**Algorithm Notes**

**Hints1: 不使用记忆做题，应使用知识体系、分析能力、逻辑能力做题。**

**Hints2:有多个if,else时，把可能情况多的选项放在前面，减少判断。**

目录

[1 Array 1](#_Toc3102050)

[1.1 Basic 1](#_Toc3102051)

[229 Majority Element II 1](#_Toc3102052)

[1.2 HashTable 2](#_Toc3102053)

[1 Two Sum 2](#_Toc3102054)

[1.3 Two Pointer 2](#_Toc3102055)

[11 container with most water 3](#_Toc3102056)

[15 3 sum 3](#_Toc3102057)

[16 3 sum closest 4](#_Toc3102058)

[18 4 sum 4](#_Toc3102059)

[209 Minimum size subarray sum 4](#_Toc3102060)

[1.4 backtracking 5](#_Toc3102061)

[39 combination sum 5](#_Toc3102062)

[40 Combination sum II 6](#_Toc3102063)

[1.5 Binary Search 6](#_Toc3102064)

[4 Median of two sorted array 6](#_Toc3102065)

[153 find minimum in rotated sorted array 7](#_Toc3102066)

[154 find minimum in rotated sorted array 8](#_Toc3102067)

[209 Minimum size subarray sum 8](#_Toc3102068)

[454 4sum II 10](#_Toc3102069)

# 1 Array

## Basic

### 31 Next Permutation

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 sorted in ascending order.

The replacement must be [**in-place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

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

核心思想：重点是要弄清楚排列组合的性质，正好比当前排列大一点的排列（或者叫下一个排列）即使从后往前找到nums[i]>nums[i-1]，然后将nums[i-1]插入到后面的对应位置，将后面数列正好大于nums[i-1]的数放到nums[i-1]原来的位置。

核心代码：

vector<int>::iterator i=nums.end()-1;

int tmp=\*i;

vector<int>::iterator loc,it;

while(i!=nums.begin()){

if(tmp<=\*(i-1)){

tmp = \*(i-1);

nums.erase(i-1);

nums.push\_back(tmp);

i=i-1;

}

else{

tmp = \*(i-1);

it = loc = nums.erase(i-1);

while(it!=nums.end()){

if(\*it>tmp){

nums.insert(it,tmp);

tmp = \*(it+1);

nums.erase(it+1);

nums.insert(loc,tmp);

return;

}

it = it+1;

}

}

}

### 229 Majority Element II

Given an integer array of size *n*, find all elements that appear more than ⌊ n/3 ⌋ times.

**Note:**The algorithm should run in linear time and in O(1) space.

核心思想：使用Boyer Moore majority vote算法，可能有几个满足条件的结果，就用几个标记量，**需要注意**，**遍历完成后剩下的标记量不一定满足条件，可能正好是最后插进来的数，所以需要再遍历一次，判断其出现的次数。**

核心代码：

int a,b,num\_a=0,num\_b=0;

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

if(num\_a!=0&&nums[i]==a)num\_a++;

else if(num\_b!=0&&nums[i]==b)num\_b++;

else if(num\_a==0){a=nums[i];num\_a=1;}

else if(num\_b==0){b=nums[i];num\_b=1;}

else{num\_a--;num\_b--;}

}

vector<int> res;

if(num\_a!=0&&ct(nums,a)>nums.size()/3)res.push\_back(a);

if(num\_b!=0&&ct(nums,b)>nums.size()/3)res.push\_back(b);

## 1.2 HashTable

利用hash数据结构记录数组中元素的出现，然后对相应的元素直接检查获取。

### 1 Two Sum

通过hash表记录各个元素的出现，然后在遍历元素的过程中，检查目标和与该元素的差是否出现，**值得注意的是：相对于完整的遍历存储好再检查，可以边遍历边检查。**

## 1.3 Two Pointer

在**有序**的前提下寻找符合特定条件的两个元素时，相对于暴力的两层遍历叠加，可以使用一头一尾两个指针，分别从前后逼近查找。

**核心代码：**

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

int res = target-nums[i];

if(num\_map.find(res)!=num\_map.end()){

result[0] = num\_map[res];

result[1] = i;

return result;

}else{

num\_map[nums[i]] = i;

}

}

### 11 container with most water

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*).  Find two lines, which together with x-axis forms a container, such that the container contains the most water.

**Example:**

**Input:** [1,8,6,2,5,4,8,3,7]

**Output:** 49

核心代码：

int maxArea(vector<int>& height) {

int l = 0, r = height.size()-1;

int max\_area = min(height[l],height[r])\*(r-l);

while(l<r){

if(height[l]<height[r])l++;

else r--;

max\_area = max(max\_area,min(height[l],height[r])\*(r-l));

}

return max\_area;

}

思路：

两个指针分别从两侧往中间寻找，每次移动**较小**的值对应的指针。为什么较小？嘻嘻

### 15 3 sum

三个数的求和，指定其中一个（依序便利），对另外两个数使用两个指针

核心代码：

while(l<r){

if(nums[l]+nums[r]<-nums[i])l++;

else if(nums[l]+nums[r]>-nums[i])r--;

else{

tmp[0] = nums[i];

tmp[1] = nums[l];

tmp[2] = nums[r];

res.push\_back(tmp);

while(l<r && nums[l]==tmp[1])l++;

while(l<r && nums[r]==tmp[2])r--;

}

}

### 16 3 sum closest

同15，不同的地方在于最近的元素需要记录最小的差

### 18 4 sum

同15，只是外面有两层遍历，相应的需要两层去重

if(i!=0 && nums[i]==nums[i-1])continue;

…

if(j!=i+1&&nums[j]==nums[j-1])continue;

### 209 Minimum size subarray sum

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum ≥ **s**.

不同于上面几个算法，两个指针分别从左从右搜索，此算法中的序列因为无序，所以不能如此，只能一个指针对序列遍历，一个指针对已遍历的序列进行缩减找更小的长度，思路既是：先找到满足条件的，再对满足条件的进行缩减。

int minSubArrayLen(int s, vector<int>& nums) {

int ans = INT\_MAX;

int left = 0, sum = 0;

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

sum+=nums[i];

while(sum>=s){

ans = min(ans,i-left+1);

sum-=nums[left++];

}

return ans==INT\_MAX?0:ans;

}

## 1.4 backtracking

对每个可能的分支遍历时，达到一定条件便回溯到之前的位置继续遍历其他分支。值得注意的是：为了节省空间和时间，结果集可以在遍历的过程中通过函数参数传递从而达到只使用一个结果集的目的。

**关键点：**

* **参数传递**
* **有进有出**

### 39 combination sum

之所以使用回溯是因为允许加和的元素没有限制，所以需要放开空间寻找，达到条件再回溯。

**Input:** candidates = [2,3,6,7], target = 7,

**A solution set is:**

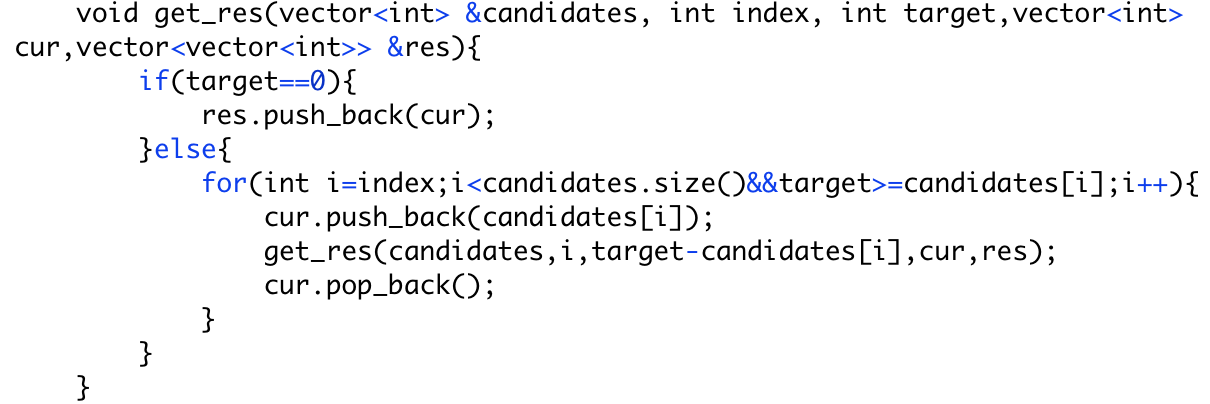
[

[7],

[2,2,3]

]

核心代码：



**注意：**

* **cur有push\_back也有pop\_back, 满足回溯要求的有进有出**
* **为了减少不必要的操作（如减去大于target的candidates元素），可以提前判断（在循环中），减少时间复杂度。**

### 40 Combination sum II

同39

## 1.5 Binary Search

在search过程中，将待search的数列两两分开处理，减少时间复杂度。

### 4 Median of two sorted array

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

**Example 1:**

nums1 = [1, 3]

nums2 = [2]

The median is 2.0

如何使用二分搜索呢，因为有序，可以理解为两个有序数列的中位数会把两个数列分别分为左半部分和右半部分，各自的左半部分的长度和等于右半部分的长度和，且**左半部分的最大值大于右半部分的最小值（又因为数列是有序的，所以只用交叉比较）**，那么如何找到切分两个数列的点，就需要用到二分搜索，这里的二分不是针对于数列的长度，而是中位数（也可以是其他顺序统计量，实际上除了第一次是中位数，后续查找时都不是中位数）的二分，中位数二分为数列1的左半部分和数列2的左半部分，然后比较临界值是否满足上述标黑的性质，根据不同情况截取掉肯定不包含中位数（顺序统计量的）的部分，然后在剩余的数列中继续寻找，注意中位数要减去截取掉的数列的长度，变成一般的顺序统计量。

核心代码：

double get\_median(vector<int> &nums1,vector<int> &nums2,int l,int r,int k,int flag){

if(nums1.size()-l>nums2.size()-r)

if(l==nums1.size()){

}else if(k==1){

}else{

int lk = min(int(nums1.size()-l),k/2);

int res\_k = k-lk;

if(nums1[l+lk-1]<nums2[r+res\_k-1])

return get\_median(nums1,nums2,l+lk,r,k-lk,flag);

else

return get\_median(nums1,nums2,l,r+res\_k,k-res\_k,flag);

}

return 0.0;

}

### 153 find minimum in rotated sorted array

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

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

Find the minimum element.

思路：因为有序，肯定可以使用二分法，关键在于如何处理旋转后尾与头相连的情况，通过观察得，当一个数列增序，**尾元素肯定大于头元素**，反之不成立，而一旦旋转，最小值肯定在非有序的一边（除非正好切分在尾部），所以可以通过标黑的性质逐渐缩减数列的范围，直到数列完全有序或只剩下两个以下元素。

核心代码：

if(nums.size()==1)return nums[0];

int l=0,r=nums.size()-1,mid;

while(l<=r){

if(l>=r-1)return min(nums[l],nums[r]);

if(nums[l]<nums[r])return nums[l];

mid = (l+r+1)/2;

if(nums[l]<nums[mid])l=mid+1;

else r=mid;

}

### 154 find minimum in rotated sorted array

同153，只是会包含相同的元素，处理思路不变，只是相同的元素在鉴别时麻烦，需要额外的时间复杂度或者空间复杂度。

*有时在情况变得复杂时，算法也许必然变得复杂，不要硬磕。*

### 209 Minimum size subarray sum

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum ≥ **s**.

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

**Output:** 2

此题可以利用二分搜索的点有两个：

1. 确定滑动窗口的起点(遍历过程O(n))，对滑动窗口的终点进行二分搜索O(log(n))，值得注意，由于二分搜索要求有序，所以搜索的对象是累和，需要额外空间。
2. 对满足条件的窗口长度二分，也就是从1到nums数列长度，二分获取待验证的窗口长度O(log(n))，然后对该窗口长度验证是否有满足条件的子序列（O(n)）

思路1核心代码：

for(int i=0;i<nums.size();i++){sums.push\_back(sums[i]+nums[i]);}

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

vector<int>::iterator tmp = lower\_bound(sums.begin(),sums.end(),sums[i]+s);

if(tmp!=sums.end())ans = min(ans, int(tmp-sums.begin())-i);

}

思路2核心代码：

int l = 1, r = nums.size(),mid;

while(l<=r){

mid = (l+r)/2;

if(find\_subarray(mid, s, nums)){

ans = mid;

r = mid-1;

}else l = mid+1;

}

### 454 4sum II

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

**Input:**

A = [ 1, 2]

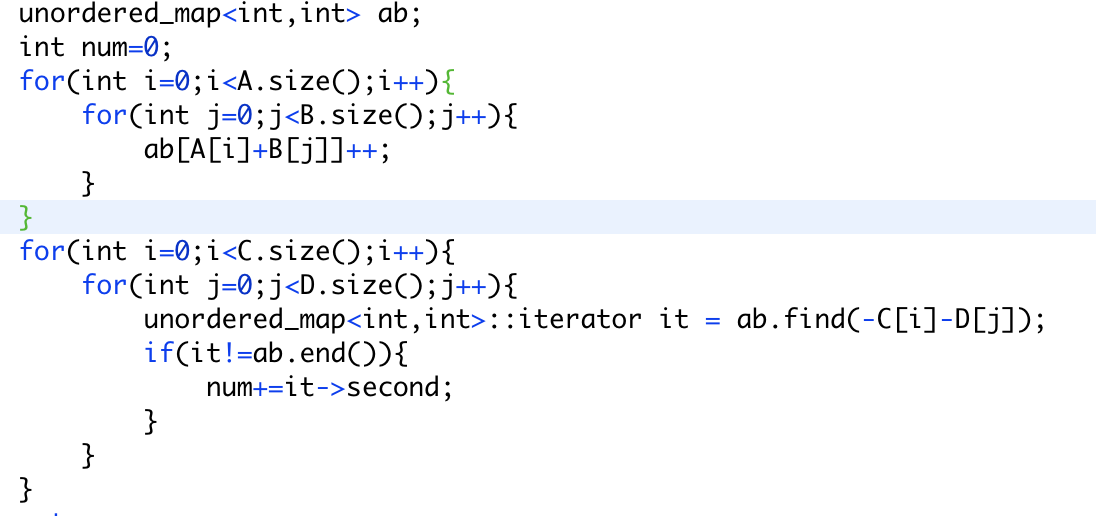
B = [-2,-1]

C = [-1, 2]

D = [ 0, 2]

不同于一般的binary search是针对于一个数列，这个是针对于多个数列，二分的含义从将一个数列分为两个数列处理变为将多个数列划分为两堆数列，依次划分到只剩两个，然后统计结果。当然，在计算的过程中，是自底向上的。

当然，这个题目要求加和为0，核心要加和，所以在统计到只有两个数列时，要两两加和，并标记加和后结果的出现次数。



**PS:** 1 不需要有序的前提下，使用unordered\_map比map快很多

2 学会利用map.find已查找的iterator，不要重复查找