Solution to leetcode

Boya Wu

目录

第 1 章　Array

1.1　Background

对数组的考察多是结合hash（加速查找过程），排序（有序查找)，ＤＰ，ＢＡＣＫＴＲＡＣＫＩＮＧ等方式。在面对纯数组技巧的题目时，可以考虑双指针和计算过程的化简方式。

1.2 Solutions

1.2.1 2sum

Question:

Given an array of integers, 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.

Example

Input: numbers=2, 7, 11, 15, target=9
Output: index1=1, index2=2

Anslysis

如果先排序后查找的话，时间复杂度为O(nlgn + n),空间复杂度为O(1)
因为array不是已排序的，且要确定真实的索引，所以不能使用双指针的方法
可以使用hash来存储每个数值所需要的补值，再通过O(n)的时间来寻找
注意使用hash来寻找时，不要处理自己

Solution

1. Hash : $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```cpp
class Solution {
    public:
        vector<int> twoSum(vector<int>& nums, int target) {
            vector<int> res;
            unordered_map<int,int> hashmap;

            for(int i=0;i<nums.size();++i){
                auto iter = hashmap.find(nums[i]);

                if(iter == hashmap.end()){
                    hashmap.insert(pair<int,int>(target - nums[i], i+1));
                }else{
                    if(i+1 < iter->second){
                        res.push_back(i+1); res.push_back(iter->second);
                    }else{
                        res.push_back(iter->second);    res.push_back(i+1);
                    }
                }
            }

            return res;
        }
};
```

1.2.2 3sum-closest

Question:

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example

Given array S = -1 2 1 -4, and target = 1.
The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Anslysis

类似于3-sum

Solution

: 时间复杂度O(n^2) , 空间复杂度O(1)

```cpp
int threeSumClosest(vector<int> &num, int target) {
    vector<int> v(num.begin(), num.end()); // I didn't wanted to disturb original array.
    int n = 0;
    int ans = 0;
    int sum;

    sort(v.begin(), v.end());

    // If less then 3 elements then return their sum
    while (v.size() <= 3) {
        return accumulate(v.begin(), v.end(), 0);
    }

    n = v.size();

    /* v[0] v[1] v[2] ... v[i] .... v[j] ... v[k] ... v[n-2] v[n-1]
     *                    v[i]  <=  v[j]  <= v[k] always, because we sorted our array.
     * Now, for each number, v[i] : we look for pairs v[j] & v[k] such that
     * absolute value of (target - (v[i] + v[j] + v[k]) is minimised.
     * if the sum of the triplet is greater then the target it implies
     * we need to reduce our sum, so we do K = K - 1, that is we reduce
     * our sum by taking a smaller number.
     * Simillarly if sum of the triplet is less then the target then we
     * increase out sum by taking a larger number, i.e. J = J + 1.
     */
    ans = v[0] + v[1] + v[2];
    for (int i = 0; i < n-2; i++) {
        int j = i + 1;
        int k = n - 1;
        while (j < k) {
            sum = v[i] + v[j] + v[k];
            if (abs(target - ans) > abs(target - sum)) {
                ans = sum;
                if (ans == target) return ans;
            }
            (sum > target) ? k-- : j++;
        }
    }
    return ans;
}
```

1.2.3 3sum

Question:

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Anslysis

我们总是需要先排序，这样利于去除重复和按序输出

思路 1，记录i+j的位置与和，耗费O(n^2),对于每个i寻找-1*nums[i],可以得到j,k.问题在于如何去除j,k的重复.

思路 2，可以在寻找j,k是,选择移动j或k.类似 2 分，但是是顺序移动，O(n)

Solution

: $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    vector<vector<int> >res;
    public:
        vector<vector<int> > threeSum(vector<int>& nums) {
            len = nums.size();

            sort(nums.begin(),nums.end());

            for(int i=0;i<len;++i) {
                if(i==0 || nums[i] != nums[i-1]) {
                    int left = i+1,right = len-1;
                    while(left < right) {
                        if(nums[i] + nums[left] + nums[right] < 0){
                            ++left;
                        }else if(nums[i] + nums[left] + nums[right] > 0)
                            {
                            --right;
                        }else{
                            vector<int> tmp;
                            tmp.push_back(nums[i]); tmp.push_back(nums[
                                left]);  tmp.push_back(nums[right]);
                            res.push_back(tmp);
                            while(left < right && nums[left] == nums[
                                left+1])   ++left;
                            while(left < right && nums[right] == nums[
                                right-1]) --right;
                            ++left; --right;
                        }
                    }
                }
            }
            return res;
        }
};
```

1.2.4 4sum

Question:

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.

Anslysis

类似3sum，注意仍然需要去除重复．在第二维的操作是基于i+1，而非 0 的

Solution

: $\boxed{\text{时间复杂度O}(n^3)\text{，空间复杂度O}(n^3)}$

```
class Solution {
    int len;
    public:
        vector<vector<int> > fourSum(vector<int>& nums, int target) {
            len = nums.size();
            sort(nums.begin(),nums.end());
            vector<vector<int> > res;

            for(int i=0;i<len-3;++i){
                if(i>0 && nums[i] == nums[i-1])
                    continue;

                for(int j=i+1;j<len-2;++j){
                    if(j>i+1 && nums[j] == nums[j-1])
                        continue;
                    int left = j+1,right = len-1;
                    while(left < right) {
                        if(nums[i] + nums[j] + nums[left] + nums[right]
                            == target){
                            vector<int> tmp;
                            tmp.push_back(nums[i]);tmp.push_back(nums[j
                                ]);tmp.push_back(nums[left]);tmp.
                                push_back(nums[right]);
                            res.push_back(tmp);
                            ++left; --right;
                            while(left < right && nums[left] == nums[
                                left-1]) ++left;
                            while(left < right && nums[right] == nums[
                                right+1]) --right;
                        }else if(nums[i] + nums[j] + nums[left] + nums[
                            right] > target){
                            --right;
                        }else{
                            ++left;
                        }
                    }
                }
            }
            return res;
        }
};
```

1.2.5 Combination-sum

Question:

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
The same repeated number may be chosen from C unlimited number of times.
Note:
All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 $\leqslant$ a2 $\leqslant$ … $\leqslant$ ak).
The solution set must not contain duplicate combinations.

Example

Given candidate set 2,3,6,7 and target 7,
A solution set is:
7
2, 2, 3

Anslysis

因为结果需要升序，所以我们先对数组排序，耗时O(nlgn)
因为每个元素可以多次使用，所以需要回溯的筛选

Solution

1. Hash : $\boxed{\text{时间复杂度O}(n^2)\text{ , 空间复杂度O(n)}}$

```
lass Solution {
public:
    std::vector<std::vector<int> > combinationSum(std::vector<int> &
        candidates, int target) {
        std::sort(candidates.begin(), candidates.end());
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum(candidates, target, res, combination, 0);
        return res;
    }
private:
    void combinationSum(std::vector<int> &candidates, int target, std::
        vector<std::vector<int> > &res, std::vector<int> &combination,
        int begin) {
        if (!target) {
            res.push_back(combination);
            return;
        }
        for (int i = begin; i != candidates.size() && target >=
            candidates[i]; ++i) {
            combination.push_back(candidates[i]);
            combinationSum(candidates, target - candidates[i], res,
                combination, i);
            combination.pop_back();
        }
    }
};
```

1.2.6 Combination-sumII

Question:

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
Each number in C may only be used once in the combination.
Note:
All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ⩽ a2 ⩽ … ⩽ ak).
The solution set must not contain duplicate combinations.

Example

For example, given candidate set 10,1,2,7,6,1,5 and target 8.
A solution set is:
1, 7
1, 2, 5
2, 6
1, 1, 6

Anslysis

与combination sum类似，但是某个元素被选过后就不能被再次使用

Solution

: 时间复杂度O(n^2) , 空间复杂度O(n)

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum2(std::vector<int> &
        candidates, int target) {
        std::sort(candidates.begin(), candidates.end());
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum2(candidates, target, res, combination, 0);
        return res;
    }
private:
    void combinationSum2(std::vector<int> &candidates, int target, std::
        vector<std::vector<int> > &res, std::vector<int> &combination,
        int begin) {
        if  (!target) {
            res.push_back(combination);
            return;
        }
        for (int i = begin; i != candidates.size() && target >=
            candidates[i]; ++i)
            if (i == begin || candidates[i] != candidates[i - 1]) {
                combination.push_back(candidates[i]);
                combinationSum2(candidates, target - candidates[i], res,
                    combination, i + 1);
                combination.pop_back();
            }
    }
};
```

1.2.7 Combination-sumIII

Question:

Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers. Ensure that numbers within the set are sorted in ascending order.

Example

1.Input: k = 3, n = 7
Output: [[1,2,4]]

2.Input: k = 3, n = 9
Output:[[1,2,6], [1,3,5], [2,3,4]]

Anslysis

回溯，控制待选元素的数目

Solution

: 时间复杂度O(n^k)，空间复杂度O(n)

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum3(int k, int n) {
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum3(n, res, combination, 1, k);
        return res;
    }
private:
    void combinationSum3(int target, std::vector<std::vector<int> > &res
        , std::vector<int> &combination, int begin, int need) {
        if (!target) {
            res.push_back(combination);
            return;
        }
        else if (!need)
            return;
        for (int i = begin; i != 10 && target >= i * need + need * (need
            - 1) / 2; ++i) {
            combination.push_back(i);
            combinationSum3(target - i, res, combination, i + 1, need -
                1);
            combination.pop_back();
        }
    }
};
```

1.2.8 Construct-binary-tree-from-preorder-and-inorder-traversal

Question:

Given preorder and inorder traversal of a tree, construct the binary tree.
Note:
You may assume that duplicates do not exist in the tree.

Anslysis

需要确定是否有重复！

Solution

Recurssion : $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```
TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {
    return create(preorder, inorder, 0, preorder.size() - 1, 0, inorder.
       size() - 1);
}

TreeNode* create(vector<int>& preorder, vector<int>& inorder, int ps,
    int pe, int is, int ie){
    if(ps > pe){
        return nullptr;
    }
    TreeNode* node = new TreeNode(preorder[ps]);
    int pos;
    for(int i = is; i <= ie; i++){
        if(inorder[i] == node->val){
            pos = i;
            break;
        }
    }
    node->left = create(preorder, inorder, ps + 1, ps + pos - is, is,
       pos - 1);
    node->right = create(preorder, inorder, pe - ie + pos + 1, pe, pos +
        1, ie);
    return node;
}
```

Recurssion : $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```
class Solution {
public:
    TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {

        if(preorder.size()==0)
            return NULL;

        stack<int> s;
        stack<TreeNode *> st;
        TreeNode *t,*r,*root;
        int i,j,f;

        f=i=j=0;
        s.push(preorder[i]);

        root = new TreeNode(preorder[i]);
        st.push(root);
        t = root;
        i++;

        while(i<preorder.size())
        {
            if(!st.empty() && st.top()->val==inorder[j])
            {
                t = st.top();
                st.pop();
                s.pop();
                f = 1;
                j++;
            }
            else
            {
                if(f==0)
                {
                    s.push(preorder[i]);
                    t -> left = new TreeNode(preorder[i]);
                    t = t -> left;
                    st.push(t);
                    i++;
                }
                else
                {
                    f = 0;
                    s.push(preorder[i]);
                    t -> right = new TreeNode(preorder[i]);
                    t = t -> right;
                    st.push(t);
                    i++;
                }
            }
        }

        return root;
    }
};
```

1.2.9 Contain-duplicate

Question:

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

Anslysis

排序，时间复杂度O(nlgn + n),空间复杂度O(1)
哈希,时间复杂度O(n),空间复杂度O(n).

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```cpp
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        return nums.size() > set<int>(nums.begin(), nums.end()).size();
    }
};
```

1.2.10 Contain-duplicateII

Question:

Given an array of integers and an integer k, find out whether there there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k.

Anslysis

这里我们需要注意两点：（1）查找存在性问题时，使用hash非常快，
(2)我们不需要记录之前的数值的具体位置，可以将hash中的数据限定在一定范围之内
2.1 每次只记录最后出现的位置；
2.2 只需要记录K个元素；
2.2比2.1更能节省空间

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O(1)}}$

```cpp
class Solution {
    int len;
    public:
        bool containsNearbyDuplicate(vector<int>& nums, int k) {
            unordered_set<int> us;
            len = nums.size();
            if(len ==0 || k < 0)    return false;

            for(int idx=0;idx<len;++idx) {
                /*
                 * it's smart that we only matain k elems in set
                 */
                if( idx > k )
                    us.erase(nums[idx - k + 1]);

                if(us.find(nums[idx]) != us.end())
                    return true;
                us.insert(nums[idx]);
            }
            return false;
        }
};
```

1.2.11 Container-with-most-water

Question:

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Anslysis

需要证明每次移动较小的挡板能得到较大的结果.
假设有挡板a,b,height[a] ¡ height[b],我们现在会移动a,假设最大的结果是移动b到b-1得来的[a,b-1]，可是这样的话b能和a构成更大的矩形，矛盾

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        int maxArea(vector<int>& height) {
            len = height.size();
            int left=0,right = len -1;
            int result = 0;
            while(left < right) {
                result = max(result, (right - left) * min(height[left],
                    height[right]));
                if(height[left] < height[right]) {
                    ++left;
                }else{
                    --right;
                }
            }
            return result;
        }
};
```

1.2.12 Find-mininum-in-rotated-sorted-array

Question:

Suppose a sorted array 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.

Anslysis

因为是sorted的，所以借助于二分
当left ¡ right时，left即为所求
当left ¿ right时，如果mid ¿ left,则寻找[mid,righti]；如果mid ¡ right,则寻找[left,mid]

Solution

: $\boxed{\text{时间复杂度O(lgn)，空间复杂度O(n)}}$

```
int findMin(vector<int> &num) {
        int start=0,end=num.size()-1;

        while (start<end) {
            if (num[start]<num[end])
                return num[start];

            int mid = (start+end)/2;

            if (num[mid]>=num[start]) {
                start = mid+1;
            } else {
                end = mid;
            }
        }

        return num[start];
    }
```

1.2.13 Find-mininum-in-rotated-sorted-arrayII

Question:

Follow up for "Find Minimum in Rotated Sorted Array":
What if duplicates are allowed?
Would this affect the run-time complexity? How and why?

Anslysis

因为sorted，所以仍然借助于二分
当left ¡ right, 则left为所求
当left ¿= right，若mid ¿ left —— mid==left!=right,则寻找(mid,right]；若mid ¡ right —— mid==right!=left,则寻找[left,mid]；若mid==left==right,则left或right前进一步

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```
class Solution {
public:
    int findMin(vector<int> &num) {
        int lo = 0;
        int hi = num.size() - 1;
        int mid = 0;

        while(lo < hi) {
            mid = lo + (hi - lo) / 2;

            if (num[mid] > num[hi]) {
                lo = mid + 1;
            }
            else if (num[mid] < num[hi]) {
                hi = mid;
            }
            else { // when num[mid] and num[hi] are same
                hi--;
            }
        }
        return num[lo];
    }
};
```

1.2.14 Find-peak-elements

Question:

A peak element is an element that is greater than its neighbors.
Given an input array where num[i] ≠ num[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 num[-1] = num[n] = -∞.

Example

In array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

Anslysis

如果使用O(n)的算法，则可以从左向右遍历求解
我们来考虑在找到某个点时出现的情况：1．left ¡ curr && curr ¿ right,则right为所求;
2．left ¡ curr && curr ¡ right，则必定存在一个peak位于curr的右侧;
3．left ¿ curr && curr ¿ right, 则必定存在一个peak位于curr的左侧;
4．left ¿ curr && right && curr，则curr的左侧和右侧必定分别存在一个peak;
我们可以参照上面的分析，使用二分的方法来合理的移动left,right指针，从而定位到心仪的数据。这样就只需要O(lgn)的时间复杂度。

Solution

: 时间复杂度O(lgn)，空间复杂度O(1)

```
class Solution {
public:
    int findPeakElement(const vector<int> &num)
    {
        int low = 0;
        int high = num.size()-1;

        while(low < high)
        {
            int mid1 = (low+high)/2;
            int mid2 = mid1+1;
            if(num[mid1] < num[mid2])
                low = mid2;
            else
                high = mid1;
        }
        return low;
    }
};
```

1.2.15 First-missing-positive

Question:

Given an unsorted integer array, find the first missing positive integer.

Anslysis

可以 " 递归 " 的安排每个数字

在寻找nums[idx]合适的位置的过程中，可以将目标位置(nums[nums[idx] - 1])中的数值与当前位置(idx)做交换，这样可以持续重复处理nums[idx]与nums[nums[idx] - 1].

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        int firstMissingPositive(vector<int>& nums) {
            len = nums.size();

            for(int idx=0;idx<len;++idx) {
                if(nums[idx] != idx+1 && nums[idx] > 0 && nums[idx] <=
                    len) { //this can be placed
                    int toBeArranged = nums[idx];//to be placed at
                        toBeArranged-1
                    nums[idx] = -1;//this position was abandened
                    while(toBeArranged > 0 && toBeArranged <= len &&
                        toBeArranged != nums[toBeArranged-1]) {
                        int oldVal = nums[toBeArranged-1];
                        nums[toBeArranged-1] = toBeArranged;
                        toBeArranged = oldVal;
                    }
                }
            }

            for(int idx=0;idx<len;++idx){
                if(nums[idx] != idx+1)
                    return idx+1;
            }
            return len;
        }
};
```

1.2.16 Insert-interval

Question:

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Anslysis

灵感来自于array的另一到题目，我们可以在newInterval被完整接纳之前一致维护着它，直到非交叉的情况，注意使用标志位来避免漏加了newInterval

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O(n)}}$

```cpp
class Solution {
    public:
        bool isOverlapped(Interval l1,Interval l2) {
            return !(l1.start > l2.end || l2.start > l1.end);
        }

        vector<Interval> insert(vector<Interval>& intervals, Interval
            newInterval) {
            vector<Interval> res;

            bool isUsed = false;
            bool hasOverlapped = false;
            for(int i=0;i<intervals.size();++i) {
                if(!isOverlapped(intervals[i],newInterval)) {
                    if(intervals[i].start > newInterval.end && !isUsed)
                        {
                         res.push_back(newInterval);
                         isUsed = true;
                    }
                    res.push_back(intervals[i]);
                }else{
                    newInterval.start = min(newInterval.start, intervals
                        [i].start);
                    newInterval.end = max(newInterval.end, intervals[i].
                        end);
                }
            }

            if(!isUsed)
                res.push_back(newInterval);

            return res;
        }
};
```

1.2.17 Jump-game

Question:

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you are able to reach the last index.

Anslysis

很多时候，对于可达性问题，我们不需要保存具体的跳转流程，只需要知道可达与否的状态

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O(1)}}$

```
class Solution {
    int len;
    public:
        bool canJump(vector<int>& nums) {
            len = nums.size();

            int maxFront = 1;
            for(int idx=0;idx<len;++idx)  {
                if(maxFront < idx + 1)  return false;
                maxFront = max(maxFront, idx+1+nums[idx]);
            }
            return true;
        }
};
```

1.2.18 Jump-gameII

Question:

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Your goal is to reach the last index in the minimum number of jumps.

Example

Given array A = [2,3,1,1,4]
The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

Anslysis

我们从左向右的扫描数组，在某个位置i可以跳到位置i + steps[i]，如果这个位置之前没有被到达，则到达该位置的最少跳数就为i + steps[i].
使用替代法来证明：首先，初始时第一个位置所能达到的位置为0 + steps[0]，所以从[0, 0+steps[0]]的最小跳数为 1；
在跳动的过程中，假设已求得达到某个位置i的最小跳数为minSteps[i]，则从 i 向后跳所能到达的位置为[i+1, i+steps[i]]，如果这个范围内存在j， j 之前还没有到达，则到达j的最小跳数为minSteps[i] + 1。假设到达 j 的跳数小于minSteps[i]+ 1 ,则 j 之前一定已经被到达过了，与假设矛盾。

Solution

: 时间复杂度O(n)，空间复杂度O(n)

```
int jump(int A[], int n) {
    if(n == 0){
        return 0;
    }
    int maxReachPos = A[0];
    int curMaxReachPos = A[0];
    int curStep = 1;
    for(int i = 1; i <= min(n, maxReachPos); i++){
        curMaxReachPos = max(curMaxReachPos, i + A[i]);
        if(i == n - 1){
            return curStep;
        }
        if(i == maxReachPos){
            maxReachPos = curMaxReachPos;
            curStep++;
        }
    }
    return 0;
}
```

1.2.19 Longest-consecutive-sequence

Question:

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.
Your algorithm should run in O(n) complexity.

Example

Given [100, 4, 200, 1, 3, 2]
The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Anslysis

顺序的遍历每个元素。对于每一个元素，我们都需要记录以其为start的序列长度和以其为end的序列长度。
因为序列长度是通过序列的左右端点来确定的。当新加入一个元素时，我们需要查看以存在序列的左右端点会不会因它而延长。

Solution

: 时间复杂度O(n)，空间复杂度O(1)

```
int longestConsecutive(vector<int> num) {
    unordered_map<int,int> m;
    int ret = 0;
    for(auto & n: num){

        //it is in the middle of some consecutive sequence OR we can say
             it is already visited earlier
        //therefore it does not contribute to a longer sequence
        if(m[n]) continue;

        //we cannot find adjacent sequences to n, therefore it is a
            single element sequence by itself
        if(m.find(n-1) == m.end() && m.find(n+1) == m.end()){ //
            ret = max(ret, m[n] = 1);
            continue;
        }

        //found a sequence at n+1
        //you may wonder what if the sequence at n+1 contains element n?
        //It it contains n, when we add the length by 1 using m[n+1]+1,
            it is wrong, right?
        //However it is not possible, because if sequence at n+1
            contains n, m[n] must have been visited earlier
        //we checked that using if(m[n]) continue; here m[n] is not yet
            visited;
        //therefore sequence m[n+1] is always on right side, we can
            safely extend the length by 1
        if(m.find(n-1)==m.end()){

            //we want to maintain the TWO boundaries of the sequence
            //the new length of the sequence is the original length m[n
                +1] incremented by 1
            //left boundary m[n] = m[n+1] +1;
            //right boundary m[n+m[n+1]] = m[n+1]+1;
            //why n+m[n+1]? it is equal to m[n+1]+(n+1)-1
            //meaning the old left boundary n+1 plus the old length m[n
                +1] minus 1
            //e.g. for sequence 3,4,5,6 m[3] = 4, and right boundary 6 =
                 3+m[3]-1 (here n+1 == 3);
            int r = m[n] = m[n+m[n+1]] = m[n+1]+1;
            ret = max(ret, r);
            continue;
        }

        //this is similar to the above case just extend to the right
        if(m.find(n+1)==m.end()){
            int r = m[n] = m[n-m[n-1]] = m[n-1]+1;
            ret = max(ret,r);
            continue;
        }

        //here, we found both sequences at n+1 and n-1, for reasons we
            explained,
        //the sequences have no overlap.
        //Now, we just need to add the length of current element n (
            which is 1) to both left and right boundaries
        //the new length will be :
        //old length of left sequence (m[n-1]) + old length of right
            sequence (m[n+1]) + 1
        //We also need to mark m[n] as visited, here we can either mark
            it with 1 or the new length;
        int r = m[n-m[n-1]] = m[n+m[n+1]] = 1+ m[n+1]+ m[n-1];
        m[n] = 1; //basically we just need to mark m[n] as any non-zero
            number
        // or we can write
        //int r = m[n] = m[n-m[n-1]] = m[n+m[n+1]] = 1+ m[n+1]+ m[n-1];
        ret = max(ret,r);
    }
    return ret;
}
```

1.2.20 Longest-rectangle-in-histogram

Question:

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

Example

Given height = [2,1,5,6,2,3]
return 10

Anslysis

对于某一高度 h ,如果我们在向后搜索的过程中没有找到比它更矮的挡板，则 h 一直在生效。
因此我们需要存储高度 h ,直到出现比它矮的挡板。
可以在最后的挡板后面插入高度为 0 的挡板来清空之前一直未被消耗的挡板。

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```
class Solution {
  public:
      int largestRectangleArea(vector<int> &height) {

          int ret = 0;
          height.push_back(0);
          vector<int> index;

          for(int i = 0; i < height.size(); i++)
          {
              while(index.size() > 0 && height[index.back()] >= height
                  [i])
              {
                  int h = height[index.back()];
                  index.pop_back();

                  int sidx = index.size() > 0 ? index.back() : -1;
                  if(h * (i-sidx-1) > ret)
                      ret = h * (i-sidx-1);
              }
              index.push_back(i);
          }

          return ret;
      }
  };
```

1.2.21 Majority-element

Question:

Given an array of size n, find the majority element. The majority element is the element that appears more than n/2 times.
You may assume that the array is non-empty and the majority element always exist in the array.

Anslysis

1.可以排序后解决
2.hash存储每个元素出现的次数
3.partition得到n/2元素
4.使用单个空间记录当前出现次数最多的元素[Boyer-Moore Majority Vote Algorithm多数投

票算法]

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

```
public class Solution {
    public int majorityElement(int[] num) {

        int major=num[0], count = 1;
        for(int i=1; i<num.length;i++){
            if(count==0){
                count++;
                major=num[i];
            }else if(major==num[i]){
                count++;
            }else count--;

        }
        return major;
    }
}
```

1.2.22 Majority-elementII

Question:

Given an array of size n, find the majority element. The majority element is the element that appears more than n/2 times.
You may assume that the array is non-empty and the majority element always exist in the array.

Anslysis

仍然使用[Boyer-Moore Majority Vote Algorithm多数投票算法]
只不过记录两个潜在对象。每次削减时要同时削减两个对象

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

```
class Solution {
        int  len;
        public:
            vector<int> majorityElement(vector<int>& nums) {
                len = nums.size();
                int cand1,cand2;
                int num1=0,num2=0;

                for(int i = 0;i < len; ++i) {
                    if(nums[i] == cand1 || num1 == 0) {
                        ++num1;
                        cand1 = nums[i];
                    }else if(nums[i] == cand2 || num2 == 0) {
                        ++num2;
                        cand2 = nums[i];
                    }else {
                        --num1;
                        --num2;
                    }
                }

                //需要检查和真实的数目cand1cand2 111223344
                num1 = num2 = 0;
                for(int i=0;i<len;++i) {
                    if(nums[i] == cand1)
                        ++num1;
                    else if(nums[i] == cand2)
                        ++num2;
                }

                vector<int> result;
                if(num1 > len / 3) result.push_back(cand1);
                if(num2 > len / 3) result.push_back(cand2);

                return result;
            }
    };
```

1.2.23 Maximal-rectangle

Question:

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

Anslysis

对每层做递推处理后，可以使用largest-rectangle-in-histogram

Solution

: 时间复杂度O(n)，空间复杂度O(n)

```
struct Info {
    int idx;
    int height;
    Info(int i,int h) : idx(i),height(h) {}
};

class Solution {
    public:
        int getMaxArea(vector<int>& steps) {
            stack<Info> st;
            st.push(Info(0,0));

            int maxArea = 0;
            steps.push_back(0);
            for(int i=0;i<steps.size();++i) {
                if(st.empty() || steps[i] >= st.top().height)
                    st.push(Info(i+1,steps[i]));
                else {
                    while(!st.empty() && steps[i] < st.top().height) {
                        //the last 0 would clear all the front
                        Info tinfo = st.top();
                        st.pop();

                        maxArea = max(maxArea, (i-st.top().idx) * tinfo.
                            height);
                    }
                    st.push(Info(i+1,steps[i]));
                }
            }
            return maxArea;
        }

        int maximalRectangle(vector<vector<char> >& matrix) {
            int cols = matrix.size();
            if(cols == 0)   return 0;
            int rows = matrix[0].size();
            if(rows == 0)   return 0;

            vector<int> steps(rows,0);

            int maxArea = 0;
            for(int i=0;i<cols;++i) {
                for(int j=0;j<rows;++j) {
                    if(matrix[i][j] == '0')
                        steps[j] = 0;
                    else
                        steps[j] += 1;
                }
                maxArea = max(maxArea, getMaxArea(steps));
            }
            return maxArea;
        }
};
```

1.2.24 Median-of-sorted-array

Question:

There are two sorted arrays nums1 and nums2 of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Anslysis

因为序列已经排序，所以可以使用二分的方法来查找
因为我们最终需要的是k=(m+n)/2，在获取了m/2和n/2后，需要分别判断m/2，n/2与k/2的关系，选取较小值
也就是讲选取的两个点前的数目和应该¡=k，否则无法判断所剩余的数组长度能否满足k个的要求

Solution

: 时间复杂度O(n)，空间复杂度O(1)

```
class Solution {
public:
    int getkth(int s[], int m, int l[], int n, int k){
        // let m <= n
        if (m > n)
            return getkth(l, n, s, m, k);
        if (m == 0)
            return l[k - 1];
        if (k == 1)
            return min(s[0], l[0]);

        int i = min(m, k / 2), j = min(n, k / 2);
        if (s[i - 1] > l[j - 1])
            return getkth(s, m, l + j, n - j, k - j);
        else
            return getkth(s + i, m - i, l, n, k - i);
        return 0;
    }

    double findMedianSortedArrays(int A[], int m, int B[], int n) {
        int l = (m + n + 1) >> 1;
        int r = (m + n + 2) >> 1;
        return (getkth(A, m ,B, n, l) + getkth(A, m, B, n, r)) / 2.0;
    }
};
```

1.2.25 Merge-intervals

Question:

Given a collection of intervals, merge all overlapping intervals.

Example

Given [1,3],[2,6],[8,10],[15,18],
return [1,6],[8,10],[15,18].

Anslysis

先按照start值进行排序，然后每次取vector的末尾来查看新interval需不需融入
sort默认按照升序排列,如果需要降序就设置a.start ¿ b.start
如果需要在class中使用比较函数，就要声明为static类型

Solution

: 时间复杂度O(n)，空间复杂度O(n)

```
static bool comp(const Interval& a, const Interval& b){
    return a.start < b.start;
}
vector<Interval> merge(vector<Interval> &intervals) {
    vector<Interval> result;
    if(intervals.empty()){
        return result;
    }
    sort(intervals.begin(), intervals.end(), comp);
    result.push_back(intervals[0]);
    for(int i = 1; i < intervals.size(); i++){
        if(intervals[i].start <= result.back().end){
            Interval temp(result.back().start, max(result.back().end,
                intervals[i].end));
            result.pop_back();
            result.push_back(temp);
        }
        else{
            result.push_back(intervals[i]);
        }
    }
    return result;
}
```

1.2.26 Merge-sorted-array

Question:

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Anslysis

反向拷贝num1中数据到尾部，空出空间来做排序操作
OH, FUCK.我们可以从头部开始的插入，我们可以从尾部开始插入！这样就避免了为空出空间而做的移动操作

Solution

1.From Front : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
        int len1;
public:
        void  copy_to_last(vector<int> & nums1,int m){
                //we need backward-copy
                len1 = nums1.size();
                int from = m - 1,to = len1 - 1;
                while(from >= 0) {
                        nums1[to] = nums1[from];
                        --from; --to;
                }
        }

        void merge(vector<int>& nums1, int m, vector<int>& nums2, int n)
            {
                if(n == 0)  return;

                if(nums1[m-1] <= nums2[0]) {//just paste to end of nums1
                        for(int i=0 ; i<n ; ++i) {
                                nums1[m + i] = nums2[i];
                        }
                        return;
                }

                copy_to_last(nums1, m);
                int mi = len1-m , ni = 0, idx = 0;
                while(mi < len1 && ni < n) {
                        if(nums1[mi] < nums2[ni]) {
                                nums1[idx] = nums1[mi];
                                ++mi;
                        }else{
                                nums1[idx] = nums2[ni];
                                ++ni;
                        }
                        ++idx;
                }

                while(mi < len1){
                    nums1[idx] = nums1[mi];
                    ++idx;  ++mi;
                }
                while(ni < n){
                    nums1[idx] = nums2[ni];
                    ++idx;  ++ni;
                }
        }
};
```

2.From Back : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
public:
    void merge(int A[], int m, int B[], int n) {
        int i=m-1;
        int j=n-1;
        int k = m+n-1;
        while(i >=0 && j>=0)
        {
            if(A[i] > B[j])
                A[k--] = A[i--];
            else
                A[k--] = B[j--];
        }
        while(j>=0)
            A[k--] = B[j--];
    }
};
```

1.2.27 Minimal-size-subarray-sum

Question:

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum $\geqslant$ s. If there isn't one, return 0 instead.

Example

Input: [2,3,1,2,4,3]
Output : s = 7

Anslysis

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

1.2.28 Next-permutation

Question:

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.
If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).
The replacement must be in-place, do not allocate extra memory.

Example

1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1

Anslysis

要依据当前的数值寻找到下一个大小的数值。
从后向前寻找升序的截止点 i ， 以及截止点前的位置 $i - 1$;在后面的升序中查找比val[i-1]大的第一个位置j，交换i-1与j,然后反转[i, end]

可使用的stl为:reverse, upperboud,swap。

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

```
class Solution {
    public:
        void nextPermutation(vector<int> &num)
        {
            if (num.empty()) return;

            // in reverse order, find the first number which is in
               increasing trend (we call it violated number here)
            int i;
            for (i = num.size()-2; i >= 0; --i)
            {
                if (num[i] < num[i+1]) break;
            }

            // reverse all the numbers after violated number
            reverse(begin(num)+i+1, end(num));
            // if violated number not found, because we have reversed
               the whole array, then we are done!
            if (i == -1) return;
            // else binary search find the first number larger than the
               violated number
            auto itr = upper_bound(begin(num)+i+1, end(num), num[i]);
            // swap them, done!
            swap(num[i], *itr);
        }
};
```

1.2.29 Pascal-triangle

Question:

Given numRows, generate the first numRows of Pascal's triangle.

Anslysis

可以从后往前处理来节省所需空间.

Solution

: 时间复杂度O(n) , 空间复杂度O(n)

```
class Solution {
    public:
        vector<vector<int> > generate(int numRows) {
            vector<vector<int> > res;
            if(numRows == 0)    return res;

            vector<int> level;
            level.push_back(1);
            for(int i=1;i<=numRows;++i){
                res.push_back(level);

                for(int idx=level.size()-1 ; idx>0 ; --idx)
                    level[idx] = level[idx] + level[idx-1];
                level.push_back(1);
            }
            return res;
        }
};
```

1.2.30 Plus-one

Question:

Given a non-negative number represented as an array of digits, plus one to the number. The digits are stored such that the most significant digit is at the head of the list.

Anslysis

注意9999的情况，可以1000后push 0

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        vector<int> plusOne(vector<int>& digits) {
            len = digits.size();
            for(int i=len-1;i>=0;--i){
                if(digits[i] == 9){
                    digits[i] = 0;
                }else{
                    ++digits[i];
                    return digits;
                }
            }
            digits[0] = 1;
            digits.push_back(0);//针对的情况，99...9910000
            return digits;
        }
};
```

1.2.31 Product-of-array-except-self

Question:

Given an array of n integers where n ¿ 1, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].
Solve it without division and in O(n).

Example

Input: [1,2,3,4]
Output : [24,12,8,6]

Anslysis

对每个位置来讲，需要其左侧和右侧的乘积。可以分别计算后合并到一起

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```
class Solution {
    int len;
    public:
        vector<int> productExceptSelf(vector<int>& nums) {
            len = nums.size();
            vector<int> result(len,1);

            //先计算左侧的乘积
            result[0] = 1;
            for(int i=1; i<len ;++i)
                result[i] = result[i-1] * nums[i-1];

            //再计算右侧对的乘积
            int rightProduct = 1;
            for(int i=len-1; i>=0; --i) {
                result[i] *= rightProduct;
                rightProduct *= nums[i];
            }

            return result;
        }
};
```

1.2.32 Remove-duplicates-from-sorted-array

Question:

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

Anslysis

双指针

Solution

1. Two pointer : $\boxed{\text{时间复杂度}O(n)\text{，空间复杂度}O(1)}$

```
class Solution {
    int len;
    public:
        int removeDuplicates(vector<int>& nums) {
            len = nums.size();
            int prev = 0;
            int newLen = 0;
            for(int i=0;i<len;++i){
                if(i==0){
                    ++newLen;
                }else if(nums[i] != nums[prev]){
                    nums[++prev] = nums[i];
                    ++newLen;
                }else{
                    ;//do nothing
                }
            }
            return newLen;
        }
};
```

1.2.33 Remove-element

Question:

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Anslysis

数组上使用双指针来删除.

Solution

1. Two Pointer : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        int removeElement(vector<int>& nums, int val) {
            len = nums.size();
            int prev = 0;
            for(int i=0;i<len;++i){
                if(val != nums[i]){
                    nums[prev++] = nums[i];
                }
            }
            return prev;
        }
};
```

1.2.34 Rotate-array

Question:

Rotate an array of n elements to the right by k steps.

Example:

With n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

Anslysis

当然可以申请$O(n)$的空间来处理.
也可以通过三次旋转来实现$O(1)$空间的处理.
也可以交换first k和last k,然后反转last n-k.这个方法相比与上面，旋转次数更少！

Solution

1. Rotate : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        void rangeRotate(vector<int>& nums,int from,int to){
            int tmp;
            while(from < to){
                tmp = nums[from];
                nums[from] = nums[to];
                nums[to] = tmp;

                ++from; --to;
            }
        }

        void rotate(vector<int>& nums, int k) {
            len = nums.size();
            k = k % len;
            if(k == 0)  return ;

            //1)    Use auxilary array,O(n), O(k) ?
            //2)    Two rotate !

            rangeRotate(nums,0,len-1);
            rangeRotate(nums,0,k-1);
            rangeRotate(nums,k,len-1);
        }
};
```

1.2.35 Rotate-image

Question:

You are given an n x n 2D matrix representing an image.
Rotate the image by 90 degrees (clockwise).

Anslysis

类似题目. 重点依然是确定四个顶点.

Solution

: $\boxed{\text{时间复杂度O}(m*n)\text{ , 空间复杂度O}(m+n)}$

```
class Solution {
    int len;
    public:
        void rotate(vector<vector<int> >& matrix) {
            len = matrix.size();

            int tmp1,tmp2;
            for(int col=0;col<=(len-1)/2;++col){
                if(len-1-col >= col){
                    for(int i=col;i<len-1-col;++i){
                        tmp1 = matrix[i][len-1-col];
                        matrix[i][len-1-col] = matrix[col][i];

                        tmp2 = matrix[len-1-col][len-1-i];
                        matrix[len-1-col][len-1-i] = tmp1;

                        tmp1  = matrix[len-1-i][col];
                        matrix[len-1-i][col] = tmp2;

                        matrix[col][i] = tmp1;
                    }
                }
            }
        }
};
```

1.2.36 Search-2D-matrix

Question:

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:
Integers in each row are sorted from left to right.
The first integer of each row is greater than the last integer of the previous row.

Example

1, 3, 5, 7
10, 11, 16, 20
23, 30, 34, 50
and target = 3

return true

Anslysis

因为是已排序的对象，所以参考二分的方法
1）首先比较最右侧（最大值们），来确定target所在的行号；然后在那一行进行二分查找
2）可以将二位矩阵看做一维数组。但是会有较高cache miss率,而且总长度(m*n）可能会对int溢出

Solution

: 时间复杂度O(lg(m*n))，空间复杂度O(1)

```
class Solution {
    public:
        bool searchMatrix(vector<vector<int> >& matrix, int target) {
            int col = matrix.size();
            if(col==0)return false;
            int row = matrix[0].size();

            int top=0,bottom=col-1;
            while(top < bottom){
                int mid = top + (bottom-top)/2;
                if(matrix[mid][row-1]==target)
                    return true;
                else if(matrix[mid][row-1] > target)
                    bottom = mid;
                else
                    top=mid+1;
            }
            int level = top;
            top=0,bottom=row-1;
            while(top <= bottom){
                int mid = top + (bottom - top) / 2;
                if(matrix[level][mid] == target)
                    return true;
                else if(matrix[level][mid] > target)
                    bottom = mid - 1;
                else
                    top = mid + 1;
            }
            return false;
    }
};
```

: $\boxed{\text{时间复杂度O(lgm + lgn) = O(lg(m*n))，空间复杂度O(1)}}$

```

class Solution {
public:
    bool searchMatrix(vector<vector<int> > &matrix, int target) {
        int n = matrix.size();
        int m = matrix[0].size();
        int l = 0, r = m * n - 1;
        while (l != r){
            int mid = (l + r - 1) >> 1;
            if (matrix[mid / m][mid % m] < target)
                l = mid + 1;
            else
                r = mid;
        }
        return matrix[r / m][r % m] == target;
    }
};
```

1.2.37 Search-insert-position

Question:

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You may assume no duplicates in the array.

Example

(1,3,5,6), 5 → 2
(1,3,5,6), 2 → 1
(1,3,5,6), 7 → 4
(1,3,5,6), 0 → 0

Anslysis

因为数组已排序，所以可以使用二分查找的方法
当最后退出时，left的位置就是第一个大于value的位置

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

```
class Solution {
    int len;
    public:
        int searchInsert(vector<int>& nums, int target) {
            len = nums.size();
            int left=0,right=len-1;
            while(left <= right){
                int mid = left + (right - left) /2;
                if(nums[mid] == target) return mid;
                if(nums[mid] > target){
                    right = mid - 1;
                }else{
                    left = mid + 1;
                }
            }
            //not found, now left
            return left;
        }
};
```

1.2.38 Set-matrix-zeros

Question:

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Anslysis

可使用标记的方法，空间复杂度为O(m+n)

我们也可以考虑如何不使用多余的空间，这样的话如何在原数据中保存我们需要的信息呢?

Solution

1. Tag : $\boxed{\text{时间复杂度O}(m*n)\text{，空间复杂度O}(m+n)}$

```
class Solution {
    public:
        void setZeroes(vector<vector<int> >& matrix) {
            int cols = matrix.size();
            if(cols==0) return ;
            int rows = matrix[0].size();
            if(rows==0) return ;

            vector<bool> colIsZero(cols,false);
            vector<bool> rowIsZero(rows,false);
            for(int i=0;i<cols;++i){
                for(int j=0;j<rows;++j){
                    if(matrix[i][j]==0){
                        colIsZero[i] = true;
                        rowIsZero[j] = true;
                    }
                }
            }
            for(int i=0;i<cols;++i){
                if(colIsZero[i] == true){
                    for(int j=0;j<rows;++j){
                        matrix[i][j] = 0;
                    }
                }
            }
            for(int i=0;i<rows;++i){
                if(rowIsZero[i] == true){
                    for(int j=0;j<cols;++j){
                        matrix[j][i] = 0;
                    }
                }
            }
        }
};
```

2. Use original matrix : $\boxed{\text{时间复杂度O}(m*n)\text{，空间复杂度O}(1)}$

```
            void setZeroes(vector<vector<int> >& matrix) {
                int cols = matrix.size();
                if(cols==0) return ;
                int rows = matrix[0].size();
                if(rows==0) return ;

                bool isColZero=false;
                for(int i=0;i<rows;++i){
                    if(matrix[0][i] == 0){
                        isColZero = true;   break;
                    }
                }
                bool isRowZero = false;
                for(int i=0;i<cols;++i){
                    if(matrix[i][0] == 0){
                        isRowZero = true;   break;
                    }
                }

                for(int i=0;i<cols;++i){
                    for(int j=0;j<rows;++j){
                        if(matrix[i][j]==0){
                            matrix[i][0] = 0;
                            matrix[0][j] = 0;
                        }
                    }
                }

                for(int i=1;i<cols;++i){
                    if(matrix[i][0] == 0){
                        for(int j=1;j<rows;++j)
                            matrix[i][j] = 0;
                    }
                }
                for(int i=1;i<rows;++i){
                    if(matrix[0][i] == 0){
                        for(int j=1;j<cols;++j)
                            matrix[j][i] = 0;
                    }
                }
                if(isColZero){
                    for(int i=0;i<rows;++i)
                        matrix[0][i] = 0;
                }
                if(isRowZero){
                    for(int i=0;i<cols;++i)
```

```
                        matrix[i][0] = 0;
                }
            }
```

1.2.39 Sort-color

Question:

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.
Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Anslysis

因为只有三种数据类型，所以不需要使用完整的排序方法．只需要两遍调整，将 0 和 2 放置到首尾的位置即可

Solution

1. Two Pointer : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        void putZero(vector<int>& nums){
            int prev = -1;
            for(int i=0;i<len;++i){
                if(nums[i] == 0)
                    swap(nums[i],nums[++prev]);
            }
        }
        void putTwo(vector<int>& nums){
            int back = len;
            for(int i=len-1;i>=0;--i){
                if(nums[i] == 2)
                    swap(nums[i], nums[--back]);
            }
        }

        void sortColors(vector<int>& nums) {
            len = nums.size();
            putZero(nums);
            putTwo(nums);
        }
};
```

1.2.40 Spiral-matrix

Question:

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

Anslysis

可分析到子块的构成

(col,col) -¿ (col,maxRow-1-col)
.....
(maxCol-1-col,col] -¿ (maxCol-1-col,maxRow-1-col)
我们需要充足的限制条件:
1. $maxCol - 1 - col \geq col \&\& maxRow - 1 - col \geq col$
2. $col \neq maxCol - 1 - col$?
3. $col \neq maxRow - 1 - col$?

Solution

: $\boxed{\text{时间复杂度O}(m*n)\text{，空间复杂度O}(m*n)}$

```
class Solution {
    public:
        vector<int> spiralOrder(vector<vector<int> >& matrix) {
            vector<int> res;
            int maxCol = matrix.size();
            if(maxCol == 0 )    return res;
            int maxRow = matrix[0].size();
            if(maxRow == 0) return res;

            for(int col=0;col <= (maxCol-1)/2;++col){
                if(maxCol-1-col >= col && maxRow-1-col >= col){
                    for(int i=col;i<=maxRow-1-col;++i)
                        res.push_back(matrix[col][i]);
                    for(int i=col+1;i<maxCol-1-col;++i)
                        res.push_back(matrix[i][maxRow-1-col]);
                    if(maxCol-1-col != col){
                        for(int i=maxRow-1-col;i>=col;--i)
                            res.push_back(matrix[maxCol-1-col][i]);
                    }
                    if(col != maxRow-1-col){
                        for(int i=maxCol-1-col-1;i>col;--i)
                            res.push_back(matrix[i][col]);
                    }
                }
            }

            return res;
        }
};
```

1.2.41 Spiral-matrixII

Question:
Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

Anslysis
可分析到子块的构成
(col,col) -¿ (col,maxRow-1-col)
.....
(maxCol-1-col,col) -¿ (maxCol-1-col,maxRow-1-col)
我们需要充足的限制条件:

1. $maxCol - 1 - col \geq col \&\& maxRow - 1 - col \geq col$
2. $col \neq maxCol - 1 - col$?
3. $col \neq maxRow - 1 - col$?

Solution

: $\boxed{\text{时间复杂度O(n*m)，空间复杂度O(m*n)}}$

```cpp
class Solution {
    public:
        vector<vector<int> > generateMatrix(int n) {
            if(n==0)    return vector<vector<int> >();
            vector<vector<int> > res(n,vector<int>(n,0));
            int number = 0;
            for(int col=0;col <= (n-1)/2;++col){
                if(n-1-col >= col){
                    for(int i=col;i<=n-1-col;++i)
                        res[col][i] = ++number;
                    for(int i=col+1;i<n-1-col;++i)
                        res[i][n-1-col] = ++number;
                    if(n-1-col != col){
                        for(int i=n-1-col;i>=col;--i)
                            res[n-1-col][i] = ++number;
                    }
                    if(n-1-col != col){
                        for(int i=n-1-col-1;i>col;--i)
                            res[i][col] = ++number;
                    }
                }
            }
            return res;
        }
};
```

1.2.42 summary-ranges

Question:
Given a sorted integer array without duplicates, return the summary of its ranges.

Example
Input : [0,1,2,4,5,7]
Output: ["0-¿2","4-¿5","7"]

Anslysis
因为待处理的数字已经排序状态，所以在融合的时候，可以仅查看已排序状态的结尾
可以借助于stack，每次查看结尾的状态，看能不能融合

Solution

1. Hash : $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$

```
class Solution {
public:
    string toString(int i) {
            string str;
            if(i == 0) {
                    str += ('0');
                    return str;
            }

            bool isNeg = i < 0;
            long il = ((long)i);
            if(il < 0)
                    il *= -1;
            while(il) {//无法处理为的情况0
                    str += (il%10 + '0');
                    il /= 10;
            }
            if(isNeg)
                    str += '-';
            reverse(str.begin(),str.end());
            return str;
    }

    vector<string> summaryRanges(vector<int>& nums) {
        vector<string>  result;

        if(nums.size() == 0)    return result;

        //用于清空之前未保存的值
        //num.back()本身不会被处理+2
        nums.push_back(nums.back()-2);

        int rangeBegin = nums[0],curr;//起码会有个元素1+1
        for(int idx=1;idx<nums.size();++idx) {
            curr = nums[idx];

            if(curr != nums[idx-1] + 1) {
                string str;
                if(nums[idx-1] == rangeBegin) {
                    str += toString(rangeBegin);
                }else {
                    str += toString(rangeBegin);
                    str += "->";
                    str += toString(nums[idx-1]);
                }
                result.push_back(str);

                rangeBegin = curr;
            }
        }
        return result;
    }
};
```

1.2.43 Search-for-a-range

Question:

Given a sorted array of integers, find the starting and ending position of a given target value.
Your algorithm's runtime complexity must be in the order of O(log n).
If the target is not found in the array, return [-1, -1].

Example

Input : [5, 7, 7, 8, 8, 10] and target value 8.
Output : [3, 4].

Anslysis

在已排序数组中定位目标值，使用二分
分别寻找左边界和右边界
可使用的方法有三种：１）自己完成二分查找的过程；２）使用stl的lowerbouder（第一个不小于target的位置）和upperbound（第一个大于target的位置）；３）equalrange一次性获取lower和upper

Solution

: $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

```cpp
class Solution {
    int len;
    public:
        int lowerBoud(vector<int>&nums,int target){
            int left=0,right=len-1;
            while(left <= right){
                int mid = left + (right - left) /2;
                if(nums[mid] < target)
                    left = mid + 1;
                else if(nums[mid] > target)
                    right = mid - 1;
                else{
                    if(mid==0 || nums[mid-1]!=target)
                        return mid+1;
                    else
                        right = mid -1;
                }
            }
            return -1;
        }
        int upperBound(vector<int>&nums,int target){
            int left=0,right=len-1;
            while(left <= right){//need to have == here
            int mid = left + (right - left) /2;
                if(nums[mid] > target)
                    right = mid - 1;
                else if(nums[mid] < target)
                    left = mid + 1;
                else{
                    if(mid == len-1 || nums[mid+1]!=target)
                        return mid+1;
                    else
                        left = mid + 1;
                }
            }
            return -1;
        }
        vector<int> searchRange2(vector<int>& nums, int target) {
            len = nums.size();
            if(len == 0)    return vector<int>(2,-1);
            vector<int>::iterator low = lowerBoud(nums,target);
            vector<int>::iterator upp = upperBound(nums,target);
            vector<int> res;
            res.push_back(low - nums.begin());  res.push_back(upp - nums.begin());
            return res;
        }
        vector<int> searchRange1(vector<int>& nums, int target) {
            vector<int>::iterator lo = std::lower_bound(nums.begin(),nums.end(),target);//first elem not less than target
            vector<int>::iterator up = std::upper_bound(nums.begin(),nums.end(),target);//first elem great than target
            if(lo == nums.end() || *lo != target)
                return vector<int>(2,-1);
            vector<int> res;
            res.push_back(lo - nums.begin());   res.push_back(up - nums.begin() -1);
            return res;
        }
        vector<int> searchRange(vector<int>& nums, int target) {
            pair<vector<int>::iterator, vector<int>::iterator > bounds;
            bounds = std::equal_range(nums.begin(), nums.end(), target);
            if(bounds.first == nums.end() || *(bounds.first) != target)
                return vector<int>(2,-1);
            vector<int> res;
            res.push_back(bounds.first - nums.begin()); res.push_back(bounds.second - nums.begin() -1);
            return res;
        }
};
```

1.2.44 Search-in-rotated-sorted-array

Question:

Suppose a sorted array 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).
You are given a target value to search. If found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.

Anslysis

使用循环不变式
若left ¡ right, 则left为所求
若left ¿ right，则求【mid，right】
若left == right，只能一个个找了

Solution

: 时间复杂度O(n)，空间复杂度O(1)

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

};
```

1.2.45 Search-in-rotated-sorted-arrayII

Question:

Follow up for "Search in Rotated Sorted Array":
What if duplicates are allowed?
Would this affect the run-time complexity? How and why?
Write a function to determine if a given target is in the array.

Anslysis

依然使用循环不变式
获取mid后，针对target与mid,left,right的关系来确定left,right指针的移动.
因为会存在重复，所以需要考虑mid,left,right都想等的情况。

Solution

: 时间复杂度O(n)，空间复杂度O(1)

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

};
```

1.2.46 trapping-rain-water.tex

Question:

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

Anslysis

因为只有两个的高度合适，构成一个桶，才能积攒水。

1） 可以寻找最高的位置后，其他的位置都能依赖它而积攒水分.

2） 每次比较当前left，right的较小值，因为这个最小值可能和已存在本侧的max和较大值来组成挡板。这里有个潜在的条件，因为每次处理的都是较小的方向，所以当height[left] ¡ height[right]时，maxLeft一定是小于heigth[right]的.

Solution

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
        int len;
public:
        /*
         * Because the water was constrained by two side of flap
         * It's covenient to get the max height, which means the other
           height can depend on it
         */
        int trap(vector<int>& height) {
            len = height.size();
            if(len == 0)    return 0;

            int MaxheightIdx = 0;//Get the max height for safety
            int currMaxHeight = height[0];
            for(int idx = 1;idx < len;++idx){
                    if(currMaxHeight < height[idx]){
                            currMaxHeight = height[idx];
                            MaxheightIdx = idx;
                    }
            }

            int result = 0;

            //we check the front part
            int partlyMaxHeight = height[0];
            int containedWater = 0;
            for(int idx = 1;idx <= MaxheightIdx; ++idx){
                    if(height[idx] < partlyMaxHeight)
                            containedWater += partlyMaxHeight - height[idx];
                    else{
                            result += containedWater;
                            containedWater = 0;
                            partlyMaxHeight = height[idx];
                    }
            }

            //we then check the back part
            partlyMaxHeight = height[len-1];
            containedWater = 0;
            for(int idx = len-2; idx >= MaxheightIdx; --idx){
                    if(height[idx] < partlyMaxHeight)
                            containedWater += partlyMaxHeight - height[idx];
                    else{
                            result += containedWater;
                            containedWater = 0;
                            partlyMaxHeight = height[idx];
                    }
            }
            return result;
        }
};
```

: 时间复杂度O(n)，空间复杂度O(1)

```
class Solution {
public:
    int trap(int A[], int n) {
        int left=0; int right=n-1;
        int res=0;
        int maxleft=0, maxright=0;
        while(left<=right){
            if(A[left]<=A[right]){
                if(A[left]>=maxleft) maxleft=A[left];
                else res+=maxleft-A[left];
                left++;
            }
            else{
                if(A[right]>=maxright) maxright= A[right];
                else res+=maxright-A[right];
                right--;
            }
        }
        return res;
    }
};
```

第 2 章　Hash

2.1　Background

哈希是用来加速查找的过程，在存在性问题中经常使用。TODO(one-note hash)

2.2 Solutions

2.2.1 anagrams

Question:

Given an array of strings, return all groups of strings that are anagrams.
Note: All inputs will be in lower-case.

Solution

anagram为异位词的意思。既然时查看含有相同数目字符类型的串的存在性，自然可以使用ｈａｓｈ来加速。
如果使用sort后查找的方法，时间复杂度为O(n*n*lgn).
如果将子串转换为数字后处理，时间复杂度为O(m*n).可以为ａ　ｚ选取不同的质数，然后使用乘法来计算每个串的值

Hash : $\boxed{\text{时间复杂度O}(m*n)\text{，空间复杂度O}(n)}$

```java
    private static final int[] PRIMES = new int[]{2, 3, 5, 7, 11 ,13,
        17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
        83, 89, 97, 101, 107};

    public List<String> anagrams(String[] strs) {
        List<String> list = new LinkedList<>();
        Map<Integer, List<String>> mapString = new HashMap<>();
        int result = -1;
        for (int i = 0; i < strs.length; i++){
            int mapping = 1;
            for (int j = 0, max = strs[i].length(); j < max; j++) {
                mapping *= PRIMES[strs[i].charAt(j) - 'a'];
            }
            List<String> strings = mapString.get(mapping);
            if (strings == null) {
                strings = new LinkedList<>();
                mapString.put(mapping, strings);
            }
            strings.add(strs[i]);
        }
        for (List<String> mapList : mapString.values()){
            if (mapList.size() > 1)
                list.addAll(mapList);
        }
        return list;
    }
```

2.2.2 count-primes

Question:

Given an array of non-negative integers, you are initially positioned at the first index of the array.
Each element in the array represents your maximum jump length at that position.
Determine if you are able to reach the last index.

Anslysis

反向思维，寻找出所有非质数后得到质数。

如果顺序的查找，判断，时间复杂度为O($n^(1.5)$)。
当使用Sieve of Eratosthenes方法时，需要处理 2 ∼$n^(1/2)$,每个元素都要处理p*p, p*p+p,p*p+2p …。所以时间复杂度为?TODO

Solution

Hash : 时间复杂度O($nlglgn$)，空间复杂度O(n)

```
class Solution {
    public:
        int countPrimes(int n) {
            vector<bool> isPrime(n+1,true);

            int end = sqrt(n);
            for(int i=2;i<=end;++i) {
                if(isPrime[i]) {
                    for(int j = i * i ; j< n; j += i)
                        isPrime[j] = false;
                }
            }

            int count = 0;
            for(int i=2;i<n;++i) {
                if(isPrime[i])
                    ++count;
            }

            return count;
        }
};
```

2.2.3 fraction-to-recurring-decimal

Question:
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.

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

Anslysis
检查是否存在环：查看处理结果在之前是否出现过。
注意正负数的问题；负数转正数溢出的问题； i n t 溢出的问题。

Solution

Hash : 时间复杂度O(n)，空间复杂度O(n)

```cpp
class Solution {
public:
    string fractionToDecimal(int numerator, int denominator) {
        string result;
        if( (numerator < 0 && denominator > 0) || (numerator > 0 &&
            denominator < 0))
            result += '-';

        //不能直接用，小心absINT_MIN
        long numeratorL = numerator > 0 ? numerator : (-1) * (long)
            numerator;
        long denominatorL = denominator > 0 ? denominator : (-1) * (long
            )denominator;

        int idx = 0;
        long integer = numeratorL / denominatorL;
        result += (integer > 0) ? to_string(integer) : "0";
        numeratorL -= integer*denominatorL;

        if(numeratorL) {
            result += '.';

            //某个出现时前段开始的位置numeratorL
            unordered_map<int,int> isSeen;

            while(numeratorL) {
                if(isSeen.find(numeratorL) == isSeen.end()) {
                        //当前这个开始的位置是numeratorLresult.size()
                        isSeen.insert(pair<int,int>(numeratorL,result.
                            size()));

                        numeratorL *= 10;
                        integer = numeratorL / denominatorL;

                        result += to_string(integer);

                        numeratorL -= integer*denominatorL;
                }else{
                        //在 重复串开头前加上号'('
                        result.insert(isSeen[numeratorL],"(");
                        result += ")";
                        break;
                }
            }
        }

        return result;
    }
};
```

2.2.4 happy-number

Question:

Write an algorithm to determine if a number is ”happy”.

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example

19 is a happy number
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1

Anslysis

需要查找当前处理的结果在之前是否出现过。

Solution

Hash : 时间复杂度O(n) , 空间复杂度O(n)

```
class Solution {
    public:
        int getSquareSum(int n) {
            int squareSum = 0;
            while(n) {
                squareSum += ((n%10) * (n%10));
                n /= 10;
            }
            return squareSum;
        }
        bool isHappy(int n) {
            int squareSum = getSquareSum(n);
            int beginLoop = squareSum;
            while(squareSum != 1) {
                squareSum = getSquareSum(squareSum);

                if(squareSum == beginLoop)
                    return false;
            }
            return true;
        }
};
```

2.2.5 substring-with-concatenation-of-all-words

Question:

You are given a string, s, and a list of words, words, that are all of the same length. Find all starting indices of substring(s) in s that is a concatenation of each word in words exactly once and without any intervening characters.

Example

s: "barfoothefoobarman"
words: ("foo", "bar")
You should return the indices: (0,9).
(order does not matter).

Anslysis

对于这个问题，我们需要检查每个位置为起点的子串的情况。在检查子串时，因为ｗｏｒｄｓ的长度都一样，所以可以很容易的确定待判断的子序列ｓ。之后需要确定ｓ是否存在与ｗｏｒｄｓ中，对于这个存在性问题，自然可以使用ｈａｓｈ来加速。

Solution

Hash : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

2.2.6 single-number.tex

Question:

Given an array of integers, every element appears twice except for one. Find that single one. Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Anslysis

偶次出现： 异或（Exclusive OR）。

Solution

Hash : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```cpp
class Solution {
    public:
        int singleNumber(vector<int>& nums) {
            int res = nums[0];
            for(int i=1;i<nums.size();++i)
                res ^= nums[i];
            return res;
        }
};
```

2.2.7 valid-sudoku

Question:

Determine if a Sudoku is valid.

A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

Anslysis

对于(i)(j)位置的某个元素k，我们需要查看行，列，三角形中是否已经存在了k。对于这个查找过程，可以使用ｈａｓｈ来加速。

在本问题中，因为待确定问题的状态空间只有１～９，所以可以使用数组解决。

Solution

Hash : $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O}(n)}$

```cpp
class Solution
{
public:
    bool isValidSudoku(vector<vector<char> > &board)
    {
        int used1[9][9] = {0}, used2[9][9] = {0}, used3[9][9] = {0};

        for(int i = 0; i < board.size(); ++ i)
            for(int j = 0; j < board[i].size(); ++ j)
                if(board[i][j] != '.')
                {
                    int num = board[i][j] - '0' - 1, k = i / 3 * 3 + j /
                        3;
                    if(used1[i][num] || used2[j][num] || used3[k][num])
                        return false;
                    used1[i][num] = used2[j][num] = used3[k][num] = 1;
                }

        return true;
    }
};
```

2.2.8 maximal-rectangle

Question:

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum ⩾ s. If there isn't one, return 0 instead.

Example

Given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length under the problem constraint.

Anslysis

类似于minimal-window-substring，维护双指针来表征区间，然后不停的更新区间即可。因为每个元素最多被访问 2 次，所以时间复杂度为 O (n）。
因为二分只适用于已排序的对象，所以我们必须获得一个已排序的对象。直接对nums做sort显然没有意义。考虑到nums中都是positive的，所以可以构造出累加和来作为待处理的对象。这样就满足了二分的基础。

Solution

Hash : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(n)}$

```cpp
class Solution {
    public:
        int countPrimes(int n) {
            vector<bool> isPrime(n+1,true);

            int end = sqrt(n);
            for(int i=2;i<=end;++i) {
                if(isPrime[i]) {
                    for(int j = i * i ; j< n; j += i)
```

```
                        isPrime[j] = false;
                    }
                }

                int count = 0;
                for(int i=2;i<n;++i) {
                    if(isPrime[i])
                        ++count;
                }

                return count;
            }
};
```

Binary Search : $\boxed{\text{时间复杂度O}(nlgn)\text{，空间复杂度O}(n)}$

```
    private int solveNLogN(int s, int[] nums) {
        int[] sums = new int[nums.length + 1];
        for (int i = 1; i < sums.length; i++) sums[i] = sums[i - 1] +
            nums[i - 1];
        int minLen = Integer.MAX_VALUE;
        for (int i = 0; i < sums.length; i++) {
            int end = binarySearch(i + 1, sums.length - 1, sums[i] + s,
                sums);
            if (end == sums.length) break;
            if (end - i < minLen) minLen = end - i;
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    private int binarySearch(int lo, int hi, int key, int[] sums) {
        while (lo <= hi) {
           int mid = (lo + hi) / 2;
           if (sums[mid] >= key){
               hi = mid - 1;
           } else {
               lo = mid + 1;
           }
        }
        return lo;
    }
```

2.2.9 max-points-on-a-line

Question:

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

Anslysis

本题的问题在于: 当使用double为索引来建立ｈａｓｈ时，无法通过比较来得到特定斜率的存在性

所以需要使用x和y来建立二维的ｈａｓｈ。

Solution

Hash : $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O}(n)}$

```cpp
class Solution {
public:
    int maxPoints(vector<Point> &points) {

        if(points.size()<2) return points.size();

        int result=0;

        for(int i=0; i<points.size(); i++) {

            map<pair<int, int>, int> lines;
            int localmax=0, overlap=0, vertical=0;

            for(int j=i+1; j<points.size(); j++) {

                if(points[j].x==points[i].x && points[j].y==points[i].y)
                    {

                    overlap++;
                    continue;
                }
                else if(points[j].x==points[i].x) vertical++;
                else {

                    int a=points[j].x-points[i].x, b=points[j].y-points[
                        i].y;
                    int gcd=GCD(a, b);

                    a/=gcd;
                    b/=gcd;

                    lines[make_pair(a, b)]++;
                    localmax=max(lines[make_pair(a, b)], localmax);
                }

                localmax=max(vertical, localmax);
            }

            result=max(result, localmax+overlap+1);
        }

        return result;
    }

private:
    int GCD(int a, int b) {

        if(b==0) return a;
        else return GCD(b, a%b);
    }
};
```

2.2.10 minimal-window-substring

Question:

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

Example

S = "ADOBECODEBANC"
T = "ABC"
Minimum window is "BANC".

Anslysis

需要留意在计算某个状态时，可以依据它的前一个状态来更新。
对于T中的每个元素，我们既需要保存它的数目，有需要它的存在性。
不同于连续子串的哈希，这里无法转换成数字来节省空间。
因为每个元素最多被处理2次，所以时间复杂度为O(2*N) = O(N)。

Solution

Hash : $\boxed{\text{时间复杂度O}(n)\text{ , 空间复杂度O}(n)}$

```cpp
class Solution {
    public:
        string minWindow(string s, string t) {
            if(s.empty() || t.empty())
                return "";

            //出现的次数 + 存在性
            int times[256] = {0};
            bool isSeen[256] = {false};
            for(int i=0;i<t.size();++i) {
                ++times[t[i]];
                isSeen[t[i]] = true;
            }

            int begin=0,end=-1;
            int count = t.size();
            int minLen = INT_MAX;
            int minBegin = -1;
            //这里需要考虑当最后一个元素满足count，可是无法被处理的情况【==0上面被处
                理后，不会进入下面】if
            //首先，考虑在后面插入辅助字符，这样必须选择中没有出现过的字符，不具有扩展
                性st
            //其次，考虑将后移一位。end
            int slen = s.size();
            while(begin < slen && end < slen) {
                if(count) {
                    ++end;
                    //只受中元素影响countt
                    if(isSeen[s[end]] && times[s[end]] > 0)
                        --count;
                    --times[s[end]];
                }else{
                    if(end - begin + 1 < minLen) {
                        minBegin = begin;
                        minLen = end - begin + 1;
                    }

                    ++times[s[begin]];
                    if(isSeen[s[begin]] && times[s[begin]] > 0)
                        ++count;
                    ++begin;
                }
            }

            if(minLen == INT_MAX)
                return "";
            return s.substr(minBegin,minLen);
        }
};
```

2.2.11 repeated-DNA-sequences

Question:

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

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT".

Return: ("AAAAACCCCC", "CCCCCAAAAA").

Anslysis

查找子串的存在性可以用h a s h来加速，这里的问题是如何减少h a s h中存储的对象的大小?

因为子串全部为1 0的长度，如果直接存储的话，每个子串都消耗1 0字节。因为子串中只会出现四种元素，所以可以考虑将子串转换为数字后处理。

此外，需要注意在生成下一个子串的哈希值时可以参考上一个子串的哈希值。

A is 0x41, C is 0x43, G is 0x47, T is 0x54

A is 0101, C is 0103, G is 0107, T is 0124

Solution

Hash : 时间复杂度O(n)，空间复杂度O(1)

```
vector<string> findRepeatedDnaSequences(string s) {
    unordered_map<int, int> m;
    vector<string> r;
    int t = 0, i = 0, ss = s.size();
    while (i < 9)
        t = t << 3 | s[i++] & 7;
    while (i < ss)
        if (m[t = t << 3 & 0x3FFFFFFF | s[i++] & 7]++ == 1)
            r.push_back(s.substr(i - 10, 10));
    return r;
}
```

2.2.12 isomorphics-string

Question:

Given two strings s and t, determine if they are isomorphic.
Two strings are isomorphic if the characters in s can be replaced to get t.
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

Solution

最重要的点就是咨询字符的范围，ASCII码足够吗？那样的话可以借助char[256]来处理

: $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O(n)}}$

```
class Solution {
    public:
        bool isIsomorphic(string s, string t) {
            unordered_map<char,char> replaceTo;
            unordered_set<char> isUsed;

            for(int i=0;i<s.size();++i) {
                if(replaceTo.find(s[i]) == replaceTo.end()) {
                    if(isUsed.find(t[i]) == isUsed.end()) {
                        replaceTo.insert(pair<char,char>(s[i],t[i]));
                        isUsed.insert(t[i]);
                    }else
                        return false;
                }else{
                    if(replaceTo.find(s[i])->second != t[i])
                        return false;
                }
            }
            return true;
        }
};
```

2.2.13 longest-substring-without-repeating-characters

Question:

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

Anslysis

我们需要想面试管确认字符的类型，是８位就能表示的char还是１６位的ＵＮＩＣＯＤＥ？
在每次遇到重复的字符后，只需要更新该字符出现的位置信息。

Solution

Hash : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(n)}$

```cpp
class Solution {
    public:
        int lengthOfLongestSubstring(string s) {
            vector<int> isSeen(256, -1);

            int maxLen = 0,begin =0;
            for(int i=0 ; i < s.size() ; ++i) {
                if(isSeen[s[i]] >= begin) {
                    maxLen = max(maxLen, i-isSeen[s[i]]);
                    begin = isSeen[s[i]] + 1;
                }
                isSeen[s[i]] = i;
            }
            maxLen = max(maxLen, (int)s.size() - begin);

            return maxLen;
        }
};
```

第 3 章　Dynamic Programming

3.1　Background

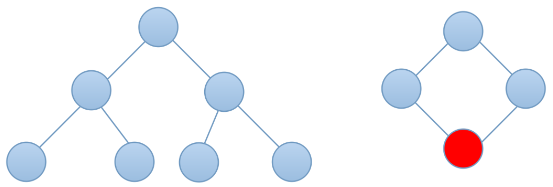
动态规划是运筹学的一个分支，是求解决策过程（ｄｅｃｉｓｉｏｎ　ｐｒｏｃｅｓｓ）最优化的数学方法，它将多阶段过程转换为一系列多阶段问题，利用各个阶段之间的关系逐个求解．动态规划的方法的基础是最优子结构和子问题重叠，无后效性．

最优子结构意味着当前问题的最优解包含了子问题的最优解．在论证问题的最优子结构性质时可以使用"ｃｕｔ－ａｎｄ－ｐａｓｔｅ"的方法，即假设子问题的解并非最优解，然后"ｃｕｔ"非最优子问题，"ｐａｓｔｅ"最优解来导出出矛盾．

子问题独立（无后效性）意味着将某个问题划分为多个ｓｔａｇｅ时，各个ｓｔａｇｅ并不使用共同的资源（矩阵连乘中子矩阵相互独立［并行的］，最短路径中先后选取的两条路径无多于一个公共节点［先后的］）.对于子阶段并不相互独立的情况（最长路径中两条路径可能使用公共节点），不能使用动态规划来求解．

子问题重叠意味着求解问题时，可能需要反复的求解某些子问题，动态规划方法中通过使用备忘录来记录已求解的子问题来避免重复的计算，可以常数时间内查看．［注意，适合分治法来处理的问题的子问题都是互异的．］

动态规划以自底向上的方法来进行阶段(ｓｔａｇｅ）决策（ｄｅｃｉｓｉｏｎ）,首先获取某阶段（ｓｔａｇｅ）的最优解，以此来计算后一阶段（ｓｔａｇｅ）．具体的实现可以分为两种：（１）Ｔｏｐ－ｄｏｗｎ，Ｒｅｃｕｒｓｉｖｅ．这种方法的好处是能自动的处理计算顺序，子问题总是被先计算；只会处理必要的自问题，不需要处理全部的状态空间．缺点是采用递归的结构,执行效率差；无法自由的控制计算顺序，导致无法妥善使用存储空间；（２）Ｂｏｔｔｏｍ－ｕｐ，Ｉｔｅｒａｔｉｖｅ．大体来讲，动态规划方法的复杂度依赖于子阶段（ｓｔａｔｅ）的总个数和每个子问题的状态（ｓｔａｔｅ）的乘积．



注意：描述子问题的空间时，可以遵循从简到繁的步骤．尽量使用最简单的空间来解决问题（维度最小）．

3.2 Classical Problem

动态规划的经典题目可以参见戴方勤的<手写代码必备>．在此包含几道算法导论中题目的论证过程，以供学习．

3.2.1 Strassen's Algorithm

矩阵连乘是动态规划中的经典题目．

如果通过穷举的方法，它的递归关系为：

$$P(n)\begin{cases}1 & \text{if } n=1\\ \sum_{n-1}^{k=1} P(k)*P(n-k) & \text{if } n\gg 2\end{cases}$$

$$P(n)\geq 1+\sum_{k=1}^{n-1}(P(k)+P(n-k)+1))\rightarrow P(n)\geq n+2*\sum_{k=1}^{n-1}P(k)$$

通过替换法可以证明:$P(n)=\Omega(2^n)$

1)．最优子结构性质：如果对$A_i...A_j$的最优解分解为$A_i...A_k$和$A_k...A_j$.则$A_i...A_k$和$A_k...A_j$也必须为最优的．如果$A_i...A_k$或$A_k...A_j$中可以为非最优，那么使用最优解来替换掉它可以得到更优的$A_i...A_j$．矛盾．

2)．构造最优解：将问题分割为两个子问题，然后根据子问题的最优解来合并．

3)．递归定义：$m[i,j]\begin{cases}0 & i=j\\ \min_{k\in[i,j)} m[i,k]+m[k+1,j]+p_{i-1}*p_k*p_j & i<j\end{cases}$

4)．计算代价：原问题只有$\binom{2}{n}+n=\theta(n^2)$个子问题．使用自底向上方法计算求解．

3.2.2 Longest Common Substring

LCS的题目为有两个字符串$x=x_1,x_2....x_n$和$y=y_1,y_2....y_n$,求X串和Y串的最长公共子串，这个子串可以是非连续的．

1)． 证明问题具有最优子结构性质．假设$x=x_1,x_2....x_m$和$y=y_1,y_2....y_n$的最长公共子串为$z=z_1,z_2....z_k$,则存在如下的递归关系：

$$LCS_{m,n}=\begin{cases}LCS_{m-1,n-1}+1 & x_m=y_n\\ \max(LCS_{m-1,n},LCS_{m,n-1}) & x_m\neq y_n\end{cases}$$

对于第一种情况，（1）如果$z_k\neq y_n$,那么可以将y_n加入到$LCS_{m,n}$得到更长的公共子串，矛盾．（2）如果$LCS_{m-1,n-1}$具有比Z_{k-1}更长的子串，那么使用更长的子串来替换，会的到比Z_k更长的子串，矛盾．

对于第二种情况，若X_{m-1}和Y_n具有比$LCS_{m-1,n}$更长的子串，那么使用它来替换$LCS_{m,n}$会的到更优的解，矛盾．

第三种情况可以参考第二种情况的证明．

2)．计算代价．原问题只有$O(m*n)$个子问题．此外，我们可以尝试状态压缩来节省空间开销．状态压缩对于只求解最优解的问题有很大帮助，但是对于需要恢复路径的问题不适宜．

3.2.3 Optimal Binary Search Tree

3.3 Solutions

3.3.1 Word Break

Question:

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

Example:

s = "leetcode",
dict = ["leet", "code"].
Return true because "leetcode" can be segmented as "leet code".

Anslysis

假设p(i)存储bool值，表示s[0,i]的子串是否可以使用dict划分.则:
$p(i) = InDict(s[i,k])\&\&p(k-1), 0 \le k < i$

Solution

1. DFS : $\boxed{\text{时间复杂度O}(2^n)\text{ , 空间复杂度O(n) TLE}}$

```cpp
class Solution {
    int maxLen;
    bool isValid;
    public:
        void btrack(string s,unordered_set<string>& wordDict,int curr,
            string &tmpStr){
            if(isValid) return ;
            if(curr == s.size()){
                isValid = wordDict.find(tmpStr) != wordDict.end();
                return ;
            }
            if(tmpStr.size() > maxLen || tmpStr.size()==maxLen &&
                wordDict.find(tmpStr)==wordDict.end()) return ;
            tmpStr.push_back(s[curr]);//append
            btrack(s,wordDict,curr+1,tmpStr);
            tmpStr.pop_back();
            if(!isValid && wordDict.find(tmpStr) != wordDict.end()){//as
                 new
                string str(tmpStr);
                tmpStr = s[curr];
                btrack(s,wordDict,curr+1,tmpStr);
                tmpStr = str;
            }
        }
        int getMaxLen(unordered_set<string>& wordDict){
            maxLen = 0;
            for(auto str : wordDict)
                maxLen = maxLen > str.size() ? maxLen : str.size();
        }
        bool wordBreak(string s, unordered_set<string>& wordDict) {
            getMaxLen(wordDict);
            string tmpStr;
            isValid = false;
            btrack(s,wordDict,0,tmpStr);
            return isValid;
        }
};
```

2. DP : $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O(n)}}$
   最优子结构的性质：如果s[1,k]可以被dict划分，s[k,j]又存在于dict中，那么自然s[i,j]就可以被dict划分；
   子问题重叠：对于s="bbcandaa"和dict=["b","bc","and","anda","a","aa"]的情况，在计算and时，and之前的可划分性（p[0,2]）被计算；在之后处理anda时，anda之前的可划分性(p[0,2])又会被需要.
   无后效性：对于某个位置的决策只依赖于前一位置是否可划分，并且只依赖于之前的某一个决策（无并行后效性）；也无后方后效性.

```
class Solution {
    public:
        bool wordBreak(string s, unordered_set<string>& wordDict) {
            vector<bool> dp(s.size()+1,false);//dp[i] -> [0,i-1]
            dp[0] = true;
            for(int i=0;i<s.size();++i){
                for(int j=i;j>=0;--j){//[j,i]
                    if(dp[j+1-1]){//[0,j-1] == true;
                        string subStr = s.substr(j,i-j+1);
                        if(wordDict.find(subStr)!=wordDict.end()){
                            dp[i+1] = true;
                            break;
                        }
                    }
                }
            }
            return dp[s.size()];
        }
};
```

   只需一个维度即可解决问题

3.3.2 Word BreakII

Question:
   Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.Return all such possible sentences.

Example:
   s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"]. A solution is ["cats and dog", "cat sand dog"].

Anslysis
   假设p(i)存储bool值，表示s[0,i]的子串是否可以使用dict划分.则:
   $p(i) = InDict(s[i,k])\&\&p(k-1), 0 \leq k < i$

Solution

1. DFS : $\boxed{\text{时间复杂度O}(2^n)\text{，空间复杂度O(n) TLE}}$

```
class Solution {
    private:
        string subStr,str;
        vector<string> res;
        unordered_set<string> myDict;
        int longestLen;
    public:
        void btrack(string s,int currIdx,string& subStr){
            if(currIdx == s.size()){
                if(subStr.size()==0){
                    res.push_back(str);
                }else if(myDict.find(subStr)!=myDict.end()){
                    string origStr(str);
                    str+= ' ';
                    str += subStr;
                    res.push_back(str);
                    str = origStr;
                }
                return ;
            }
            //not merge current -> subStr must be valid
            if(myDict.find(subStr)!=myDict.end()){
                string origStr(str);
                string origSubStr(subStr);

                str+= ' ';//always have a leading blank
                str += subStr;
                subStr = s[currIdx];
                btrack(s,currIdx+1,subStr);
                subStr = origSubStr;
                str = origStr;
            }
            subStr.push_back(s[currIdx]);
            if(subStr.size() <= longestLen)//too long to discard
                btrack(s,currIdx+1,subStr);
            subStr.pop_back();
        }
        void getLongestLen(){
            longestLen = 0;
            for(auto iter=myDict.begin();iter!=myDict.end();++iter){
                longestLen = max(longestLen, (int)(*iter).size());
            }
        }

        vector<string> wordBreak(string s, unordered_set<string>&
            wordDict) {
            myDict = wordDict;
            getLongestLen();
            btrack(s,0,subStr);
            return res;
        }
};
```

2. DP : $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O}(n^2)}$

```
class Solution {
    int len;
    vector<string> vs;
    vector<string> path;
    public:
        void genPath(string& s,int currIdx,vector<vector<int> > &dp){
            if(currIdx == 0){
                //从中生成tmp
                string str;
                for(int i=path.size()-1;i>=0;--i){
                    str += path[i];
                    str += ' ';
                }
                str.pop_back();

                vs.push_back(str);
                return ;
            }

            for(int i = 0 ; i < dp[currIdx].size() ; ++i){
                string part =  s.substr(dp[currIdx][i]-1 , currIdx - dp[currIdx][i] + 1);
                path.push_back(part);
                genPath(s, dp[currIdx][i]-1,dp);
                path.pop_back();
            }
        }

        vector<string> wordBreak(string s, unordered_set<string>& wordDict) {
            len = s.size();
            if(len ==0 || wordDict.size() ==0 ) return vs;

            vector<vector<int> > dp(len+1,vector<int>());
            dp[0].push_back(-1);

            for(int i=0;i<len;++i){
                for(int j=0;j<=i;++j){
                    if(wordDict.find(s.substr(j,i-j+1)) != wordDict.end() && dp[j].size() != 0){
                        dp[i+1].push_back(j+1);
                    }
                }
            }

            genPath(s,len,dp);
            return vs;
        }
};
```

与之前的题目相比多了恢复路径的过程，这就需要我们在dp时存储中间跳转信息.

3.3.3 Wildcard Matching

Question:

Implement wildcard pattern matching with support for '?' and '*'.
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial).

Example:

isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false

Anslysis
要注意的一点时，这道题需要提前的剪枝．预先判断s和p的实体长度来作比较．

Solution

1. Greedy : $\boxed{\text{时间复杂度O}(n*m)\text{，空间复杂度O}(1)\text{ TLE}}$
证明！

```
class Solution {
    private:
        int slen,plen;
    public:
        bool isSame(char c1,char c2){
            return c2=='?'||c1==c2;
        }

        bool isMatch(string s, string p){
            slen = s.size();    plen = p.size();

            int pStrongLen = 0;
            for(int i=0;i<plen;++i){
                if(p[i] != '*')
                    ++pStrongLen;
            }
            if(pStrongLen > slen)
                return false;//even * all became empty,still longer

            int sidx=0,pidx=0;
            int currSIdx=-1,currPIdx=-1;
            while((sidx < slen && pidx < plen) || (currSIdx >= 0 &&
                currSIdx < slen)){
                if(sidx == slen && pidx == plen)    return true;
                if(sidx == slen){
                    while(pidx < plen && p[pidx] == '*')
                        ++pidx;
                    if(pidx == plen)
                        return true;
                    else{
                        sidx = ++currSIdx;
                        pidx = currPIdx;
                        continue;
                    }
                }
                if(pidx == plen){
                    sidx = ++currSIdx;
                    pidx = currPIdx;
                    continue;
                }

```

```
                    if(p[pidx] == '*'){
                        currSIdx = sidx;
                        currPIdx = ++pidx;
                    }else{
                        if(isSame(s[sidx],p[pidx])){
                            ++sidx;     ++pidx;
                        }else if(currSIdx >= 0 && currSIdx < slen){
                            sidx = ++currSIdx;
                            pidx = currPIdx;
                        }else{
                            return false;
                        }
                    }
                }

                //currSIdx is used-up
                if(sidx == slen && pidx == plen)    return true;
                if(sidx == slen){
                    while(pidx < plen && p[pidx] == '*')
                        ++pidx;
                    return pidx == plen;
                }
                if(pidx == plen)    return false;
            }
};
```

2. DP : $\boxed{\text{时间复杂度O}(m*n)\text{ , 空间复杂度O}(m*n)\text{ TLE}}$
本题目子问题有明显的重叠性，独立性．适用于DP求解．
注意使用轮转数组来避免MLE．在使用轮转数组时，我需要注意的是在复用之前被抛弃的维度空间时，要全部清理这个维度，也就是对于这个维度的每个空间都填充上自己对应的值．第一次做这道题时，只处理了true的情况，未合理的填充false，到时复用了前一维度的true．

```
class Solution {
    private:
        int slen,plen;
    public:
        bool isSame(char c1,char c2){
            return c2=='?'||c1==c2;
        }

        bool isMatch(string s, string p){
            int m = s.size();
            int n = p.size();

            int numStar=0;
            for(int i=0;i<n;++i)
                if(p[i] == '*')
                    ++numStar;
            if(n - numStar > m) return false;

            vector<vector<bool> > dp(2,vector<bool>(n+1,false));

            dp[0][0] = true;
            for(int i=1;i<=n && dp[0][i-1];++i){
                if(p[i-1] == '*')
                    dp[0][i] = true;
            }

            for(int i=1;i<=m;++i){
                dp[i%2][0] = false;
                for(int j=1;j<=n;++j){
                    if(p[j-1] != '*'){
                        if(isSame(s[i-1], p[j-1]) && dp[(i-1)%2][j-1])
                            dp[i%2][j] = true;
                        else
                            dp[i%2][j] = false;
                    }else{
                        if(dp[(i-1)%2][j] || dp[i%2][j-1])
                            dp[i%2][j] = true;
                        else
                            dp[i%2][j] = false;
                    }
                }
            }
            return dp[m%2][n];
        }
};
```

3. Backtracking : $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O(n)}}$

```
```

第 4 章　Binary Search

4.1　Background

二分查找又称为折半查找，它要求待查找的数组按照某键值有序的存放．每次都选取数组的中间位置的key与带寻找的key值作比较，以此来排除掉一半的范围．

从二分查找延伸而来的算法有，查找在不破坏有序状态下可插入key的第一个位置(lowerbound),和查找在不破坏有序状态下可插入key的最后一个位置(upperbound).

在ＳＴＬ中包含了二分查找相关的泛型算法，binary_search, lower_bound, upper_bound.

4.2 Classical Problem

动态规划的经典题目可以参见戴方勤的<手写代码必备>. 在此包含几道算法导论中题目的论证过程, 以供学习.

4.2.1 STL Lower Bound

```
/*
 * lower_bound有序区间上()
 *
 * 返回第一个 >=value 的地址
 * [begin,end)中返回pos使得,[begin,pos)都<value,[pos,end)中元素都>=value
 */
template<typename ForwardIterator, typename Tp, typename Distance>
ForwardIterator _lower_bound(ForwardIterator begin,ForwardIterator end,
    const Tp& value,Distance*,forward_iterator_tag) {
    Distance len = 0, half = 0;
    distance(begin,end,len);

    ForwardIterator mid = begin;
    //使用了和结合的方式beginlen因为,的局限性ForwardIterator
    while(len) {
        half = len / 2;
        mid = advance(begin,half);

        if(*mid < value) {
            begin = ++mid;//<的值都可以跳过value
            len = len - half - 1;
        }else {
            len = half;
            //因为是求lower_bound
            //当前这个>=的value*可能为最终结果mid所以不能舍弃,
        }
    }
    return begin;
}
template<typename RandomAccessIterator,typename Tp,typename Distance>
RandomAccessIterator _lower_bound(RandomAccessIterator begin,
    RandomAccessIterator end,const Tp& value,Distance,
    random_access_iterator_tag) {
    Distance half = 0 , len = end - begin;

    RandomAccessIterator mid = begin;
    while(len) {
        half = len / 2;
        mid = begin + half;

        if(*mid < value) {
            len = len - half - 1;
            begin = ++mid;
        }else{
            len = half;
        }
    }
    return begin;
}
template<typename ForwardIterator,typename Tp>
```

```
ForwardIterator lower_bound(ForwardIterator begin,ForwardIterator end,
    const Tp& value) {
    typename iterator_traits<ForwardIterator>::difference_type distance;
    typename iterator_traits<ForwardIterator>::iterator_category ic;
    return _lower_bound(begin,end,value,distance,ic);
}
```

4.2.2 STL Upper Bound

```
/*
 * lower_bound有序区间上()
 *
 * 返回第一个 >=value 的地址
 * [begin,end)中返回pos使得,[begin,pos)都<value,[pos,end)中元素都>=value
 */
template<typename ForwardIterator, typename Tp, typename Distance>
ForwardIterator _lower_bound(ForwardIterator begin,ForwardIterator end,
    const Tp& value,Distance*,forward_iterator_tag) {
    Distance len = 0, half = 0;
    distance(begin,end,len);

    ForwardIterator mid = begin;
    //使用了和结合的方式beginlen因为,的局限性ForwardIterator
    while(len) {
        half = len / 2;
        mid = advance(begin,half);

        if(*mid < value) {
            begin = ++mid;
            //<的值都可以跳过value
            len = len - half - 1;
        }else {
            len = half;
            //因为是求lower_bound
            //当前这个>=的value*可能为最终结果mid所以不能舍弃,
        }
    }
    return begin;
}
template<typename RandomAccessIterator,typename Tp,typename Distance>
RandomAccessIterator _lower_bound(RandomAccessIterator begin,
    RandomAccessIterator end,const Tp& value,Distance,
    random_access_iterator_tag) {
    Distance half = 0 , len = end - begin;

    RandomAccessIterator mid = begin;
    while(len) {
        half = len / 2;
        mid = begin + half;

        if(*mid < value) {
            len = len - half - 1;
            begin = ++mid;
        }else{
            len = half;
        }
    }
```

```
        return begin;
    }
    template<typename ForwardIterator,typename Tp>
    ForwardIterator lower_bound(ForwardIterator begin,ForwardIterator end,
        const Tp& value) {
        typename iterator_traits<ForwardIterator>::difference_type distance;
        typename iterator_traits<ForwardIterator>::iterator_category ic;
        return _lower_bound(begin,end,value,distance,ic);
    }
```

4.3 Solutions

4.3.1 Divide Two Integer

Question:

Divide two integers without using multiplication, division and mod operator.
If it is overflow, return MAX_INT.

Anslysis

假设被除数为s，除数为t，则s和 t 之间存在$s/t = k$.
$=> s = k * t = ((0/1) * 2^m + (0/1) * 2^{m-1} + ... + (0/1) * 2^0) * t$
我们可以使用移位操作作来判断每个位置的存在性.
这里的性质说明了，任意正整数都可以表示为2^n的和，而确定存在性的过程可以使用移位操作来二分的处理.
这题目无法使用*，所以需要对t做移位操作作来模拟乘二的行为.
要处理 0 为除数， 0 为被除数，有正有负，除不尽，能除尽的情况
在进行处理时，推荐使用long long类型，因为int类型在临界情况下会溢出，造成死循环.

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

```
class Solution {
    public:
        int divide(int dividend, int divisor) {
            if(divisor == 0)    return INT_MAX;
            if(dividend == 0)   return 0;

            int flag = (dividend>0&&divisor<0 || dividend<0&&divisor>0)
                ?-1:1;
            long long ldividend = abs((long long)dividend);
            //注意啊！无法使用来接受int
            long long ldivisor = abs((long long)divisor);
            if(ldividend < ldivisor)    return 0;
            if(ldividend == ldivisor)   return flag;

            long long result = 0;
            //在处理／1时会造成（ｉｎｔINT_MINresult)溢出
            while(ldividend >= ldivisor){
                long long tmpRes = 1;
                long long moreDivisor = ldivisor;
                //这里需要使用long long
                //不然可能造成溢出后进入死循环!
                while((moreDivisor << 1) <= ldividend){
                    moreDivisor <<= 1;  tmpRes <<= 1;
                }
                result += tmpRes;
                ldividend -= moreDivisor;
            }
            if(flag == -1){
                if(result >= -1 * (long long)(INT_MIN))
                    return INT_MIN;
                else
                    return flag * result;
            }else{
                if(result >= (long long)INT_MAX)
                    return INT_MAX;
                else
                    return flag *result;
            }
        }
};
```

Binary Search : $\boxed{\text{时间复杂度O(1)?，空间复杂度O(1)}}$

因为结果为int类型，所以只需要考虑 3 2 位的移动情况，所以可以分次遍历divisor所能右移的情况.
注意，需要从 3 1 到 0，

```
        int divide(int dividend, int divisor) {
            if(divisor == 0)    return INT_MAX;
            if(dividend == 0)   return 0;

            int flag = (dividend>0&&divisor<0 || dividend<0&&divisor>0)
                ?-1:1;
            long long ldividend = abs((long long)dividend);
            //注意啊！无法使用来接受int
            long long ldivisor = abs((long long)divisor);
            if(ldividend < ldivisor)    return 0;
            if(ldividend == ldivisor)   return flag;

            long long result = 0;
            for(int i=31;i>=0;--i){
                if((ldivisor<<i) <= ldividend){
                    result += ((long long)1)<<i;
                    //需要使用(long long), 1
                    //否则1<<得到后转换为iintlong long
                    ldividend -= (ldivisor<<i);
                }
            }
            if(flag == -1){
                if(result >= -1 * (long long)(INT_MIN))
                    return INT_MIN;
                else
                    return flag * result;
            }else{
                if(result >= (long long)INT_MAX)
                    return INT_MAX;
                else
                    return flag *result;
            }
        }
```

4.3.2 Find Minimum In Rotated Sorted Array

Question:
Divide two integers without using multiplication, division and mod operator.
If it is overflow, return MAX_INT.

Anslysis

使用循环不变式（loop invariant）.
根据实际情况来决定left,right的移动程度

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

```
class Solution {
    public:
        int findMin(vector<int>& nums) {
            int len = nums.size();
            //if(len == 0)  return ?;
            int left = 0,right = len -1 ;
            while(left < right){
                if(nums[left] > nums[right]){
                    int mid =  (left & right) + ((left ^ right) >> 1);
                    //avoid overflow
                    //小心，的优先级高于+>>
                    if(nums[mid] >= nums[left])
                        left = mid + 1;
                    else
                        right = mid;
                }else{
                    break;
                }
            }
            return nums[left];
        }
};
```

4.3.3 Find Minimum In Rotated Sorted ArrayII

Question:

Divide two integers without using multiplication, division and mod operator.
If it is overflow, return MAX_INT.

Anslysis

需要注意在$mid = left = right$时，必须检测mid两侧，可能会退化为O(n)的方法.
使用$low + (high - low)/2$来避免位运算的优先级问题

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

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

4.3.4 Find Peak Element

Question:

A peak element is an element that is greater than its neighbors.
Given an input array where $num[i] \neq num[i+1]$, 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 $num[-1] = num[n] = -Œ$.

Example

In array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

Anslysis

可以使用顺序查找的方法，也可以使用二分的方法.
当使用二分的方法时，需要判断mid与$mid+1$的关系.

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

```
class Solution {
public:
    int findPeakElement(const vector<int> &num)
    {
        int low = 0;
        int high = num.size()-1;

        while(low < high)
        {
            int mid1 = (low+high)/2;
            int mid2 = mid1+1;
            if(num[mid1] < num[mid2])
                low = mid2;
            else
                high = mid1;
        }
```

```
            return low;
        }
};
```

Sequential Search : $\boxed{\text{时间复杂度O}(n)?\text{，空间复杂度O}(1)}$

在使用顺序查找的方法时，不需要比较$a[i]a[i-1]a[i+1]$的关系，只需检测当前的值是否大于前一位置的值，如果大于，则继续寻找，表明现在处于升序阶段；如果小于，则可以确定这里是降序阶段的起始位置．

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

4.3.5 Minimum Size Subarray Sum

Question:

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the $sum \geq s$. If there isn't one, return 0 instead.

Example

For example, given the array [2,3,1,2,4,3] and s = 7,
the subarray [4,3] has the minimal length under the problem constraint.

Anslysis

这里的subarray代表连续的子序列．我们可以观察在加法过程中的重复计算的元素并加以使用．
本题目可以使用二分来求解．虽然数组是无序的，但是因为数组内没有负数，所以从0到i的和是递增的．满足二分查找的要求．

Solution

Sequential Search : $\boxed{\text{时间复杂度O}(n)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        int minSubArrayLen(int s, vector<int>& nums) {
            len = nums.size();

            int sum=0;
            int minLen = -1;
            int start;
            for(int i=0;i<len;++i){
                sum += nums[i];
                if(sum >= s){//the sum from 0 to i
                    start = i;
                    minLen = i - 0 + 1;
                    break;
                }
            }
            if(minLen == -1)    return 0;

            bool isUpdate = false;
            for(int i=1;i<len;++i){
                sum -= nums[i-1];
                if(sum >= s){
                    isUpdate = true;
                    minLen = min(minLen,start-i+1);
                }else{
                    for(int j=start+1;j<len;++j){
                        sum += nums[j];
                        if(sum >= s){
                            isUpdate = true;
                            start = j;
                            minLen = min(minLen, j-i+1);
                            break;
                        }
                    }
                }
                if(!isUpdate)
                    break;
            }
            return minLen;
        }
};
```

Binary Search : 时间复杂度O($n * lgn$)，空间复杂度O(1)

```
    private int solveNLogN(int s, int[] nums) {
        int[] sums = new int[nums.length + 1];
        for (int i = 1; i < sums.length; i++) sums[i] = sums[i - 1] +
            nums[i - 1];
        int minLen = Integer.MAX_VALUE;
        for (int i = 0; i < sums.length; i++) {
            int end = binarySearch(i + 1, sums.length - 1, sums[i] + s,
                sums);
            if (end == sums.length) break;
            if (end - i < minLen) minLen = end - i;
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    private int binarySearch(int lo, int hi, int key, int[] sums) {
        while (lo <= hi) {
           int mid = (lo + hi) / 2;
           if (sums[mid] >= key){
               hi = mid - 1;
           } else {
               lo = mid + 1;
           }
        }
        return lo;
    }
```

4.3.6 Pow

Question:

Implement pow(x, n).

Anslysis

要分析x和n可能的取值情况. $x = 0$,$n = 0$, $x < 0$, $n < 0$,n的奇偶性质
需要特别注意的是：在对某个负数取正时，必须要考虑到复数的取值范围大于正数!
本题也可以遍历 n 的0 31位上的 0 ／ 1 情况.

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

```
class Solution {
    public:
        double myPow(double x, int n) {
            if(abs(x) < 1e-9)   return 0;
            if(n == 0)  return 1;

            bool isRo = false;
            long long newN = n;
            if(n < 0){//注意为负数的情况n
                //Oh..no.. n==INT_MIN
                isRo = true;
                newN = -1 * newN;
            }

            double res1 =x,res2 = 1;
            while(newN > 1){
                if(newN % 2 == 0){
                    res1 *= res1;
                    newN = newN/2;
                }else{
                    res2 *= res1;
                    newN -= 1;
                }
            }
            if(isRo)
                return (1/(res1*res2));
            else
                return res1*res2;
        }
};
```

Binary Search : $\boxed{\text{时间复杂度O}(lgn)?\text{，空间复杂度O}(1)}$

遍历n的 0 ～ 3 1 位上的0/1情况.
但是，这道题目的x的取值为double类型，无法使用移位操作，所以这种方法不可行.

4.3.7 Range Search

Question:

Given a sorted array of integers, find the starting and ending position of a given target value.
Your algorithm's runtime complexity must be in the order of O(log n).
If the target is not found in the array, return [-1, -1].

Example

Given [5, 7, 7, 8, 8, 10] and target value 8,
return [3, 4].

Anslysis

本题目是lower_bound和upper_bound.
在写二分算法时，注意while中条件的控制

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{ , 空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        int lowerBoud(vector<int>&nums,int target){
            int left=0,right=len-1;
            while(left <= right){
                int mid = left + (right - left) /2;
                if(nums[mid] < target)
                    left = mid + 1;
                else if(nums[mid] > target)
                    right = mid - 1;
                else{
                    if(mid==0 || nums[mid-1]!=target)
                        return mid;
                    else
                        right = mid -1;
                }
            }
            return -1;
        }
        int upperBound(vector<int>&nums,int target){
            int left=0,right=len-1;
            while(left <= right){
                int mid = left + (right - left) /2;
                if(nums[mid] > target)
                    right = mid - 1;
                else if(nums[mid] < target)
                    left = mid + 1;
                else{
                    if(mid == len-1 || nums[mid+1]!=target)
                        return mid;
                    else
                        left = mid + 1;
                }
            }
```

```
                return -1;
            }
            vector<int> searchRange(vector<int>& nums, int target) {
                len = nums.size();
                if(len == 0)    return vector<int>(2,-1);
                int low = lowerBoud(nums,target);
                int upp = upperBound(nums,target);
                vector<int> res;
                res.push_back(low); res.push_back(upp);
                return res;
            }
};;
```

Binary Search : $\boxed{\text{时间复杂度O}(lgn)?\text{，空间复杂度O}(1)}$
遍历n的 0 ∼ 3 1 位上的0/1情况.
但是，这道题目的x的取值为double类型，无法使用移位操作，所以这种方法不可行.

4.3.8 Search 3D Matrix

Question:
Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:
Integers in each row are sorted from left to right.
The first integer of each row is greater than the last integer of the previous row.

Anslysis
因为矩阵满足按序排列的性质，所以可使用二分查找.

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lg(m*n))\text{，空间复杂度O}(1)}$

```
class Solution {
    public:
        bool searchMatrix(vector<vector<int> >& matrix, int target) {
            int col = matrix.size();
            if(col==0)return false;
            int row = matrix[0].size();

            int top=0,bottom=col-1;
            while(top < bottom){
                int mid = top + (bottom-top)/2;
                if(matrix[mid][row-1]==target)
                    return true;
                else if(matrix[mid][row-1] > target)
                    bottom = mid;
                else
                    top=mid+1;
            }
            int level = top;
            top=0,bottom=row-1;
            while(top <= bottom){
                int mid = top + (bottom - top) / 2;
                if(matrix[level][mid] == target)
                    return true;
                else if(matrix[level][mid] > target)
                    bottom = mid - 1;
                else
                    top = mid + 1;
            }
            return false;
        }
};
```

Use STL : $\boxed{\text{时间复杂度O}(lg(m*n))\text{，空间复杂度O}(1)}$

```
        vector<int> searchRange(vector<int>& nums, int target) {
            pair<vector<int>::iterator, vector<int>::iterator > bounds;
            bounds = std::equal_range(nums.begin(), nums.end(), target);
            if(bounds.first == nums.end() || *(bounds.first) != target)
                return vector<int>(2,-1);
            vector<int> res;
            res.push_back(bounds.first - nums.begin()); res.push_back(
                bounds.second - nums.begin() -1);
            return res;
        }
        vector<int> searchRange(vector<int>& nums, int target) {
            vector<int>::iterator lo = std::lower_bound(nums.begin(),
                nums.end(),target);//first elem not less than target
            vector<int>::iterator up = std::upper_bound(nums.begin(),
                nums.end(),target);//first elem great than target
            if(lo == nums.end() || *lo != target)
                return vector<int>(2,-1);
            vector<int> res;
            res.push_back(lo - nums.begin());   res.push_back(up - nums.
                begin() -1);
            return res;
        }
```

4.3.9 Search In Roated Sorted Array

Question:

Suppose a sorted array is rotated at some pivot unknown to you beforehand.
You are given a target value to search. If found in the array return its index, otherwise return -1.
ou may assume no duplicate exists in the array.

Anslysis

与Find-Minimum-in-Rotated-Sorted-Array.tex类似.
在二分时，可以关注target在特定区间的存在性来判断left，right的移动方式

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

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

                if(nums[mid] >= nums[left]){
                    if(target >= nums[left] && nums[mid] > target)
                        right = mid - 1;
                    else
                        left = mid + 1;
                }else{
                    if(target <= nums[right] && nums[mid] < target)
                        left = mid + 1;
                    else
                        right = mid - 1;
                }
            }
            return -1;
        }

};
```

4.3.10 Search In Roated Sorted ArrayII

Question:

Follow up for "Search in Rotated Sorted Array":
What if duplicates are allowed?

Anslysis

类似于Search-In-Rotated-Sorted-Array
在二分时，可以关注target在特定区间的存在性来判断left，right的移动方式

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

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

};
```

4.3.11 Search Insertion Position

Question:

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You may assume no duplicates in the array.

Anslysis

待处理数组为已排序状态，可以使用二分查找的方法.

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{，空间复杂度O}(1)}$

```
class Solution {
    int len;
    public:
        int searchInsert(vector<int>& nums, int target) {
            len = nums.size();
            int left=0,right=len-1;
            while(left <= right){
                int mid = left + (right - left) /2;
                if(nums[mid] == target) return mid;
                if(nums[mid] > target){
                    right = mid - 1;
                }else{
                    left = mid + 1;
                }
            }
            //not found, now left
            return left;
        }
};
```

4.3.12 Sqrt

Question:

Implement int sqrt(int x).
Compute and return the square root of x.

Anslysis

我们可以使用类似二分的方法来寻找结果．小心待处理的数太大造成溢出的问题.

Solution

Binary Search : $\boxed{\text{时间复杂度O}(lgn)\text{ , 空间复杂度O}(1)}$

```
class Solution {
    public:
        int mySqrt(int x) {
            long long newX = x;
            long long left=0,right=newX;
            long long multiply;
            while(left <= right){
                long long mid = left + (right - left)/2;
                multiply = mid * mid;
                if(multiply == newX)    return mid;
                if(multiply > newX)
                    right = mid - 1;
                else
                    left = mid + 1;
            }
            if(left * left == x)
                return left;
            return left - 1;
        }
};
```

Sequential Search : $\Box$

本题目不能采用遍历比特位的方法．因为在比例比特位的方法中，我们的思路是$(a+b)^2 = a^2+2ab+b^2$,但是中间的$2ab$无法处理.