**数组中除每位外的乘积**

不能使用除法

vector<int> productExceptSelf(vector<int>& nums) {

int n = nums.size();

vector<int> pro(n, 1);

for (int i = 1; i < n; i++)

pro[i] = pro[i - 1] \* nums[i - 1];

int right = 1;

for (int i = n - 1; i >= 0; i--)

{

pro[i] \*= right;

right \*= nums[i];

}

return pro;

}

**数组中不相邻数值最大和**

int rob(vector<int>& nums) {

if (nums.size() <= 1)

return nums.empty() ? 0 : nums[0];

int preTwo = nums[0];

int pre = max(nums[0], nums[1]);

int res = pre;

for (int i = 2; i < nums.size(); i++)

{

res = max(nums[i] + preTwo, pre);

preTwo = pre;

pre = res;

}

return res;

}

**调整数组顺序使奇数位于偶数前面**

**且奇数与奇数，偶数与偶数的相对位置不变**

void reOrderArray(vector<int> &array) {

vector<int> temp;

int len = 0;

for (auto it : array)

{

if (it & 1)

{

temp.insert(temp.begin() + len, it);

len++;

}

else

temp.push\_back(it);

}

array.swap(temp);

}

**把数组中0移动数组最后**

**并保持非0元素位置不变**

void moveZeroes(vector<int>& nums) {

int k = 0;

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

if (nums[i] == 0)

k++;

else

swap(nums[i - k], nums[i]);

}

}

**旋转数组的最小数字**

int minNumberInRotateArray(vector<int> rotateArray) {

int low = 0, high = rotateArray.size() - 1;

while (low < high) {

if (rotateArray[low] < rotateArray[high])

return rotateArray[low];

int mid = low + (high - low) / 2;

if (rotateArray[mid] >= rotateArray[low])

low = mid + 1;

else

high = mid;

}

return rotateArray[low];

}

**最小（大）的K个数**

红黑树：

struct comp{

bool operator()(int v1, int v2){ return v1>v2; }

};

vector<int> GetLeastNumbers\_Solution(vector<int> input, int k) {

vector<int> res;

if (k < 1 || (int)input.size() < k)

return res;

set<int, comp> s;

for (auto it : input) {

if ((int)s.size() < k)

s.insert(it);

else if (\*(s.begin()) > it) {

s.erase(s.begin());

s.insert(it);

}

}

res.insert(res.begin(), s.begin(), s.end());

return res;

}

快排思想：

int Partition(vector<int>& input, int begin, int end)

{

int low = begin;

int high = end;

int pivot = input[low];

while (low<high)

{

while (low<high&&pivot <= input[high])

high--;

swap(input[low], input[high]);

while (low<high&&pivot >= input[low])

low++;

swap(input[low], input[high]);

}

return low;

}

vector<int> GetLeastNumbers\_Solution(vector<int> input, int k) {

int len = input.size();

if (k<1 || len<k)

return vector<int>();

if (len == k)

return input;

int start = 0;

int end = len - 1;

int index = Partition(input, start, end);

while (index != (k - 1))

{

if (index>k - 1)

{

end = index - 1;

index = Partition(input, start, end);

}

else

{

start = index + 1;

index = Partition(input, start, end);

}

}

vector<int> res(input.begin(), input.begin() + k);

return res;

}

**连续子数组最大和**

int FindGreatestSumOfSubArray(vector<int> array) {

int sum = 0;

int res = 0x80000000;

for (auto it : array) {

if (sum <= 0)

sum = it;

else

sum += it;

if (sum > res)

res = sum;

}

return res;

}

**最大连续子数组乘积**

int maxProduct(vector<int>& nums) {

int Min = nums[0], Max = nums[0];

int max\_val = Max;

for (int i = 1; i<nums.size(); i++)

{

int cur\_prod1 = Max \* nums[i];

int cur\_prod2 = Min \* nums[i];

Max = max(max(cur\_prod1, cur\_prod2), nums[i]);

Min = min(min(cur\_prod1, cur\_prod2), nums[i]);

max\_val = max(Max, max\_val);

}

return max\_val;

}

**数组中只出现一次的两个数字**

void FindNumsAppearOnce(vector<int> data, int\* num1, int \*num2) {

int lastBit = 0;

for (auto it : data)

lastBit ^= it;

lastBit ^= lastBit&(lastBit - 1);

\*num1 = 0;

\*num2 = 0;

for (auto it : data) {

if (lastBit & it)

\*num1 ^= it;

else

\*num2 ^= it;

}

}

**数组中重复的数字**

**N+1的数组，包含1-n之间的数，只有一个数字重复（可以重复多次）**

int findDuplicate(vector<int>& nums) {

int low = 0;

int high = nums.size() - 1;;

while (low<high){

int mid = (low + high) / 2;

int count = 0;

for (int num : nums){

if (num <= mid) count++;

}

if (count>mid) high = mid;

else low = mid + 1;

}

return low;

}

**数组中和为指定值的两个数**

**假设有唯一解，且不能使用一个数两次**

vector<int> twoSum(vector<int>& nums, int target) {

unordered\_map<int, int> numMap;

vector<int> result;

int n = (int)nums.size();

for (int it = 0; it < n; ++it)

{

int anotherNum = target - nums[it];

if (numMap.find(anotherNum) != numMap.end())

{

result.push\_back(numMap[anotherNum]);

result.push\_back(it);

break;

}

numMap[nums[it]] = it;

}

return result;

}

**数组中三个数之和为0**

vector<vector<int>> threeSum(vector<int>& nums) {

int n = nums.size();

vector<vector<int>> result;

sort(nums.begin(), nums.end()); //sort array fitst

for (int i = 0; i < n; ++i)

{

int low = i + 1, high = n - 1;

while (low < high)

{

int lo = nums[low];

int hi = nums[high];

int sum = nums[i] + lo + hi;

if (sum == 0)

{

result.push\_back({ nums[i], lo, hi });

while (nums[low] == lo) //skip the same element

++low;

while (nums[high] == hi) //skip the same element

--high;

}

else if (sum > 0) //forward pointer to middle

--high;

else

++low;

}

while (nums[i] == nums[i + 1]) //skip the same element

++i;

}

return result;

}

**和为指定值的所有组合**

1. 每个数可以用无数次
2. 每个数只能用一次

class Solution {

public:

vector<vector<int>> re;

vector<int> t;

vector<vector<int>> combinationSum(vector<int>& can, int target) {

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

backTrack(can, target, 0);

return re;

}

void backTrack(vector<int>& can, int target, int k)

{

if (target == 0)

{

re.push\_back(t);

return;

}

for (int i = k; i < can.size(); ++i)

{

if (can[i] > target)

break;

t.push\_back(can[i]);

backTrack(can, target - can[i], i);

t.pop\_back();

//如果每个数只能用一次，跳过相同数字

//while (i + 1 < can.size() && can[i + 1] == can[i])

// i++;

}

}

};

**包含最多的水**

int maxArea(vector<int>& height) {

int n = height.size();

int res = 0;

int x = 0, y = n - 1;

while (x < y)

{

int hi = min(height[x], height[y]);

int v = (y - x)\*hi;

res = max(res, v);

while (x<y && height[x] <= hi)

++x;

while (x<y && height[y] <= hi)

--y;

}

return res;

}

**装最多的水**

****

int trap(vector<int>& height) {

int l = 0,

r = height.size() - 1,

level = 0,

water = 0;

while (l < r) {

int lower = height[height[l] < height[r] ? l++ : r--];

level = max(level, lower);

water += level - lower;

}

return water;

}

**删除有序数组中重复的数**

**1.只保留一个**

int removeDuplicates(vector<int>& nums) {

int n = nums.size();

int count = 0;

for (int i = 1; i < n; i++){

if (nums[i] == nums[i – count-1]) count++;

else nums[i - count] = nums[i];

}

return n - count;

}

2.最多保留两个重复的

int removeDuplicates(vector<int>& nums) {

int n = nums.size();

int count = 0;

for (int i = 2; i < n; i++){

if (nums[i] == nums[i – count - 2]) count++;

else nums[i - count] = nums[i];

}

return n - count;

}

**删除数组中指定的元素**

int removeElement(vector<int>& nums, int val) {

int cnt = 0;

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

if (nums[i] == val)

cnt++;

else

nums[i - cnt] = nums[i];

}

return nums.size() - cnt;

}

**合法数独**

bool isValidSudoku(vector<vector<char>>& board) {

int h1[9][9] = { 0 }; //行使用标识

int h2[9][9] = { 0 }; //列使用标识

int h3[9][9] = { 0 }; //正方形使用标识

for (int i = 0; i < 9; i++)

{

for (int j = 0; j < 9; ++j)

{

if (board[i][j] != '.')

{

int m = board[i][j] - '0' - 1;

int k = i / 3 \* 3 + j / 3;

if (h1[i][m] || h2[j][m] || h3[k][m])

return false;

else

h1[i][m] = h2[j][m] = h3[k][m] = 1;

}

}

}

return true;

}

**解数独**

class Solution {

public:

bool flag = false;

bool row[9][9] = { false };

bool col[9][9] = { false };

bool area[9][9] = { false };

void solveSudoku(vector<vector<char>>& board) {

for (int i = 0; i < 9; ++i)

for (int j = 0; j < 9; ++j)

if (board[i][j] != '.')

{

int m = board[i][j] - '0' - 1;

row[i][m] = col[j][m] = area[i / 3 \* 3 + j / 3][m] = true;

}

dfs(board, 0, 0);

}

void dfs(vector<vector<char>>& board, int i, int j)

{

if (flag == true)

return;

if (i >= 9)

{

flag = true;

return;

}

if (board[i][j] != '.')

{

if (j < 8)

dfs(board, i, j + 1); //先访问行

else

dfs(board, i + 1, 0); //如果已经到了一行的末尾，则重新访问下一行的开始

if (flag) return;

}

else

{

int k = 3 \* (i / 3) + j / 3;

for (int v = 0; v < 9; v++)

{

if (!col[j][v] && !row[i][v] && !area[k][v])

{

board[i][j] = v + 1 + '0';

col[j][v] = row[i][v] = area[k][v] = true;

if (j < 8)

dfs(board, i, j + 1);

else

dfs(board, i + 1, 0);

col[j][v] = row[i][v] = area[k][v] = false;

if (flag) return;

}

}

board[i][j] = '.';

}

}

};

**全排列**

1. 没有重复

vector<vector<int>> permute(vector<int>& nums) {

vector<vector<int>> re;

permutation(nums, 0, re);

return re;

}

void permutation(vector<int>& nums, int start, vector<vector<int>> &re)

{

if (start == nums.size())

{

re.push\_back(nums);

return;

}

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

{

swap(nums[i], nums[start]);

permutation(nums, start + 1, re);

swap(nums[i], nums[start]);

}

}

1. 包含重复

vector<vector<int>> permuteUnique(vector<int>& nums) {

vector<vector<int>> re;

vector<int> r;

unordered\_map<int, int> numMap;

for (auto it : nums)

numMap[it]++;

permutation(numMap, nums.size(), r, re);

return re;

}

void permutation(unordered\_map<int, int> &numMap, int start, vector<int>& r, vector<vector<int>> &re)

{

if (start == 0)

{

re.push\_back(r);

return;

}

for (auto &p : numMap)

{

if (p.second <= 0)

continue;

p.second--;

r.push\_back(p.first);

permutation(numMap, start - 1, r, re);

r.pop\_back();

p.second++;

}

}

**字符全排列**

**假设字符全为小写字母，输出按字典序排列**

class Solution {

public:

vector<string> res;

vector<string> Permutation(string str) {

if (str.empty())

return res;

vector<int> hash(26, 0);

for (auto ch : str)

hash[ch - 'a']++;

string t;

helper(t, str.size(), hash);

return res;

}

void helper(string &t, int k, vector<int> &hash) {

if (k == 0) {

res.push\_back(t);

return;

}

for (int i = 0; i < 26; i++) {

if (hash[i] <= 0)

continue;

hash[i]--;

t.push\_back(i + 'a');

helper(t, k - 1, hash);

t.pop\_back();

hash[i]++;

}

}

};

**N选K的组合**

**N个数选k个数的不同集合**

vector<vector<int>> combine(int n, int k) {

vector<vector<int>> res;

vector<int> t(k, 0);

backTrack(t, n, k, 0, res);

return res;

}

void backTrack(vector<int> &t, int n, int k, int j, vector<vector<int>> &res)

{

if (k == 0)

{

res.push\_back(t);

return;

}

for (int i = j; i < n; i++)

{

t[t.size() - k] = i + 1;

backTrack(t, n, k - 1, i + 1, res);

}

}

**数组的子集**

**1.数组中没有重复元素**

**[1,2,3]的所有子集：[[3],[1],[2],[1,2,3],[1,3],[2,3],[1,2],[ ]]**

vector<vector<int>> subsets(vector<int>& nums) {

vector<vector<int>> res;

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

{

vector<int> t(i, 0);

backTrack(t, 0, i, nums.size(), res, nums);

}

return res;

}

void backTrack(vector<int> &t, int start, int k, int n, vector<vector<int>> &res, vector<int>& nums)

{

if (k == 0)

{

res.push\_back(t);

return;

}

for (int i = start; i < n; ++i)

{

t[t.size() - k] = nums[i];

backTrack(t, i + 1, k - 1, n, res, nums);

}

}

**2.数组中有重复元素**

[1,2,2]的所有子集：[[1],[2],[1,2][2,2],[1,2,2],[ ]]

vector<vector<int>> subsetsWithDup(vector<int>& nums) {

vector<vector<int>> res;

sort(nums.begin(), nums.end()); //先排序

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

{

vector<int> t(i);

backTrack(nums, 0, i, t, res);

}

return res;

}

void backTrack(vector<int> &nums, int start, int k, vector<int> &t, vector<vector<int>> &res)

{

if (k == 0)

{

res.push\_back(t);

return;

}

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

{

t[t.size() - k] = nums[i];

backTrack(nums, i + 1, k - 1, t, res);

while (i + 1 < nums.size() && nums[i + 1] == nums[i]) //跳过重复元素

i++;

}

}

**最长连续子序列（LCS）**

int longestConsecutive(vector<int>& nums) {

int res = 0;

unordered\_set<int> s(nums.begin(), nums.end());

for (int val : nums) {

if (s.count(val))

{

s.erase(val);

int pre = val - 1, next = val + 1;

while (s.count(pre)) s.erase(pre--);

while (s.count(next)) s.erase(next++);

res = max(res, next - pre - 1);

}

}

return res;

}

**最长递增子序列（LIS）**

动态规划：对于每一个元素，循环它前面的元素，如果比它小，dp[i]的值更新为dp[i]和dp[j]+1的最大值。

int lengthOfLIS(vector<int>& nums) {

int n = nums.size();

if (n == 0) return 0;

vector<int> dp(n, 1);

int res = 1;

for (int i = 1; i < n; i++) {

for (int j = 0; j < i; j++)

if (nums[j] < nums[i])

dp[i] = max(dp[i], dp[j] + 1);

res = max(res, dp[i]);

}

return res;

}

其他方法：

int lengthOfLIS(vector<int>& nums) {

vector<int> res;

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

auto it = lower\_bound(res.begin(), res.end(), nums[i]);

if (it == res.end())

res.push\_back(nums[i]);

else

\*it = nums[i];

}

return res.size();

}

**按高度重建数组**

**数组元素pair{h,k}，h表示当前人的身高，k表示该人前面身高大于等于h人的个数。重建数组，使得所有元素满足这种要求。**

vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {

vector<pair<int, int>> res;

auto comp = [](const pair<int, int> &p1, const pair<int, int> &p2)

{

return p1.first > p2.first || (p1.first == p2.first && p1.second < p2.second);

};

//先按身高递增排序，如果身高相同，则按k递增排序

sort(people.begin(), people.end(), comp);

//按k的大小插入到对于位置

for (auto it : people)

res.insert(res.begin() + it.second, it);

return res;

}