数组中存在大量的重复数字时，就会破坏二分查找法的机制，将无法取得 O(lgn) 的时间复杂度，又将会回到简单粗暴的 O(n)，比如这两种情况：{2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} 和 {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}，可以发现，当第一个数字和最后一个数字，还有中间那个数字全部相等的时候，二分查找法就崩溃了，因为它无法判断到底该去左半边还是右半边。这种情况下，将右指针左移一位（或者将左指针右移一位），略过一个相同数字，这对结果不会产生影响，因为只是去掉了一个相同的，然后对剩余的部分继续用二分查找法，在最坏的情况下，比如数组所有元素都相同，时间复杂度会升到 O(n)，参见代码如下：

解法一：

```cpp
class Solution {
public:
    int findMin(vector<int>& nums) {
        int left = 0, right = (int)nums.size() - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) left = mid + 1;
            else if (nums[mid] < nums[right]) right = mid;
            else --right;
        }
        return nums[right];
    }
};
```

跟之前那道 Find Minimum in Rotated Sorted Array 一样，还是可以用分治法 Divide and Conquer 来解，还是由热心网友 howard144 提供，不过写法跟之前那道略有不同，只有在 nums[start] < nums[end] 的时候，才能返回 nums[start]，等于的时候不能返回，比如 [3, 1, 3] 这个数组，或者当 start 等于 end 成立的时候，也可以直接返回 nums[start]，后面的操作跟之前那道题相同，每次将区间 [start, end] 从中间 mid 位置分为两段，分别调用递归函数，并比较返回值，每次取返回值较小的那个即可，参见代码如下：

解法二：

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        return helper(nums, 0, (int)nums.size() - 1);
    }
    int helper(vector<int>& nums, int start, int end) {
        if (start == end) return nums[start];
        if (nums[start] < nums[end]) return nums[start];
        int mid = (start + end) / 2;
        return min(helper(nums, start, mid), helper(nums, mid + 1, end));
    }
};
```

Find Minimum in Rotated Sorted Array

156. Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Example:

```
Input: [1,2,3,4,5]

    1
   / \
  2   3
 / \
4   5

Output: return the root of the binary tree [4,5,2,#,#,3,1]

   4
  / \
 5   2
    / \
   3   1
```

Clarification:

Confused what `[4,5,2,#,#,3,1]` means? Read more below on how binary tree is serialized on OJ.

The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

```
    1
   / \
  2   3
     /
    4
     \
      5
```

The above binary tree is serialized as `[1,2,3,#,#,4,#,#,5]`.

这道题让我们把一棵二叉树上下颠倒一下，而且限制了右节点要么为空要么一定会有对应的左节点。上下颠倒后原来二叉树的最左子节点变成了根节点，其对应的右节点变成了其左子节点，其父节点变成了其右子节点，相当于顺时针旋转了一下。对于一般树的题都会有迭代和递归两种解法，这道题也不例外，先来看看递归的解法。对于一个根节点来说，目标是将其左子节点变为根节点，右子节点变为左子节点，原根节点变为右子节点，首先判断这个根节点是否存在，且其有没有左子节点，如果不满足这两个条件的话，直接返回即可，不需要翻转操作。那么不停的对左子节点调用递归函数，直到到达最左子节点开始翻转，翻转好最左子节点后，开始回到上一个左子节点继续翻转即可，直至翻转完整棵树，参见代码如下：

解法一：

```cpp
class Solution {
public:
    TreeNode *upsideDownBinaryTree(TreeNode *root) {
        if (!root || !root->left) return root;
        TreeNode *l = root->left, *r = root->right;
        TreeNode *res = upsideDownBinaryTree(l);
        l->left = r;
        l->right = root;
        root->left = NULL;
        root->right = NULL;
        return res;
    }
};
```

下面我们来看迭代的方法，和递归方法相反的时，这个是从上往下开始翻转，直至翻转到最左子节点，参见代码如下：

解法二：

```cpp
class Solution {
public:
    TreeNode *upsideDownBinaryTree(TreeNode *root) {
        TreeNode *cur = root, *pre = NULL, *next = NULL, *tmp = NULL;
        while (cur) {
            next = cur->left;
            cur->left = tmp;
            tmp = cur->right;
            cur->right = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
};
```

158. Read N Characters Given Read4 II - Call multiple times

Given a file and assume that you can only read the file using a given method `read4`, implement a method `read` to read *n* characters. Your method `read` may be called multiple times.

Method read4:

The API `read4` reads 4 consecutive characters from the file, then writes those characters into the buffer array `buf`.

The return value is the number of actual characters read.

Note that `read4()` has its own file pointer, much like `FILE *fp` in C.

Definition of read4:

```
    Parameter:  char[] buf
    Returns:    int

Note: buf[] is destination not source, the results from read4 will be copied to buf[]
```

Below is a high level example of how `read4` works:

```
File file("abcdefghijk"); // File is "abcdefghijk", initially file pointer (fp) points to 'a'
char[] buf = new char[4]; // Create buffer with enough space to store characters
read4(buf); // read4 returns 4. Now buf = "abcd", fp points to 'e'
read4(buf); // read4 returns 4. Now buf = "efgh", fp points to 'i'
read4(buf); // read4 returns 3. Now buf = "ijk", fp points to end of file
```

Method read:

By using the `read4` method, implement the method `read` that reads *n* characters from the file and store it in the buffer array `buf`. Consider that you cannot manipulate the file directly.

The return value is the number of actual characters read.

Definition of read:

```
    Parameters: char[] buf, int n
    Returns:    int

Note: buf[] is destination not source, you will need to write the results to buf[]
```

Example 1:

```
File file("abc");
Solution sol;
// Assume buf is allocated and guaranteed to have enough space for storing all characters from the file.
sol.read(buf, 1); // After calling your read method, buf should contain "a". We read a total of 1 character from the file, so return 1.
sol.read(buf, 2); // Now buf should contain "bc". We read a total of 2 characters from the file, so return 2.
sol.read(buf, 1); // We have reached the end of file, no more characters can be read. So return 0.
```

Example 2:

```
File file("abc");
Solution sol;
sol.read(buf, 4); // After calling your read method, buf should contain "abc". We read a total of 3 characters from the file, so return 3.
sol.read(buf, 1); // We have reached the end of file, no more characters can be read. So return 0.
```

Note:

1. Consider that you cannot manipulate the file directly, the file is only accesible for `read4` but not for `read`.
2. The `read` function may be called multiple times.
3. Please remember to RESET your class variables declared in Solution, as static/class variables are persisted across multiple test cases. Please see here for more details.
4. You may assume the destination buffer array, `buf`, is guaranteed to have enough space for storing *n* characters.
5. It is guaranteed that in a given test case the same buffer `buf` is called by `read`.

这道题是之前那道 Read N Characters Given Read4 的拓展，那道题说 read 函数只能调用一次，而这道题说 read 函数可以调用多次，那么难度就增加了，为了更简单直观的说明问题，举个简单的例子吧，比如：

buf = "ab", [read(1),read(2)]，返回 ["a","b"]

那么第一次调用 read(1) 后，从 buf 中读出一个字符，就是第一个字符a，然后又调用了一个 read(2)，想取出两个字符，但是 buf 中只剩一个b了，所以就把取出的结果就是b。再来看一个例子：

buf = "a", [read(0),read(1),read(2)]，返回 ["","a",""]

第一次调用 read(0)，不取任何字符，返回空，第二次调用 read(1)，取一个字符，buf 中只有一个字符，取出为a，然后再调用 read(2)，想取出两个字符，但是 buf 中没有字符了，所以取出为空。

但是这道题我不太懂的地方是明明函数返回的是 int 类型啊，为啥 OJ 的 output 都是 vector 类的，然后我就在网上找了下面两种能通过OJ的解法，大概看了看，也是看的个一知半解，貌似是用两个变量 readPos 和 writePos 来记录读取和写的位置，i从0到n开始循环，如果此时读和写的位置相同，那么调用 read4 函数，将结果赋给 writePos，把 readPos 置零，如果 writePos 为零的话，说明 buf 中没有东西了，返回当前的坐标i。然后用内置的 buff 变量的 readPos 位置覆盖输入字符串 buf 的i位置，如果完成遍历，返回n，参见代码如下：

解法一：

```cpp
// Forward declaration of the read4 API.
int read4(char *buf);

class Solution {
public:
    int read(char *buf, int n) {
        for (int i = 0; i < n; ++i) {
            if (readPos == writePos) {
                writePos = read4(buff);
                readPos = 0;
                if (writePos == 0) return i;
            }
            buf[i] = buff[readPos++];
        }
        return n;
    }
private:
    int readPos = 0, writePos = 0;
    char buff[4];
};
```

下面这种方法和上面的方法基本相同，稍稍改变了些解法，使得看起来更加简洁一些：

解法二：

```cpp
// Forward declaration of the read4 API.
int read4(char *buf);

class Solution {
public:
    int read(char *buf, int n) {
        int i = 0;
        while (i < n && (readPos < writePos || (readPos = 0) < (writePos = read4(buff))))
            buf[i++] = buff[readPos++];
        return i;
    }
    char buff[4];
    int readPos = 0, writePos = 0;
};
```

159. Longest Substring with At Most Two Distinct Characters

Given a string *s* , find the length of the longest substring *t* that contains at most 2 distinct characters.

Example 1:

```
Input: "eceba"
Output: 3
Explanation: _t_ is "ece" which its length is 3.
```

Example 2:

```
Input: "ccaabbb"
Output: 5
Explanation: _t_ is "aabbb" which its length is 5.
```

这道题给我们一个字符串，让求最多有两个不同字符的最长子串。那么首先想到的是用 HashMap 来做，HashMap 记录每个字符的出现次数，然后如果 HashMap 中的映射数量超过两个的时候，这里需要删掉一个映射，比如此时 HashMap 中e有2个，c有1个，此时把b也存入了 HashMap，那么就有三对映射了，这时 left 是0，先从e开始，映射值减1，此时e还有1个，不删除，left 自增1。这时 HashMap 里还有三对映射，此时 left 是1，那么到c了，映射值减1，此时e映射为0，将e从 HashMap 中删除，left 自增1，然后更新结果为 i - left + 1，以此类推直至遍历完整个字符串，参见代码如下：

解法一：

```
class Solution {
public:
    int lengthOfLongestSubstringTwoDistinct(string s) {
        int res = 0, left = 0;
        unordered_map<char, int> m;
        for (int i = 0; i < s.size(); ++i) {
            ++m[s[i]];
            while (m.size() > 2) {
                if (--m[s[left]] == 0) m.erase(s[left]);
                ++left;
            }
            res = max(res, i - left + 1);
        }
        return res;
    }
};
```

我们除了用 HashMap 来映射字符出现的个数，还可以映射每个字符最新的坐标，比如题目中的例子 "eceba"，遇到第一个e，映射其坐标0，遇到c，映射其坐标1，遇到第二个e时，映射其坐标2，当遇到b时，映射其坐标3，每次都判断当前 HashMap 中的映射数，如果大于2的时候，那么需要删掉一个映射，还是从 left=0 时开始向右找，看每个字符在 HashMap 中的映射值是否等于当前坐标 left，比如第一个e，HashMap 此时映射值为2，不等于 left 的0，那么 left 自增1，遇到c的时候，HashMap 中c的映射值是1，和此时的 left 相同，那么我们把c删掉，left 自增1，再更新结果，以此类推直至遍历完整个字符串，参见代码如下：

解法二：

```cpp
class Solution {
public:
    int lengthOfLongestSubstringTwoDistinct(string s) {
        int res = 0, left = 0;
        unordered_map<char, int> m;
        for (int i = 0; i < s.size(); ++i) {
            m[s[i]] = i;
            while (m.size() > 2) {
                if (m[s[left]] == left) m.erase(s[left]);
                ++left;
            }
            res = max(res, i - left + 1);
        }
        return res;
    }
};
```

后来又在网上看到了一种解法，这种解法是维护一个 sliding window，指针 left 指向起始位置，right 指向 window 的最后一个位置，用于定位 left 的下一个跳转位置，思路如下：

\1. 若当前字符和前一个字符相同，继续循环。

\2. 若不同，看当前字符和 right 指的字符是否相同

 (1) 若相同，left 不变，右边跳到 i - 1

 (2) 若不同，更新结果，left 变为 right+1，right 变为 i - 1

最后需要注意在循环结束后，还要比较结果 res 和 s.size() - left 的大小，返回大的，这是由于如果字符串是 "ecebaaa"，那么当 left=3 时，i=5,6 的时候，都是继续循环，当i加到7时，跳出了循环，而此时正确答案应为 "baaa" 这4个字符，而我们的结果 res 只更新到了 "ece" 这3个字符，所以最后要判断 s.size() - left 和结果 res 的大小。

另外需要说明的是这种解法仅适用于于不同字符数为2个的情况，如果为k个的话，还是需要用上面两种解法。

解法三：

```cpp
class Solution {
public:
    int lengthOfLongestSubstringTwoDistinct(string s) {
        int left = 0, right = -1, res = 0;
        for (int i = 1; i < s.size(); ++i) {
            if (s[i] == s[i - 1]) continue;
            if (right >= 0 && s[right] != s[i]) {
                res = max(res, i - left);
                left = right + 1;
            }
            right = i - 1;
        }
        return max(s.size() - left, res);
    }
};
```

还有一种不使用 HashMap 的解法，是在做 Fruit Into Baskets 这道题的时候在论坛上看到的，其实这两道题除了背景设定之外没有任何的区别，代码基本上都可以拷来直接用的。这里使用若干的变量，其中 cur 为当前最长子串的长度，first 和 second 为当前候选子串中的两个不同的字符，cntLast 为 second 字符的连续长度。遍历所有字符，假如遇到的字符是 first 和 second 中的任意一个，那么 cur 可以自增1，否则 cntLast 自增1，因为若是新字符的话，默认已经将 first 字符淘汰了，此时候选字符串由 second 字符和这个新字符构成，所以当前长度是 cntLast+1。然后再来更新 cntLast，假如当前字符等于 second 的话，cntLast 自增1，否则均重置为1，因为 cntLast 统计的就是 second 字符的连续长度。然后再来判断若当前字符不等于 second，则此时 first 赋值为 second， second 赋值为新字符。最后不要忘了用 cur 来更新结果 res，参见代码如下：

解法四：

```
class Solution {
public:
    int lengthOfLongestSubstringTwoDistinct(string s) {
        int res = 0, cur = 0, cntLast = 0;
        char first, second;
        for (char c : s) {
            cur = (c == first || c == second) ? cur + 1 : cntLast + 1;
            cntLast = (c == second) ? cntLast + 1 : 1;
            if (c != second) {
                first = second; second = c;
            }
            res = max(res, cur);
        }
        return res;
    }
};
```

160. Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

```
A:          a1 → a2
                   ↘
                     c1 → c2 → c3
                   ↗
B:     b1 → b2 → b3
```

begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return `null`.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

Credits:
Special thanks to @stellari for adding this problem and creating all test cases.

我还以为以后在不能免费做OJ的题了呢，想不到 OJ 又放出了不需要买书就能做的题，业界良心啊，哈哈~。这道求两个链表的交点题要求执行时间为 O(n)，则不能利用类似冒泡法原理去暴力查找相同点，事实证明如果链表很长的话，那样的方法效率很低。我也想到会不会是像之前删除重复元素的题一样需要用两个指针来遍历，可是想了好久也没想出来怎么弄。无奈上网搜大神们的解法，发觉其实解法很简单，因为如果两个链长度相同的话，那么对应的一个个比下去就能找到，所以只需要把长链表变短即可。具体算法为：分别遍历两个链表，得到分别对应的长度。然后求长度的差值，把较长的那个链表向后移动这个差值的个数，然后一一比较即可。代码如下：

C++ 解法一：

```cpp
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (!headA || !headB) return NULL;
        int lenA = getLength(headA), lenB = getLength(headB);
        if (lenA < lenB) {
            for (int i = 0; i < lenB - lenA; ++i) headB = headB->next;
        } else {
            for (int i = 0; i < lenA - lenB; ++i) headA = headA->next;
        }
        while (headA && headB && headA != headB) {
            headA = headA->next;
            headB = headB->next;
        }
        return (headA && headB) ? headA : NULL;
    }
    int getLength(ListNode* head) {
        int cnt = 0;
        while (head) {
            ++cnt;
            head = head->next;
        }
        return cnt;
    }
};
```

Java 解法一：

```java
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) return null;
        int lenA = getLength(headA), lenB = getLength(headB);
        if (lenA > lenB) {
            for (int i = 0; i < lenA - lenB; ++i) headA = headA.next;
        } else {
            for (int i = 0; i < lenB - lenA; ++i) headB = headB.next;
        }
        while (headA != null && headB != null && headA != headB) {
            headA = headA.next;
            headB = headB.next;
        }
        return (headA != null && headB != null) ? headA : null;
    }
    public int getLength(ListNode head) {
        int cnt = 0;
        while (head != null) {
            ++cnt;
            head = head.next;
        }
        return cnt;
    }
}
```

这道题还有一种特别巧妙的方法，虽然题目中强调了链表中不存在环，但是我们可以用环的思想来做，我们让两条链表分别从各自的开头开始往后遍历，当其中一条遍历到末尾时，我们跳到另一个条链表的开头继续遍历。两个指针最终会相等，而且只有两种情况，一种情况是在交点处相遇，另一种情况是在各自的末尾的空节点处相等。为什么一定会相等呢，因为两个指针走过的路程相同，是两个链表的长度之和，所以一定会相等。这个思路真的很巧妙，而且更重要的是代码写起来特别的简洁，参见代码如下：

C++ 解法二：

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (!headA || !headB) return NULL;
        ListNode *a = headA, *b = headB;
        while (a != b) {
            a = a ? a->next : headB;
            b = b ? b->next : headA;
        }
        return a;
    }
};
```

Java 解法二：

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) return null;
        ListNode a = headA, b = headB;
        while (a != b) {
            a = (a != null) ? a.next : headB;
            b = (b != null) ? b.next : headA;
        }
        return a;
    }
}
```

类似题目：

Minimum Index Sum of Two Lists

161. One Edit Distance

Given two strings *s* and *t* , determine if they are both one edit distance apart.

Note:

There are 3 possiblities to satisify one edit distance apart:

1. Insert a character into *s* to get *t*
2. Delete a character from *s* to get *t*
3. Replace a character of *s* to get *t*

Example 1:

```
Input: _s_ = "ab", _t_ = "acb"
Output: true
Explanation: We can insert 'c' into _s_  to get  _t._
```

Example 2:

```
Input: _s_ = "cab", _t_ = "ad"
Output: false
Explanation: We cannot get _t_ from _s_ by only one step.
```

Example 3:

```
Input: _s_ = "1203", _t_ = "1213"
Output: true
Explanation: We can replace '0' with '1' to get  _t._
```

这道题是之前那道 Edit Distance 的拓展，然而这道题并没有那道题难，这道题只让我们判断两个字符串的编辑距离是否为1，那么只需分下列三种情况来考虑就行了：

\1. 两个字符串的长度之差大于1，直接返回False。

\2. 两个字符串的长度之差等于1，长的那个字符串去掉一个字符，剩下的应该和短的字符串相同。

\3. 两个字符串的长度之差等于0，两个字符串对应位置的字符只能有一处不同。

分析清楚了所有的情况，代码就很好写了，参见如下：

解法一：

```cpp
class Solution {
public:
    bool isOneEditDistance(string s, string t) {
        if (s.size() < t.size()) swap(s, t);
        int m = s.size(), n = t.size(), diff = m - n;
        if (diff >= 2) return false;
        else if (diff == 1) {
            for (int i = 0; i < n; ++i) {
                if (s[i] != t[i]) {
                    return s.substr(i + 1) == t.substr(i);
                }
            }
            return true;
        } else {
            int cnt = 0;
            for (int i = 0; i < m; ++i) {
                if (s[i] != t[i]) ++cnt;
            }
            return cnt == 1;
        }
    }
};
```

我们实际上可以让代码写的更加简洁，只需要对比两个字符串对应位置上的字符，如果遇到不同的时候，这时看两个字符串的长度关系，如果相等，则比较当前位置后的字串是否相同，如果s的长度大，那么比较s的下一个位置开始的子串，和t的当前位置开始的子串是否相同，反之如果t的长度大，则比较t的下一个位置开始的子串，和s的当前位置开始的子串是否相同。如果循环结束，都没有找到不同的字符，那么此时看两个字符串的长度是否相差1，参见代码如下：

解法二：

```
class Solution {
public:
    bool isOneEditDistance(string s, string t) {
        for (int i = 0; i < min(s.size(), t.size()); ++i) {
            if (s[i] != t[i]) {
                if (s.size() == t.size()) return s.substr(i + 1) == t.substr(i + 1);
                if (s.size() < t.size()) return s.substr(i) == t.substr(i + 1);
                else return s.substr(i + 1) == t.substr(i);
            }
        }
        return abs((int)s.size() - (int)t.size()) == 1;
    }
};
```

162. Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array `nums`, where `nums[i] ≠ nums[i+1]`, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that `nums[-1] = nums[n] = -∞`.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 2
Explanation: 3 is a peak element and your function should return the index number 2.
```

Example 2:

```
Input: nums = [1,2,1,3,5,6,4]
Output: 1 or 5
Explanation: Your function can return either index number 1 where the peak element is 2,
             or index number 5 where the peak element is 6.
```

Note:

Your solution should be in logarithmic complexity.

这道题是求数组的一个峰值，如果这里用遍历整个数组找最大值肯定会出现Time Limit Exceeded，但题目中说了这个峰值可以是局部的最大值，所以我们只需要找到第一个局部峰值就可以了。所谓峰值就是比周围两个数字都大的数字，那么只需要跟周围两个数字比较就可以了。既然要跟左右的数字比较，就得考虑越界的问题，题目中给了nums[-1] = nums[n] = -∞，那么我们其实可以把这两个整型最小值直接加入到数组中，然后从第二个数字遍历到倒数第二个数字，这样就不会存在越界的可能了。由于题目中说了峰值一定存在，那么有一个很重要的corner case我们要注意，就是当原数组中只有一个数字，且是整型最小值的时候，我们如果还要首尾垫数字，就会形成一条水平线，从而没有峰值了，所以我们对于数组中只有一个数字的情况在开头直接判断一下即可，参见代码如下：

C++ 解法一：

```cpp
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        if (nums.size() == 1) return 0;
        nums.insert(nums.begin(), INT_MIN);
        nums.push_back(INT_MIN);
        for (int i = 1; i < (int)nums.size() - 1; ++i) {
            if (nums[i] > nums[i - 1] && nums[i] > nums[i + 1]) return i - 1;
        }
        return -1;
    }
};
```

Java 解法一：

```java
class Solution {
    public int findPeakElement(int[] nums) {
        if (nums.length == 1) return 0;
        int[] newNums = new int[nums.length + 2];
        System.arraycopy(nums, 0, newNums, 1, nums.length);
        newNums[0] = Integer.MIN_VALUE;
        newNums[newNums.length - 1] = Integer.MIN_VALUE;
        for (int i = 1; i < newNums.length - 1; ++i) {
            if (newNums[i] > newNums[i - 1] && newNums[i] > newNums[i + 1]) return i - 1;
        }
        return -1;
    }
}
```

我们可以对上面的线性扫描的方法进行一些优化，可以省去首尾垫值的步骤。由于题目中说明了局部峰值一定存在，那么实际上可以从第二个数字开始往后遍历，如果第二个数字比第一个数字小，说明此时第一个数字就是一个局部峰值；否则就往后继续遍历，现在是个递增趋势，如果此时某个数字小于前面那个数字，说明前面数字就是一个局部峰值，返回位置即可。如果循环结束了，说明原数组是个递增数组，返回最后一个位置即可，参见代码如下：

C++ 解法二：

```cpp
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] < nums[i - 1]) return i - 1;
        }
        return nums.size() - 1;
    }
};
```

Java 解法二：

```java
public class Solution {
    public int findPeakElement(int[] nums) {
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] < nums[i - 1]) return i - 1;
        }
        return nums.length - 1;
    }
}
```

由于题目中提示了要用对数级的时间复杂度，那么我们就要考虑使用类似于二分查找法来缩短时间，由于只是需要找到任意一个峰值，那么我们在确定二分查找折半后中间那个元素后，和紧跟的那个元素比较下大小，如果大于，则说明峰值在前面，如果小于则在后面。这样就可以找到一个峰值了，代码如下：

C++ 解法三：

```cpp
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < nums[mid + 1]) left = mid + 1;
            else right = mid;
        }
        return right;
    }
};
```

Java 解法三：

```java
public class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < nums[mid + 1]) left = mid + 1;
            else right = mid;
        }
        return right;
    }
}
```

163. Missing Ranges

Given a sorted integer array *nums* , where the range of elements are in the inclusive range [*lower* , *upper*], return its missing ranges.

Example:

```
Input: _nums_ = [0, 1, 3, 50, 75], _lower_ = 0 and _upper_ = 99,
Output: ["2", "4->49", "51->74", "76->99"]
```

这道题让我们求缺失区间，跟之前那道 Summary Ranges 很类似，给了一个空间的范围 [lower upper]，缺失的区间的范围需要在给定的区间范围内。遍历 nums 数组，假如当前数字 num 大于 lower，说明此时已经有缺失区间，至少缺失一个 lower 数字，此时若 num-1 大于 lower，说明缺失的是一个区间 [lower, num-1]，否则就只加入一个数字即可。由于 OJ 之后加入了许多 tricky 的 test cases，使得论坛上很多解法都 fail 了。其实很多是跪在了整型溢出，当数组中有整型最大值时，此时 lower 更新为 num+1 时就会溢出，所以在更新之前要先判断一下，若 num 已经是整型最大值了，直接返回结果 res 即可；否则才更新 lower 继续循环。for 循环退出后，此时可能还存在缺失区间，就是此

时 lower 还小于等于 upper 时，可以会缺失 lower 这个数字，或者 [lower, upper] 区间，最后补上这个区间就可以通过啦，参见代码如下：

```cpp
class Solution {
public:
    vector<string> findMissingRanges(vector<int>& nums, int lower, int upper) {
        vector<string> res;
        for (int num : nums) {
            if (num > lower) res.push_back(to_string(lower) + (num - 1 > lower ? ("->" + to_string(num - 1)) : ""));
            if (num == upper) return res;
            lower = num + 1;
        }
        if (lower <= upper) res.push_back(to_string(lower) + (upper > lower ? ("->" + to_string(upper)) : ""));
        return res;
    }
};
```

164. Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Return 0 if the array contains less than 2 elements.

Example 1:

```
Input: [3,6,9,1]
Output: 3
Explanation: The sorted form of the array is [1,3,6,9], either
             (3,6) or (6,9) has the maximum difference 3.
```

Example 2:

```
Input: [10]
Output: 0
Explanation: The array contains less than 2 elements, therefore return 0.
```

Note:

- You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.
- Try to solve it in linear time/space.

遇到这类问题肯定先想到的是要给数组排序，但是题目要求是要线性的时间和空间，那么只能用桶排序或者基排序。这里用桶排序 Bucket Sort 来做，首先找出数组的最大值和最小值，然后要确定每个桶的容量，即为 (最大值 - 最小值) / 个数 + 1，在确定桶的个数，即为 (最大值 - 最小值) / 桶的容量 + 1，然后需要在每个桶中找出局部最大值和最小值，而最大间距的两个数不会在同一个桶中，而是一个桶的最小值和另一个桶的最大值之间的间距，这是因为所有的数字要尽量平均分配到每个桶中，而不是都拥挤在一个桶中，这样保证了最大值和最小值一定不会在同一个桶中，具体的证明博主也不会，只是觉得这样想挺有道理的，各位看官大神们若知道如何证明请务必留言告诉博主啊，参见代码如下：

```cpp
class Solution {
public:
    int maximumGap(vector<int>& nums) {
        if (nums.size() <= 1) return 0;
        int mx = INT_MIN, mn = INT_MAX, n = nums.size(), pre = 0, res = 0;
        for (int num : nums) {
            mx = max(mx, num);
            mn = min(mn, num);
        }
        int size = (mx - mn) / n + 1, cnt = (mx - mn) / size + 1;
        vector<int> bucket_min(cnt, INT_MAX), bucket_max(cnt, INT_MIN);
        for (int num : nums) {
            int idx = (num - mn) / size;
            bucket_min[idx] = min(bucket_min[idx], num);
            bucket_max[idx] = max(bucket_max[idx], num);
        }
        for (int i = 1; i < cnt; ++i) {
            if (bucket_min[i] == INT_MAX || bucket_max[i] == INT_MIN) continue;
            res = max(res, bucket_min[i] - bucket_max[pre]);
            pre = i;
        }
        return res;
    }
};
```

166. Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

- Given numerator = 1, denominator = 2, return "0.5".
- Given numerator = 2, denominator = 1, return "2".
- Given numerator = 2, denominator = 3, return "0.(6)".

Credits:
Special thanks to @Shangrila for adding this problem and creating all test cases.

这道题还是比较有意思的，开始还担心万一结果是无限不循环小数怎么办，百度之后才发现原来可以写成分数的都是有理数，而有理数要么是有限的，要么是无限循环小数，无限不循环的叫无理数，例如圆周率pi或自然数e等，小学数学没学好，汗！由于还存在正负情况，处理方式是按正数处理，符号最后在判断，那么我们需要把除数和被除数取绝对值，那么问题就来了：由于整型数INT的取值范围是-2147483648～2147483647，而对-2147483648取绝对值就会超出范围，所以我们需要先转为long long型再取绝对值。那么怎么样找循环呢，肯定是再得到一个数字后要看看之前有没有出现这个数。为了节省搜索时间，我们采用哈希表来存数每个小数位上的数字。还有一个小技巧，由于我们要算出小数每一位，采取的方法是每次把余数乘10，再除以除数，得到的商即为小数的下一位数字。等到新算出来的数字在之前出现过，则在循环开始出加左括号，结束处加右括号。代码如下：

```
class Solution {
public:
    string fractionToDecimal(int numerator, int denominator) {
        int s1 = numerator >= 0 ? 1 : -1;
        int s2 = denominator >= 0 ? 1 : -1;
        long long num = abs( (long long)numerator );
        long long den = abs( (long long)denominator );
        long long out = num / den;
        long long rem = num % den;
        unordered_map<long long, int> m;
        string res = to_string(out);
        if (s1 * s2 == -1 && (out > 0 || rem > 0)) res = "-" + res;
        if (rem == 0) return res;
        res += ".";
        string s = "";
        int pos = 0;
        while (rem != 0) {
            if (m.find(rem) != m.end()) {
                s.insert(m[rem], "(");
                s += ")";
                return res + s;
            }
            m[rem] = pos;
            s += to_string((rem * 10) / den);
            rem = (rem * 10) % den;
            ++pos;
        }
        return res + s;
    }
};
```

167. Two Sum II - Input array is sorted

Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a specific target number.
The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.
You may assume that each input would have exactly one solution.
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2

这又是一道Two Sum的衍生题，作为LeetCode开山之题，我们务必要把Two Sum及其所有的衍生题都拿下，这道题其实应该更容易一些，因为给定的数组是有序的，而且题目中限定了一定会有解，我最开始想到的方法是二分法来搜索，因为一定有解，而且数组是有序的，那么第一个数字肯定要小于目标值target，那么我们每次用二分法来搜索target - numbers[i]即可，代码如下：

解法一：

```
// O(nlgn)
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        for (int i = 0; i < numbers.size(); ++i) {
            int t = target - numbers[i], left = i + 1, right =
numbers.size();
            while (left < right) {
                int mid = left + (right - left) / 2;
                if (numbers[mid] == t) return {i + 1, mid + 1};
                else if (numbers[mid] < t) left = mid + 1;
                else right = mid;
            }
        }
        return {};
    }
};
```

但是上面那种方法并不efficient，时间复杂度是O(nlgn)，我们再来看一种O(n)的解法，我们只需要两个指针，一个指向开头，一个指向末尾，然后向中间遍历，如果指向的两个数相加正好等于target的话，直接返回两个指针的位置即可，若小于target，左指针右移一位，若大于target，右指针左移一位，以此类推直至两个指针相遇停止，参见代码如下：

解法二：

```
// O(n)
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int l = 0, r = numbers.size() - 1;
        while (l < r) {
            int sum = numbers[l] + numbers[r];
            if (sum == target) return {l + 1, r + 1};
```

```
                else if (sum < target) ++l;
                else --r;
            }
            return {};
        }
};
```

类似题目：

Two Sum III - Data structure design

173. Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling `next()` will return the next smallest number in the BST.

Note: `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

这道题主要就是考二叉树的中序遍历的非递归形式，需要额外定义一个栈来辅助，二叉搜索树的建树规则就是左<根<右，用中序遍历即可从小到大取出所有节点。代码如下：

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class BSTIterator {
public:
    BSTIterator(TreeNode *root) {
        while (root) {
            s.push(root);
            root = root->left;
        }
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !s.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *n = s.top();
        s.pop();
        int res = n->val;
        if (n->right) {
            n = n->right;
            while (n) {
                s.push(n);
                n = n->left;
            }
        }
        return res;
    }
private:
    stack<TreeNode*> s;
};

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = BSTIterator(root);
 * while (i.hasNext()) cout << i.next();
 */
```

174. Dungeon Game

The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN`.

-2 (K)	**-3**	**3**
-5	-10	1
10	30	-5 (P)

Note:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Credits:
Special thanks to @stellari for adding this problem and creating all test cases.

这道王子救公主的题还是蛮新颖的，我最开始的想法是比较右边和下边的数字的大小，去大的那个，但是这个算法对某些情况不成立，比如下面的情况：

1 (K)	**-3**	**3**
0	-2	0
-3	-3	-3 (P)

如果按我的那种算法走的路径为 1 -> 0 -> -2 -> 0 -> -3, 这样的话骑士的起始血量要为5，而正确的路径应为 1 -> -3 -> 3 -> 0 -> -3, 这样骑士的骑士血量只需为3。无奈只好上网看大神的解法，发现统一都是用动态规划 Dynamic Programming 来做，建立一个二维数组 dp，其中 dp[i][j] 用来表示当前位置 (i, j) 出发的起始血量，最先处理的是公主所在的房间的起始生命值，然后慢慢向第一个房间扩散，不断的得到各个位置的最优的生命值。逆向推正是本题的精髓所在啊，仔细想想也是，如果从起始位置开始遍历，我们并不知道初始时应该初始化的血量，但是到达公主房间后，我们知道血量至少不能小于1，如果公主房间还需要掉血的话，那么掉血后剩1才能保证起始位置的血量最小。那么下面来推导状态转移方程，首先考虑每个位置的血量是由什么决定的，骑士会挂主要是因为去了下一个房间时，掉血量大于本身的血值，而能去的房间只有右边和下边，所以当前位置的血量是由右边和下边房间的可生存血量决定的，进一步来说，应该是由较小的可生存血量决定的，因为较我们需要起始血量尽可能的少，因为我们是逆着往回推，骑士逆向进入房间后 PK 后所剩的血量就是骑士正向进入房间时 pk 前的起始血量。所以用当前房间的右边和下边房间中骑士的较小血量减去当前房间的数字，如果是负数或着0，说明当前房间是正数，这样骑士进入当前房间后的生命值是1就行了，因为不会减血。而如果差是正数的话，当前房间的血量可能是正数也可能是负数，但是骑士进入当前房间后的生命值就一定要是这个差值。所以我们的状态转移方程是 dp[i][j] = max(1, min(dp[i+1][j], dp[i][j+1]) - dungeon[i][j])。为了更好的处理边界情况，我们的二维 dp 数组比原数组的行数列数均多1个，先都初始化为整型数最大值 INT_MAX，由于我们知道到达公主房间后，骑士火拼完的血量至少为1，那么此时公主房间的右边和下边房间里的数字我们就都设置为1，这样到达公主房间的生存血量就是1减去公主房间的数字和1相比较，取较大值，就没有问题了，代码如下：

解法一：

```
class Solution {
public:
    int calculateMinimumHP(vector<vector<int>>& dungeon) {
        int m = dungeon.size(), n = dungeon[0].size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, INT_MAX));
        dp[m][n - 1] = 1; dp[m - 1][n] = 1;
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                dp[i][j] = max(1, min(dp[i + 1][j], dp[i][j + 1]) - 
dungeon[i][j]);
            }
        }
        return dp[0][0];
    }
};
```

我们可以对空间进行优化，使用一个一维的 dp 数组，并且不停的覆盖原有的值，参见代码如下：

解法二：

```
class Solution {
public:
    int calculateMinimumHP(vector<vector<int>>& dungeon) {
        int m = dungeon.size(), n = dungeon[0].size();
        vector<int> dp(n + 1, INT_MAX);
        dp[n - 1] = 1;
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                dp[j] = max(1, min(dp[j], dp[j + 1]) - dungeon[i][j]);
            }
        }
        return dp[0];
    }
};
```

177. Nth Highest Salary

Write a SQL query to get the *n* th highest salary from the `Employee` table.

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

For example, given the above Employee table, the *n* th highest salary where *n* = 2 is `200`. If there is no *n* th highest salary, then the query should return `null`.

这道题是之前那道Second Highest Salary的拓展，根据之前那道题的做法，我们可以很容易的将其推展为N，根据对Second Highest Salary中解法一的分析，我们只需要将OFFSET后面的1改为N-1就行了，但是这样MySQL会报错，估计不支持运算，那么我们可以在前面加一个SET N = N - 1，将N先变成N-1再做也是一样的：

解法一：

```sql
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
  SET N = N - 1;
  RETURN (
      SELECT DISTINCT Salary FROM Employee GROUP BY Salary
      ORDER BY Salary DESC LIMIT 1 OFFSET N
  );
END
```

根据对Second Highest Salary中解法四的分析，我们只需要将其1改为N-1即可，这里却支持N-1的计算，参见代码如下：

解法二：

```sql
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
  RETURN (
      SELECT MAX(Salary) FROM Employee E1
      WHERE N - 1 =
      (SELECT COUNT(DISTINCT(E2.Salary)) FROM Employee E2
      WHERE E2.Salary > E1.Salary)
  );
END
```

当然我们也可以通过将最后的>改为>=，这样我们就可以将N-1换成N了：

解法三：

```sql
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
  RETURN (
      SELECT MAX(Salary) FROM Employee E1
      WHERE N =
      (SELECT COUNT(DISTINCT(E2.Salary)) FROM Employee E2
      WHERE E2.Salary >= E1.Salary)
  );
END
```

类似题目：

Second Highest Salary

178. Rank Scores

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

For example, given the above `Scores` table, your query should generate the following report (order by highest score):

```
+-------+------+
| Score | Rank |
+-------+------+
| 4.00  | 1    |
| 4.00  | 1    |
| 3.85  | 2    |
| 3.65  | 3    |
| 3.65  | 3    |
| 3.50  | 4    |
+-------+------+
```

这道题给了我们一个分数表，让我们给分数排序，要求是相同的分数在相同的名次，下一个分数在相连的下一个名次，中间不能有空缺数字，这道题我是完全照着史蒂芬大神的帖子来写的，膜拜大神中...大神总结了四种方法，那么我们一个一个的来膜拜学习，首先看第一种解法，解题的思路是对于每一个分数，找出表中有多少个大于或等于该分数的不同的分数，然后按降序排列即可，参见代码如下：

解法一：

```
SELECT Score,
(SELECT COUNT(DISTINCT Score) FROM Scores WHERE Score >= s.Score) Rank
FROM Scores s ORDER BY Score DESC;
```

跟上面的解法思想相同，就是写法上略有不同：

解法二：

```sql
SELECT Score,
(SELECT COUNT(*) FROM (SELECT DISTINCT Score s FROM Scores) t WHERE s >= Score) Rank
FROM Scores ORDER BY Score DESC;
```

下面这种解法使用了内交，Join是Inner Join的简写形式，自己和自己内交，条件是右表的分数大于等于左表，然后群组起来根据分数的降序排列，十分巧妙的解法：

解法三：

```sql
SELECT s.Score, COUNT(DISTINCT t.Score) Rank
FROM Scores s JOIN Scores t ON s.Score <= t.Score
GROUP BY s.Id ORDER BY s.Score DESC;
```

下面这种解法跟上面三种的画风就不太一样了，这里用了两个变量，变量使用时其前面需要加@，这里的：= 是赋值的意思，如果前面有Set关键字，则可以直接用=号来赋值，如果没有，则必须要使用:=来赋值，两个变量rank和pre，其中rank表示当前的排名，pre表示之前的分数，下面代码中的<>表示不等于，如果左右两边不相等，则返回true或1，若相等，则返回false或0。初始化rank为0，pre为-1，然后按降序排列分数，对于分数4来说，pre赋为4，和之前的pre值-1不同，所以rank要加1，那么分数4的rank就为1，下面一个分数还是4，那么pre赋值为4和之前的4相同，所以rank要加0，所以这个分数4的rank也是1，以此类推就可以计算出所有分数的rank了。

解法四：

```sql
SELECT Score,
@rank := @rank + (@pre <> (@pre := Score)) Rank
FROM Scores, (SELECT @rank := 0, @pre := -1) INIT
ORDER BY Score DESC;
```

179. Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

Example 1:

```
Input: [10,2]
Output: "210"
```

Example 2:

```
Input: [3,30,34,5,9]
Output: "9534330"
```

Note: The result may be very large, so you need to return a string instead of an integer.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

这道题给了我们一个数组，让将其拼接成最大的数，那么根据题目中给的例子来看，主要就是要给数组进行排序，但是排序方法不是普通的升序或者降序，因为9要排在最前面，而9既不是数组中最大的也不是最小的，所以要自定义排序方法。如果不参考网友的解法，博主估计是无法想出来的。这种解法对于两个数字a和b来说，如果将其都转为字符串，如果 ab > ba，则a排在前面，比如9和34，由于934>349，所以9排在前面，再比如说 30 和3，由于 303<330，所以3排在 30 的前面。按照这种规则对原数组进行排序后，将每个数字转化为字符串再连接起来就是最终结果。代码如下：

```
class Solution {
public:
    string largestNumber(vector<int>& nums) {
        string res;
        sort(nums.begin(), nums.end(), [](int a, int b) {
           return to_string(a) + to_string(b) > to_string(b) +
to_string(a);
        });
        for (int i = 0; i < nums.size(); ++i) {
            res += to_string(nums[i]);
        }
        return res[0] == '0' ? "0" : res;
    }
};
```

180. Consecutive Numbers

Write a SQL query to find all numbers that appear at least three times consecutively.

```
+----+-----+
| Id | Num |
+----+-----+
| 1  |  1  |
| 2  |  1  |
| 3  |  1  |
| 4  |  2  |
| 5  |  1  |
| 6  |  2  |
| 7  |  2  |
+----+-----+
```

For example, given the above `Logs` table, `1` is the only number that appears consecutively for at least three times.

这道题给了我们一个Logs表，让我们找Num列中连续出现相同数字三次的数字，那么由于需要找三次相同数字，所以我们需要建立三个表的实例，我们可以用l1分别和l2, l3内交，l1和l2的Id下一个位置比，l1和l3的下两个位置比，然后将Num都相同的数字返回即可：

解法一:

```sql
SELECT DISTINCT l1.Num FROM Logs l1
JOIN Logs l2 ON l1.Id = l2.Id - 1
JOIN Logs l3 ON l1.Id = l3.Id - 2
WHERE l1.Num = l2.Num AND l2.Num = l3.Num;
```

下面这种方法没用用到Join，而是直接在三个表的实例中查找，然后把四个条件限定上，就可以返回正确结果了：

解法二:

```sql
SELECT DISTINCT l1.Num FROM Logs l1, Logs l2, Logs l3
WHERE l1.Id = l2.Id - 1 AND l2.Id = l3.Id - 1
AND l1.Num = l2.Num AND l2.Num = l3.Num;
```

再来看一种画风截然不同的方法，用到了变量count和pre，分别初始化为0和-1，然后需要注意的是用到了IF语句，MySQL里的IF语句和我们所熟知的其他语言的if不太一样，相当于我们所熟悉的三元操作符a?b:c，若a真返回b，否则返回c，具体可看这个帖子。那么我们先来看对于Num列的第一个数字1，pre由于初始化是-1，和当前Num不同，所以此时count赋1，此时给pre赋为1，然后Num列的第二个1进来，此时的pre和Num相同了，count自增1，到Num列的第三个1进来，count增加到了3，此时满足了where条件，t.n >= 3，所以1就被select出来了，以此类推遍历完整个Num就可以得到最终结果：

解法三：

```sql
SELECT DISTINCT Num FROM (
SELECT Num, @count := IF(@pre = Num, @count + 1, 1) AS n, @pre := Num
FROM Logs, (SELECT @count := 0, @pre := -1) AS init
) AS t WHERE t.n >= 3;
```

184. Department Highest Salary

The `Employee` table holds all employees. Every employee has an Id, a salary, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
+----+-------+--------+--------------+
```

The `Department` table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who have the highest salary in each of the departments. For the above tables, Max has the highest salary in the IT department and Henry has the highest salary in the Sales department.

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| Sales      | Henry    | 80000  |
+------------+----------+--------+
```

这道题让给了我们两张表，Employee表和Department表，让我们找系里面薪水最高的人的，实际上这题是Second Highest Salary和Combine Two Tables的结合题，我们既需要联合两表，又要找到最高薪水，那么我们首先让两个表内交起来，然后将结果表需要的列都标明，然后就是要找最高的薪水，我们用Max关键字来实现，参见代码如下：

解法一：

```sql
SELECT d.Name AS Department, e1.Name AS Employee, e1.Salary FROM Employee e1
JOIN Department d ON e1.DepartmentId = d.Id WHERE Salary IN
(SELECT MAX(Salary) FROM Employee e2 WHERE e1.DepartmentId = e2.DepartmentId);
```

我们也可以不用Join关键字，直接用Where将两表连起来，然后找最高薪水的方法和上面相同：

解法二：

```sql
SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e, Department d
WHERE e.DepartmentId = d.Id AND e.Salary = (SELECT MAX(Salary) FROM Employee e2 WHERE e2.DepartmentId = d.Id);
```

下面这种方法没用用到Max关键字，而是用了>=符号，实现的效果跟Max关键字相同，参见代码如下：

解法三：

```sql
SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e, Department d
WHERE e.DepartmentId = d.Id AND e.Salary >= ALL (SELECT Salary FROM Employee e2 WHERE e2.DepartmentId = d.Id);
```

185. Department Top Three Salaries

The `Employee` table holds all employees. Every employee has an Id, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
+----+-------+--------+--------------+
```

The `Department` table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who earn the top three salaries in each of the department. For the above tables, your SQL query should return the following rows.

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Randy    | 85000  |
| IT         | Joe      | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
```

这道题是之前那道Department Highest Salary的拓展，难度标记为Hard，还是蛮有难度的一道题，综合了前面很多题的知识点，首先看使用Select Count(Distinct)的方法，我们内交Employee和Department两张表，然后我们找出比当前薪水高的最多只能有两个，那么前三高的都能被取出来了，参见代码如下：

解法一：

```
SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e
JOIN Department d on e.DepartmentId = d.Id
WHERE (SELECT COUNT(DISTINCT Salary) FROM Employee WHERE Salary > e.Salary
AND DepartmentId = d.Id) < 3 ORDER BY d.Name, e.Salary DESC;
```

下面这种方法将上面方法中的<3换成了IN (0, 1, 2)，是一样的效果：

解法二：

```
SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM Employee e,
Department d
WHERE (SELECT COUNT(DISTINCT Salary) FROM Employee WHERE Salary > e.Salary
AND DepartmentId = d.Id) IN (0, 1, 2) AND e.DepartmentId = d.Id ORDER BY
d.Name, e.Salary DESC;
```

或者我们也可以使用Group by Having Count(Distinct ..) 关键字来做：

解法三：

```
SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM
(SELECT e1.Name, e1.Salary, e1.DepartmentId FROM Employee e1 JOIN Employee
e2
ON e1.DepartmentId = e2.DepartmentId AND e1.Salary <= e2.Salary GROUP BY
e1.Id
HAVING COUNT(DISTINCT e2.Salary) <= 3) e JOIN Department d ON
e.DepartmentId = d.Id
ORDER BY d.Name, e.Salary DESC;
```

下面这种方法略微复杂一些，用到了变量，跟Consecutive Numbers中的解法三使用的方法一样，目的是为了给每个人都按照薪水的高低增加一个rank，最后返回rank值小于等于3的项即可，参见代码如下：

解法四：

```
SELECT d.Name AS Department, e.Name AS Employee, e.Salary FROM
(SELECT Name, Salary, DepartmentId,
@rank := IF(@pre_d = DepartmentId, @rank + (@pre_s <> Salary), 1) AS rank,
@pre_d := DepartmentId, @pre_s := Salary
FROM Employee, (SELECT @pre_d := -1, @pre_s := -1, @rank := 1) AS init
ORDER BY DepartmentId, Salary DESC) e JOIN Department d ON e.DepartmentId =
d.Id
WHERE e.rank <= 3 ORDER BY d.Name, e.Salary DESC;
```

186. Reverse Words in a String II

Given an input string __ , reverse the string word by word.

Example:

```
Input:  ["t","h","e"," ","s","k","y"," ","i","s"," ","b","l","u","e"]
Output: ["b","l","u","e"," ","i","s"," ","s","k","y"," ","t","h","e"]
```

Note:

- A word is defined as a sequence of non-space characters.
- The input string does not contain leading or trailing spaces.
- The words are always separated by a single space.

Follow up: Could you do it *in-place* without allocating extra space?

这道题让我们翻转一个字符串中的单词，跟之前那题 Reverse Words in a String 没有区别，由于之前那道题就是用 in-place 的方法做的，而这道题反而更简化了题目，因为不考虑首尾空格了和单词之间的多空格了，方法还是很简单，先把每个单词翻转一遍，再把整个字符串翻转一遍，或者也可以调换个顺序，先翻转整个字符串，再翻转每个单词，参见代码如下：

解法一：

```
class Solution {
public:
    void reverseWords(vector<char>& str) {
        int left = 0, n = str.size();
        for (int i = 0; i <= n; ++i) {
            if (i == n || str[i] == ' ') {
                reverse(str, left, i - 1);
                left = i + 1;
            }
        }
        reverse(str, 0, n - 1);
    }
    void reverse(vector<char>& str, int left, int right) {
        while (left < right) {
            char t = str[left];
```

```
                str[left] = str[right];
                str[right] = t;
                ++left; --right;
            }
        }
};
```

我们也可以使用 C++ STL 中自带的 reverse 函数来做，先把整个字符串翻转一下，然后再来扫描每个字符，用两个指针，一个指向开头，另一个开始遍历，遇到空格停止，这样两个指针之间就确定了一个单词的范围，直接调用 reverse 函数翻转，然后移动头指针到下一个位置，在用另一个指针继续扫描，重复上述步骤即可，参见代码如下：

解法二：

```
class Solution {
public:
    void reverseWords(vector<char>& str) {
        reverse(str.begin(), str.end());
        for (int i = 0, j = 0; i < str.size(); i = j + 1) {
            for (j = i; j < str.size(); ++j) {
                if (str[j] == ' ') break;
            }
            reverse(str.begin() + i, str.begin() + j);
        }
    }
};
```

187. Repeated DNA Sequences

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

Example:

```
Input: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

Output: ["AAAAACCCCC", "CCCCCAAAAA"]
```

看到这道题想到这应该属于 CS 的一个重要分支生物信息 Bioinformatics 研究的内容，研究 DNA 序列特征的重要意义自然不用多说，但是对于我们广大码农来说，还是专注于算法吧，此题还是用位操作 Bit Manipulation 来求解，计算机由于其二进制存储的特点可以很巧妙的解决一些问题，像之前的 Single Number 和 Single Number II 都是很巧妙利用位操作来求解。此题由于构成输入字符串的字符只有四种，分别是 A, C, G, T，下面来看下它们的 ASCII 码用二进制来表示：

A: 0100 0 **001**　C: 0100 0 **011**　G: 0100 0 **111**　T: 0101 0 **100**

由于目的是利用位来区分字符，当然是越少位越好，通过观察发现，每个字符的后三位都不相同，故而可以用末尾三位来区分这四个字符。而题目要求是 10 个字符长度的串，每个字符用三位来区分，10 个字符需要30位，在 32 位机上也 OK。为了提取出后 30 位，还需要用个 mask，取值为 0x7ffffff，用此 mask 可取出后27位，再向左平移三位即可。算法的思想是，当取出第十个字符时，将其存在 HashMap 里，和该字符串出现频率映射，之后每向左移三位替换一个字符，查找新字符串在 HashMap 里出现次数，如果之前刚好出现过一次，则将当前字符串存入返回值的数组并将其出现次数加一，如果从未出现过，则将其映射到1。为了能更清楚的阐述整个过程，就用题目中给的例子来分析整个过程：

首先取出前九个字符 AAAAACCCC，根据上面的分析，用三位来表示一个字符，所以这九个字符可以用二进制表示为 001001001001001011011011011，然后继续遍历字符串，下一个进来的是C，则当前字符为 AAAAACCCCC，二进制表示为 001001001001001011011011011011，然后将其存入 HashMap 中，用二进制的好处是可以用一个 int 变量来表示任意十个字符序列，比起直接存入字符串大大的节省了内存空间，然后再读入下一个字符C，则此时字符串为 AAAACCCCCA，还是存入其二进制的表示形式，以此类推，当某个序列之前已经出现过了，将其存入结果 res 中即可，参见代码如下：

解法一：

```cpp
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        vector<string> res;
        if (s.size() <= 10) return res;
        int mask = 0x7ffffff, cur = 0;
        unordered_map<int, int> m;
        for (int i = 0; i < 9; ++i) {
            cur = (cur << 3) | (s[i] & 7);
        }
        for (int i = 9; i < s.size(); ++i) {
            cur = ((cur & mask) << 3) | (s[i] & 7);
            if (m.count(cur)) {
                if (m[cur] == 1) res.push_back(s.substr(i - 9, 10));
                ++m[cur];
            } else {
                m[cur] = 1;
            }
        }
        return res;
    }
};
```

上面的方法可以写的更简洁一些，这里可以用 HashSet 来代替 HashMap，只要当前的数已经在 HashSet 中存在了，就将其加入 res 中，这里 res 也定义成 HashSet，这样就可以利用 HashSet 的不能有重复项的特点，从而得到正确的答案，最后将 HashSet 转为 vector 即可，参见代码如下：

解法二：

```cpp
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        unordered_set<string> res;
        unordered_set<int> st;
        int cur = 0;
        for (int i = 0; i < 9; ++i) cur = cur << 3 | (s[i] & 7);
        for (int i = 9; i < s.size(); ++i) {
            cur = ((cur & 0x7ffffff) << 3) | (s[i] & 7);
            if (st.count(cur)) res.insert(s.substr(i - 9, 10));
            else st.insert(cur);
        }
        return vector<string>(res.begin(), res.end());
    }
};
```

上面的方法都是用三位来表示一个字符，这里可以用两位来表示一个字符，00 表示A，01 表示C，10 表示G，11 表示T，那么总共需要 20 位就可以表示十个字符流，其余的思路跟上面的方法完全相同，注意这里的 mask 只需要表示 18 位，所以变成了 0x3ffff，参见代码如下：

解法三：

```cpp
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        unordered_set<string> res;
        unordered_set<int> st;
        unordered_map<int, int> m{{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};
        int cur = 0;
        for (int i = 0; i < 9; ++i) cur = cur << 2 | m[s[i]];
        for (int i = 9; i < s.size(); ++i) {
            cur = ((cur & 0x3ffff) << 2) | (m[s[i]]);
            if (st.count(cur)) res.insert(s.substr(i - 9, 10));
            else st.insert(cur);
        }
        return vector<string>(res.begin(), res.end());
    }
};
```

如果不需要考虑节省内存空间，那可以直接将 10个 字符组成字符串存入 HashSet 中，那么也就不需要 mask 啥的了，但是思路还是跟上面的方法相同：

解法四：

```
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        unordered_set<string> res, st;
        for (int i = 0; i + 9 < s.size(); ++i) {
            string t = s.substr(i, 10);
            if (st.count(t)) res.insert(t);
            else st.insert(t);
        }
        return vector<string>{res.begin(), res.end()};
    }
};
```

188. Best Time to Buy and Sell Stock IV

Say you have an array for which the *i* th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note:
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

这道题实际上是之前那道 Best Time to Buy and Sell Stock III 买股票的最佳时间之三的一般情况的推广，还是需要用动态规划Dynamic programming来解决，具体思路如下：

这里我们需要两个递推公式来分别更新两个变量local和global，参见网友Code Ganker的博客，我们其实可以求至少k次交易的最大利润。我们定义local[i][j]为在到达第i天时最多可进行j次交易并且最后一次交易在最后一天卖出的最大利润，此为局部最优。然后我们定义global[i][j]为在到达第i天时最多可进行j次交易的最大利润，此为全局最优。它们的递推式为：

local[i][j] = max(global[i - 1][j - 1] + max(diff, 0), local[i - 1][j] + diff)

global[i][j] = max(local[i][j], global[i - 1][j]),

其中局部最优值是比较前一天并少交易一次的全局最优加上大于0的差值，和前一天的局部最优加上差值后相比，两者之中取较大值，而全局最优比较局部最优和前一天的全局最优。

但这道题还有个坑，就是如果k的值远大于prices的天数，比如k是好几百万，而prices的天数就为若干天的话，上面的DP解法就非常的没有效率，应该直接用Best Time to Buy and Sell Stock II 买股票的最佳时间之二的方法来求解，所以实际上这道题是之前的二和三的综合体，代码如下：

```
class Solution {
public:
    int maxProfit(int k, vector<int> &prices) {
        if (prices.empty()) return 0;
        if (k >= prices.size()) return solveMaxProfit(prices);
        int g[k + 1] = {0};
        int l[k + 1] = {0};
        for (int i = 0; i < prices.size() - 1; ++i) {
            int diff = prices[i + 1] - prices[i];
            for (int j = k; j >= 1; --j) {
                l[j] = max(g[j - 1] + max(diff, 0), l[j] + diff);
                g[j] = max(g[j], l[j]);
            }
        }
        return g[k];
    }
    int solveMaxProfit(vector<int> &prices) {
        int res = 0;
        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] - prices[i - 1] > 0) {
                res += prices[i] - prices[i - 1];
            }
        }
        return res;
    }
};
```

192. Word Frequency

Write a bash script to calculate the frequency of each word in a text file `words.txt`.

For simplicity sake, you may assume:

- `words.txt` contains only lowercase characters and space `' '` characters.
- Each word must consist of lowercase characters only.
- Words are separated by one or more whitespace characters.

For example, assume that `words.txt` has the following content:

```
the day is sunny the the
the sunny is is
```

Your script should output the following, sorted by descending frequency:

```
the 4
is 3
sunny 2
day 1
```

Note:
Don't worry about handling ties, it is guaranteed that each word's frequency count is unique.

[show hint]

Hint:
Could you write it in one-line using Unix pipes?

这道题给了我们一个文本文件，让我们统计里面单词出现的个数，提示中让我们用管道Pipes来做，在之前那道Tenth Line中，我们使用过管道命令，管道命令的讲解请参见这个帖子。提示中让我们用管道连接各种命令，然后一行搞定，那么我们先来看第一种解法，乍一看啥都不明白，咋办？没关系，容我慢慢来讲解。首先用的关键字是grep命令，该命令一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来，详解请参见这个帖子。后面紧跟的-oE '[a-z]+'参数表示原文本内容变成一个单词一行的存储方式，于是此时文本的内容就变成了：

```
the
day
is
sunny
the
the
the
sunny
is
```

下面的sort命令就是用来排序的，参见这个帖子。排完序的结果为：

```
day
is
is
is
sunny
sunny
the
the
the
the
```

后面的uniq命令是表示去除重复行命令(参见这个帖子)，后面的参数-c表示在每行前加上表示相应行目出现次数的前缀编号，得到结果如下：

```
 1 day
 3 is
 2 sunny
 4 the
```

然后我们再sort一下，后面的参数-nr表示按数值进行降序排列，得到结果：

```
 4 the
 3 is
 2 sunny
 1 day
```

而最后的awk命令就是将结果输出，两列颠倒位置即可：

解法一：

```
grep -oE '[a-z]+' words.txt | sort | uniq -c | sort -nr | awk '{print $2" "$1}'
```

下面这种方法用的关键字是tr命令，该命令主要用来进行字符替换或者大小写替换，详解请参见这个帖子。后面紧跟的-s参数表示如果发现连续的字符，就把它们缩减为1个，而后面的' '和'\n'就是空格和回车，意思是把所有的空格都换成回车，那么第一段命令tr -s ' ' '\n' < words.txt 就好理解了，将单词之间的空格都换成回车，跟上面的第一段实现的作用相同，后面就完全一样了，参见上面的讲解：

解法二：

```
tr -s ' ' '\n' < words.txt | sort | uniq -c | sort -nr | awk '{print $2, $1}'
```

下面这种方法就没有用管道命令进行一行写法了，而是使用了强大的文本分析工具awk进行写类C的代码来实现，这种写法在之前的那道Transpose File已经讲解过了，这里就不多说了，最后要注意的是sort命令的参数-nr -k 2表示按第二列的降序数值排列：

解法三：

```
awk '{
    for (i = 1; i <= NF; ++i) ++s[$i];
} END {
    for (i in s) print i, s[i];
}' words.txt | sort -nr -k 2
```

194. Transpose File

Given a text file `file.txt`, transpose its content.

You may assume that each row has the same number of columns and each field is separated by the `' '` character.

For example, if `file.txt` has the following content:

```
name age
alice 21
ryan 30
```

Output the following:

```
name alice ryan
age 21 30
```

这道题让我们转置一个文件，其实感觉就是把文本内容当做了一个矩阵，每个单词空格隔开看做是矩阵中的一个元素，然后将转置后的内容打印出来。那么我们先来看使用awk关键字的做法，关于awk的介绍可以参见这个帖子。其中NF表示当前记录中的字段个数，就是有多少列，NR表示已经读出的记录数，就是行号，从1开始。那么在这里NF是2，因为文本只有两列，这里面这个for循环还跟我们通常所熟悉for循环不太一样，通常我们以为i只能是1和2，然后循环就结束了，而这里的i实际上遍历的数字为1,2,1,2,1,2，我们可能看到实际上循环了3遍1和2，而行数正好是3，可能人家就是这个机制吧。知道了上面这些，那么下面的代码就不难理解了，遍历过程如下：

i = 1, s = [name]

i = 2, s = [name; age]

i = 1, s = [name alice; age]

i = 2, s = [name alice; age 21]

i = 1, s = [name alice ryan; age 21]

i = 2, s = [name alice ryan; age 21 30]

然后我们再将s中的各行打印出来即可，参见代码如下：

解法一：

```
awk '{
    for (i = 1; i <= NF; ++i) {
        if (NR == 1) s[i] = $i;
        else s[i] = s[i] " " $i;
    }
} END {
    for (i = 1; s[i] != ""; ++i) {
        print s[i];
    }
}' file.txt
```

下面这种方法和上面的思路完全一样，但是代码风格不一样，上面是C语言风格，而这个完全就是Bash脚本的风格了，我们用read关键字，我们可以查看read的用法read: usage: read [-ers] [-u fd] [-t timeout] [-p prompt] [-a array] [-n nchars] [-d delim] [name ...]。那么我们知道-a表示数组，将读出的每行内容存入数组line中，那么下一行for中的一堆特殊字符肯定让你头晕眼花，关于shell中的特殊变量可以参见这个帖子，其实我也不能算特别理解下面的代码，大概觉得跟上面的思路一样，求大神来具体给讲解下哈：

解法二：

```
while read -a line; do
    for ((i = 0; i < "${#line[@]}"; ++i)); do
        a[$i]="${a[$i]} ${line[$i]}"
    done
done < file.txt
for ((i = 0; i < ${#a[@]}; ++i)); do
    echo ${a[i]}
done
```

199. Binary Tree Right Side View

Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.

For example:
Given the following binary tree,

```
   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

You should return `[1, 3, 4]`.

Credits:
Special thanks to @amrsaqr for adding this problem and creating all test cases.

这道题要求我们打印出二叉树每一行最右边的一个数字，实际上是求二叉树层序遍历的一种变形，我们只需要保存每一层最右边的数字即可，可以参考我之前的博客 Binary Tree Level Order Traversal 二叉树层序遍历，这道题只要在之前那道题上稍加修改即可得到结果，还是需要用到数据结构队列queue，遍历每层的节点时，把下一层的节点都存入到queue中，每当开始新一层节点的遍历之前，先把新一层最后一个节点值存到结果中，代码如下：

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> rightSideView(TreeNode *root) {
        vector<int> res;
        if (!root) return res;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            res.push_back(q.back()->val);
            int size = q.size();
            for (int i = 0; i < size; ++i) {
                TreeNode *node = q.front();
                q.pop();
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }
        return res;
    }
};
```

200. Number of Islands

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input:
11110
11010
11000
00000

Output: 1
```

Example 2:

```
Input:
11000
11000
00100
00011

Output: 3
```

Credits:
Special thanks to @mithmatt for adding this problem and creating all test cases.

这道求岛屿数量的题的本质是求矩阵中连续区域的个数，很容易想到需要用深度优先搜索 DFS 来解，我们需要建立一个 visited 数组用来记录某个位置是否被访问过，对于一个为 ‘1’ 且未被访问过的位置，递归进入其上下左右位置上为 ‘1’ 的数，将其 visited 对应值赋为 true，继续进入其所有相连的邻位置，这样可以将这个连通区域所有的数找出来，并将其对应的 visited 中的值赋 true，找完相邻区域后，将结果 res 自增1，然后再继续找下一个为 ‘1’ 且未被访问过的位置，以此类推直至遍历完整个原数组即可得到最终结果，代码如下：

解法一：

```cpp
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        int m = grid.size(), n = grid[0].size(), res = 0;
        vector<vector<bool>> visited(m, vector<bool>(n));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '0' || visited[i][j]) continue;
                helper(grid, visited, i, j);
                ++res;
            }
        }
        return res;
    }
    void helper(vector<vector<char>>& grid, vector<vector<bool>>& visited, int x, int y) {
        if (x < 0 || x >= grid.size() || y < 0 || y >= grid[0].size() || grid[x][y] == '0' || visited[x][y]) return;
        visited[x][y] = true;
        helper(grid, visited, x - 1, y);
        helper(grid, visited, x + 1, y);
        helper(grid, visited, x, y - 1);
        helper(grid, visited, x, y + 1);
    }
};
```

当然，这种类似迷宫遍历的题目 DFS 和 BFS 两对好基友肯定是形影不离的，那么 BFS 搞起。其实也很简单，就是在遍历到 ‘1’ 的时候，且该位置没有被访问过，那么就调用一个 BFS 即可，借助队列 queue 来实现，现将当前位置加入队列，然后进行 while 循环，将队首元素提取出来，并遍历其周围四个位置，若没有越界的话，就将 visited 中该邻居位置标记为 true，并将其加入队列中等待下次遍历即可，参见代码如下：

解法二：

```cpp
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        int m = grid.size(), n = grid[0].size(), res = 0;
        vector<vector<bool>> visited(m, vector<bool>(n));
        vector<int> dirX{-1, 0, 1, 0}, dirY{0, 1, 0, -1};
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == '0' || visited[i][j]) continue;
                ++res;
                queue<int> q{{i * n + j}};
                while (!q.empty()) {
                    int t = q.front(); q.pop();
                    for (int k = 0; k < 4; ++k) {
                        int x = t / n + dirX[k], y = t % n + dirY[k];
                        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] == '0' || visited[x][y]) continue;
                        visited[x][y] = true;
                        q.push(x * n + y);
                    }
                }
            }
        }
        return res;
    }
};
```