**实验1**：155.最小栈

设计一个支持 push ，pop ，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) —— 将元素 x 推入栈中。

pop() —— 删除栈顶的元素。

top() —— 获取栈顶元素。

getMin() —— 检索栈中的最小元素。

示例:

输入：

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

输出：

[null,null,null,null,-3,null,0,-2]

解释：

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); --> 返回 -3.

minStack.pop();

minStack.top(); --> 返回 0.

minStack.getMin(); --> 返回 -2.

本题目用到的是老师在课上讲到的方法：辅助栈。即主栈作为存储栈存贮元素、副栈栈顶永远存着当前元素中最小的元素。用一个pair作为一个存储单元，实现辅助栈。

class MinStack {

private:

stack<pair<int, int>> st;

public:

/\*\* initialize your data structure here. \*/

MinStack() {

}

void push(int x) {

if (st.size() == 0) {

st.push({x,x});

}

else {

st.push({ x , min(x,st.top().second) });

}

}

void pop() {

st.pop();

}

int top() {

return st.top().first;

}

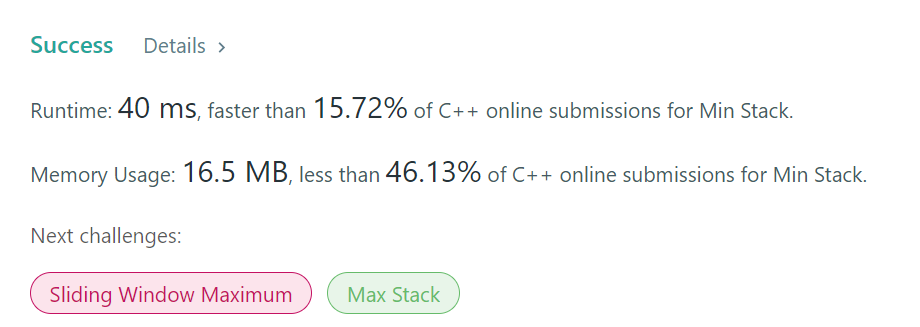
int getMin() {

return st.top().second;

}

};

运行截图：



**实验2**：1497. Check If Array Pairs Are Divisible by k

给你一个整数数组 arr 和一个整数 k ，其中数组长度是偶数，值为 n 。

现在需要把数组恰好分成 n / 2 对，以使每对数字的和都能够被 k 整除。

如果存在这样的分法，请返回 True ；否则，返回 False 。

示例 1：

输入：arr = [1,2,3,4,5,10,6,7,8,9], k = 5

输出：true

解释：划分后的数字对为 (1,9),(2,8),(3,7),(4,6) 以及 (5,10) 。

分析：首先可以直观地想到一个笨办法：设置一个ismatch数组，初始时全部置为false；对于每一个数，从后向前扫描一遍、若遇到满足条件的数，将这两个数字的ismatch置为true。最后扫描ismatch数组中是否全部为true即可。

这个方法确实可以解决问题，但是超时了。代码如下：

bool canArrange(vector<int>& arr, int k) {

bool ismatch[100000]; //max vector length

memset(ismatch, false, sizeof(ismatch));

int i = 0; int j = arr.size();

while (i < arr.size()) {

for (j = arr.size() - 1; j >= 0; j--) {

if ( (arr[i] + arr[j]) % k == 0 && ismatch[i]==false && ismatch[j]==false && i!=j) { ismatch[i] = true; ismatch[j] = true; break; }

}

j = arr.size() - 1;

i++;

}

bool ans = true;

int w = 0;

while (w < arr.size() ) {

if (ismatch[w] == false) { ans = false; }

w++;

}

return ans;

}

然后考虑修改思路：对于每一个数，根据它模k的余数来分类统计个数；例如mod[i]为模k为i的数的个数。只要确保：

1.box[i] 和 box[k - i]相等；注意,当k为偶数时,还要额外判断k/2位置的值是否为偶数个。

2. box[0]的值的个数为偶数个

就满足要求。这种算法只需要一重循环，时间复杂度相比上笨办法由n^2降到了n，满足时间要求。

代码如下：

class Solution {

public:

bool canArrange(vector<int>& arr, int k) {

vector<int> mod(k);

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

mod[(arr[i] % k + k) % k]++;

}

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

if (mod[i] != mod[k - i]) {

return false;

}

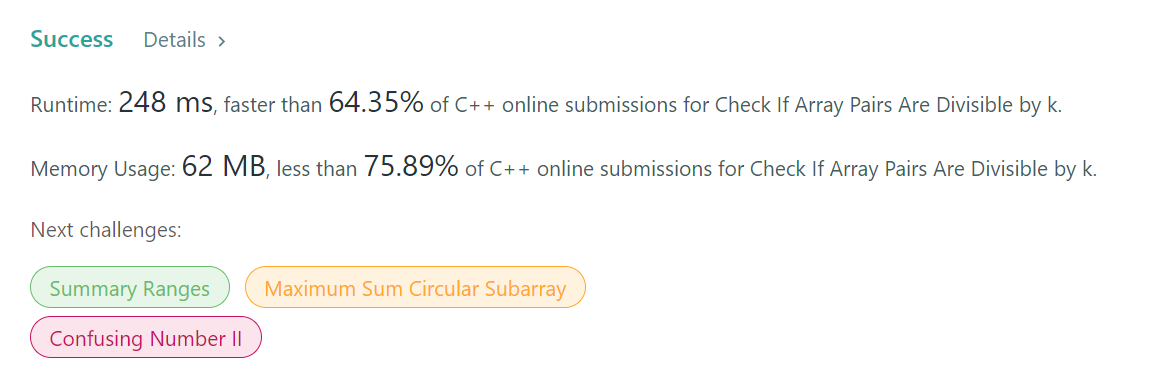
}

return mod[0] % 2 == 0;

}

};

运行截图：



**实验3**：134.Gas Station

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

分析题目要求，我们不难得出：

//1.出发站点，gas[i] > cost[i]

//2.设置一个rest[]; rest[i] = rest[i - 1] + gas[i] - cost[i - 1];

//仔细一想不需要数组，单独变量rest即可。初始时rest = 0。

//3.能走一圈的条件：每一个点都有rest[i] + gas[i] > cost[i]

//4.如何选择一个出发点？设expectation[i] = gas[i] - cost[i];

//找到环形序列的最大子序列和，子序列开头就是出发点！

那么，环形序列的最大子序列和怎么求呢？

这里我参考了老师讲的动态规划法求普通序列的最大子序列；

只要把dp数组的循环范围改为[i..2\*n-1]即可。

举例：现有环形序列1 -3 4 -2；我们考虑普通序列1 -3 4 -2 1 -3 4，即可求得最大子序列为4 -2 1。代码如下：

int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

if (gas.size() == 1) {

return gas[0] >= cost[0]？0：-1;

}

vector<int> expect;

int n = gas.size();

for (int i = 0; i < gas.size(); i++) { // 求环形序列expect的最大子序列

expect.push\_back(gas[i] - cost[i]);

}

int rest = 0;

int dp[200000];

dp[0] = 0;

for (int j = 1; j <=n\*2-1; j++)

dp[j] = max(dp[j-1] + expect[(j - 1 + n) % n], expect[(j - 1 + n) % n]);

int maxj = 1; int k = 0; int j = 0;

for (j = 2; j <= n\*2-1; j++) //求dp中最大元素dp[

if (dp[j] > dp[maxj]) maxj = j;

for (k = maxj; k >= 1; k--) //k为出发点

if (dp[k] <= 0) break;

for ( j = k+1; j < k + n; j++) { // 考虑出发点的下一个点的rest

rest = rest + gas[(j - 1 + n) % n] - cost[(j-1+n)%n ];

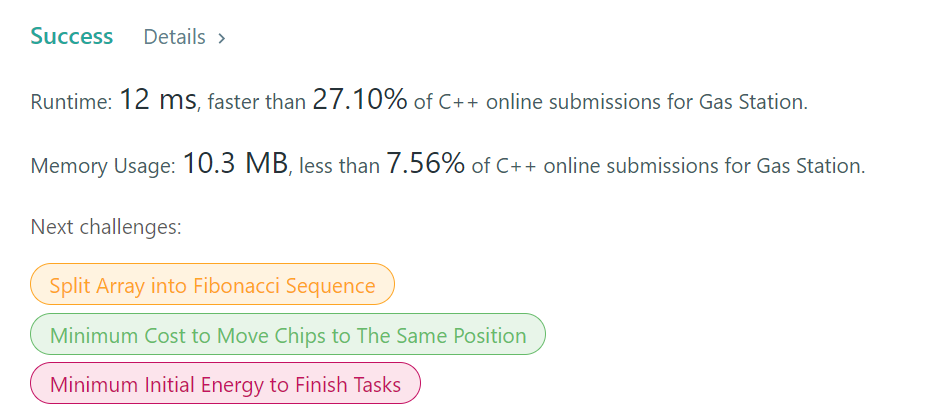
if (rest + gas[j%n] < cost[j%n ]) { return -1; }

}

return k;

}

运行截图：



**实验4**：131.分割回文串

给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。

返回 s 所有可能的分割方案。

示例:

输入: "aab"

输出:

[

["aa","b"],

["a","a","b"]

]

//分析得知，该问题需要找到所有可能的解，且不改变原字符串的顺序，因此为一个子集树回溯法问题。

//每次分割一次目标字符串，得到的结果存进一个vector<string>中；

//每次调用算法得到的vector<string>再存进一个公共的vector<vector<string>>中。

//因此，在题目所给的partition方法中首先定义一个vector<vector<string>> anslist，然后定义回溯方法，回溯方法要添加对anslist的引用；

//然后定义一个vector<string> ans，用于存放单次分割产生的答案

//在回溯方法中定义一个用于存放单个分割结果，若得到结果就将其存入anslist中。

代码如下：

class Solution {

private:

vector<vector<string>> result;

vector<string> path; // 放已经回文的子串

void backtracking(const string& s, int startIndex) {

// 如果起始位置已经大于s的大小，说明已经找到了一组分割方案了

if (startIndex >= s.size()) {

result.push\_back(path);

return;

}

for (int i = startIndex; i < s.size(); i++) {

if (isPalindrome(s, startIndex, i)) { // 是回文子串

// 获取[startIndex,i]在s中的子串

string str = s.substr(startIndex, i - startIndex + 1);

path.push\_back(str);

}

else { // 不是回文，跳过

continue;

}

backtracking(s, i + 1); // 寻找i+1为起始位置的子串

path.pop\_back(); // 回溯过程，弹出本次已经填在的子串

}

}

bool isPalindrome(const string& s, int start, int end) {

for (int i = start, j = end; i < j; i++, j--) {

if (s[i] != s[j]) {

return false;

}

}

return true;

}

public:

vector<vector<string>> partition(string s) {

result.clear();

path.clear();

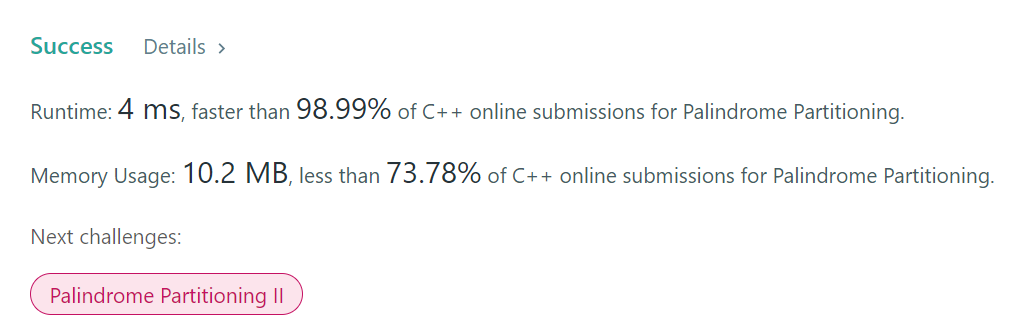
backtracking(s, 0);

return result;

}

};

运行截图：



**实验5**：78.Subsets

Given an integer array nums, return all possible subsets (the power set).

The solution set must not contain duplicate subsets.

求幂集问题，经典的子集树回溯法。其实是因为做难题太打击自信，做一个基础题缓一缓……

靠，其实也不简单，即使已经知道了应该用什么方法，落实到具体的数据结构和问题上还是要想一下。一想一调试好长时间就过去了……

代码如下：

class Solution {

private:

vector<int> res; // 存放单个集合

vector<vector<int>> reslist; // 存放结果

void dfs(vector<int>& nums, int i, vector<bool> x) //n：集合元素个数，也即解空间树层数 i：当前解空间树层数；x[]解向量

{

int n = nums.size();

if (i >= n) {

for (int i = 0; i < n; i++) { //得到了一个幂集

if (x[i] == true) { res.push\_back(nums[i]); }

}

reslist.push\_back(res);

res.clear();

}

else

{

x[i] = false; dfs(nums, i + 1, x); //不选择a[i]

x[i] = true; dfs(nums, i + 1, x); //选择a[i]

}

}

public:

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

int n = nums.size();

vector<bool> x(n);

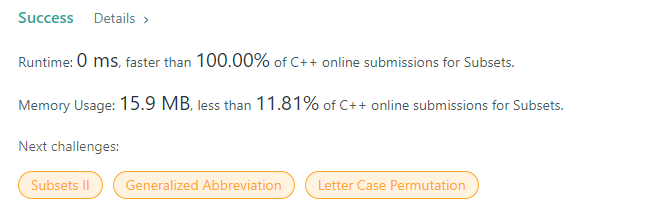
dfs(nums, 0, x);

return reslist;

}

};

第一次faster than 100%，纪念下：



**实验**6：90.Subsets II

Given a collection of integers that might contain duplicates, **nums**, return all possible subsets (the power set).

//如今原集合中可能有重复的元素，但幂集中不能有重复的元素；

//因此在上题解法基础上，在将res加入到reslist前，判断一下有无重复，若有则不加入。

//如何判断重复呢？将两个集合中的元素排序一下，排序后逐一比较，若完全相同，两集合相同。（目前只想到这个笨办法，好在题目的时间复杂度放得很宽……）

代码如下：

class Solution {

private:

vector<int> res; // 存放单个集合

vector<vector<int>> reslist; // 存放结果

bool issameset(vector<int> set1, vector<int> set2) {

sort(set1.begin(),set1.end());

sort(set2.begin(), set2.end());

return set1 == set2;

}

bool contain(vector<vector<int>> list, vector<int> item) {

vector<vector<int>>::iterator it=list.begin();

for (it; it < list.end(); it++) {

if (issameset(\*it,item)) { return true; }

}

return false;

}

void dfs(vector<int>& nums, int i, vector<bool> x) //n：集合元素个数，也即解空间树层数 i：当前解空间树层数；x[]解向量

{

int n = nums.size();

if (i >= n) {

for (int i = 0; i < n; i++) { //得到了一个幂集

if (x[i] == true) { res.push\_back(nums[i]); }

}

if(!contain(reslist,res)){

reslist.push\_back(res);

}

res.clear();

}

else

{

x[i] = false; dfs(nums, i + 1, x); //不选择a[i]

x[i] = true; dfs(nums, i + 1, x); //选择a[i]

}

}

public:

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

int n = nums.size();

vector<bool> x(n);

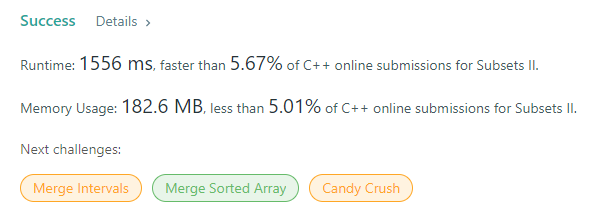
dfs(nums, 0, x);

return reslist;

}

};

结果也是意料之内的很慢：



**实验7**：77.组合

//给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。

//使用回溯法 - 子集树来解决该问题。

//一个问题有很多步、每一步有不同的选择、又要求出问题的所有解，就可以使用回溯法。

//现在来看组合问题：从1234中找出所有组合，应当是：12 13 14 23 24 34

//也就是说，包含i的组合，只需要在[i..end]中找即可

//此处有一个细节：必须先记录合法答案，再判断是否超出范围；否则会丢失【i，n】这个解。比如（2，3），就会丢失[1，3]，[2，3]

代码如下：

class Solution {

public:

vector<int> temp;

vector<vector<int>> ans;

void dfs(int cur, int n, int k) {

// 记录合法的答案

if (temp.size() == k) {

ans.push\_back(temp);

return;

}

//超出范围

if (cur > n) {

return;

}

// 考虑选择当前位置

temp.push\_back(cur);

dfs(cur + 1, n, k);

temp.pop\_back();

// 考虑不选择当前位置

dfs(cur + 1, n, k);

}

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

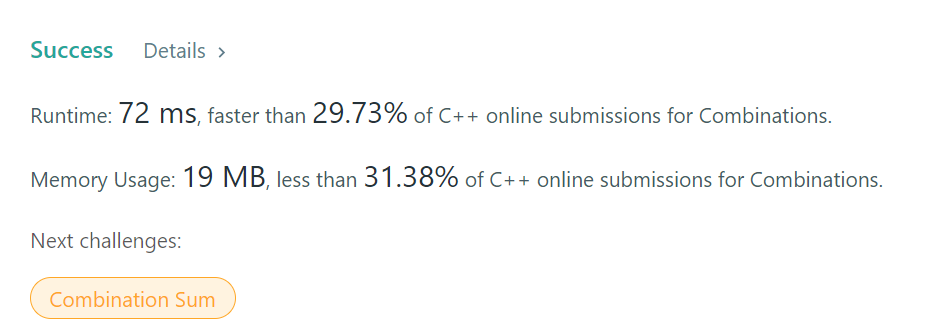
dfs(1, n, k);

return ans;

}

};

运行截图：



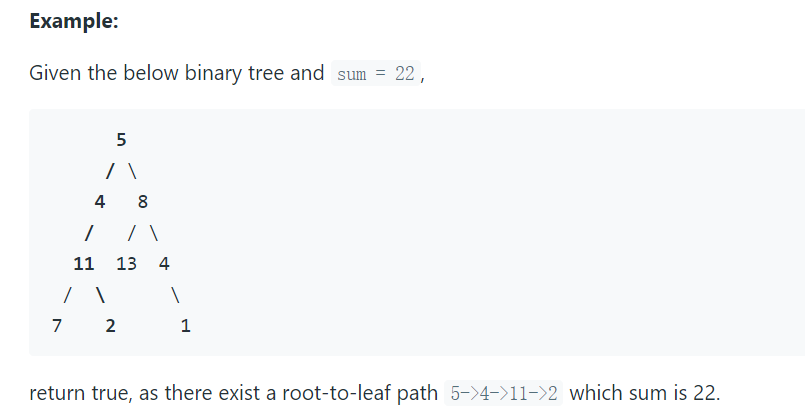
**实验8**：112.路径总和

//给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

//

//说明 : 叶子节点是指没有子节点的节点。

图例：



//本题目需要遍历所有结点；题目所给的二叉树就是解空间树，很显然可以用回溯法解决。

//考虑某一结点的子节点时，将sum=sum-当前结点val即可。

//到叶子节点时，只需要判断叶子节点值是否等于传入的sum。

//先判断左子树；若左子树无解，再判断右子树，体现回溯思想。

代码如下：

//Definition for a binary tree node.

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode() : val(0), left(nullptr), right(nullptr) {}

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

TreeNode(int x, TreeNode\* left, TreeNode\* right) : val(x), left(left), right(right) {}

};

class Solution {

private:

int total = 0;

public:

bool hasPathSum(TreeNode\* root, int sum) {

if (root == NULL) {

cout << "";

return false;

}

if (root->left == NULL && root->right == NULL) { //到达叶子节点

if (root->val == sum) {

return true;

}

return false;

}

//total += root->val;

//int left = (root->left == NULL) ? 0 : root->left->val;

//int right = (root->right == NULL) ? 0 : root->right->val;//保存左右结点的值

if (hasPathSum(root->left, sum - root->val) == true) {

return true;

}

else {

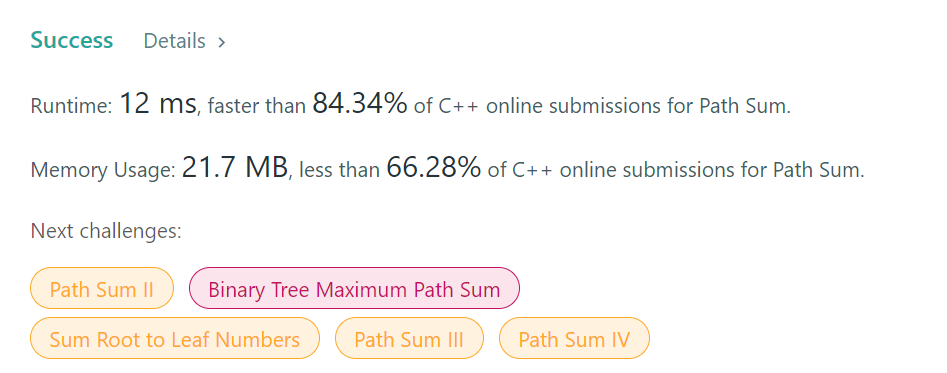
return hasPathSum(root->right, sum - root->val);

}

}

};

运行截图：



**实验9**：46.全排列

//给定一个 没有重复 数字的序列，返回其所有可能的全排列。

//

//一个经典的排列树问题。做点基础题找找自信……

//根据回溯法.排列树框架：

//最初的排列开始，【第i层代表处理第i个元素】；

//几个子树分别代表：和a【i】交换（和自己交换就是不交换）。所以排列中有几个元素，第一层就有几个分支；

//第i层要处理的第i个元素，都是和从i开始（也就是不换）往后的元素换；每一个共父结点群，第一个节点都和父亲节点一样！

//后面每下一层少一个分支。（第i个元素不和第i-1个元素换，防止换回去，即回到上一层的状态）

//来个框架：

//void dfs(int a[], int n, int i) //求a[0..n-1]的全排列

//{

// if (i >= n) //递归出口

// dispasolution(a, n);

// else

// {

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

// {

// swap(a[i], a[j]); //交换a[i]与a[j]:每一个元素只与自己及其后面的元素交换

// dfs(a, n, i + 1);

// swap(a[i], a[j]); //交换a[i]与a[j]：若没有找到解，恢复

// }

// }

//}

class Solution {

private:

vector<int> ans;

vector<vector<int>> anslist;

void dfs(vector<int>& nums, int i) { //i从0开始

if (i >= nums.size()) {

ans = nums;

anslist.push\_back(ans);

}

else {

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

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

dfs(nums, i + 1);

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

}

}

}

public:

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

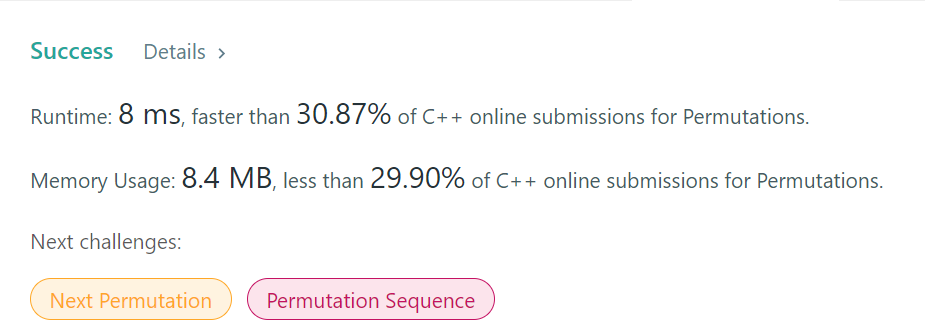
dfs(nums, 0);

return anslist;

}

};

运行截图：



**实验室10**：47.全排列II

//给定一个可包含重复数字的序列 nums ，按任意顺序 返回所有不重复的全排列。

//输入：nums = [1, 1, 2]

//输出：

//[[1, 1, 2],

//[1, 2, 1],

//[2, 1, 1]]

//

//和排列类似，但在加入时需要判读：anslist中是否已经包含该ans；

//如何判断重复呢？将两个集合中的元素逐一比较，若完全相同，两集合相同（老笨蛋了）

//不难得到代码：

class Solution {

private:

vector<int> ans;

vector<vector<int>> anslist;

bool contain(vector<vector<int>> list, vector<int> item) {

vector<vector<int>>::iterator it = list.begin();

for (it; it < list.end(); it++) {

if (\*it == item) { return true; }

}

return false;

}

void dfs(vector<int>& nums, int i) { //i从0开始

if (i >= nums.size()) {

ans = nums;

if (!contain(anslist, ans)) {

anslist.push\_back(ans);

}

}

else {

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

if (nums[i] != nums[j]) {

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

dfs(nums, i + 1);

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

}

}

}

}

public:

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

dfs(nums, 0);

return anslist;

}

};

//然而这么做超时了……事实上排列树的时间复杂度本身很高，这个算法已经无法在时间复杂度上进一步优化了。

//因此我们可以使用传统的回溯法，将排列过程想象成：有n个空格，现在我们要往这n个空格里面填入数字；

//我们首先对序列进行sort，保证相同的数字是相邻的；

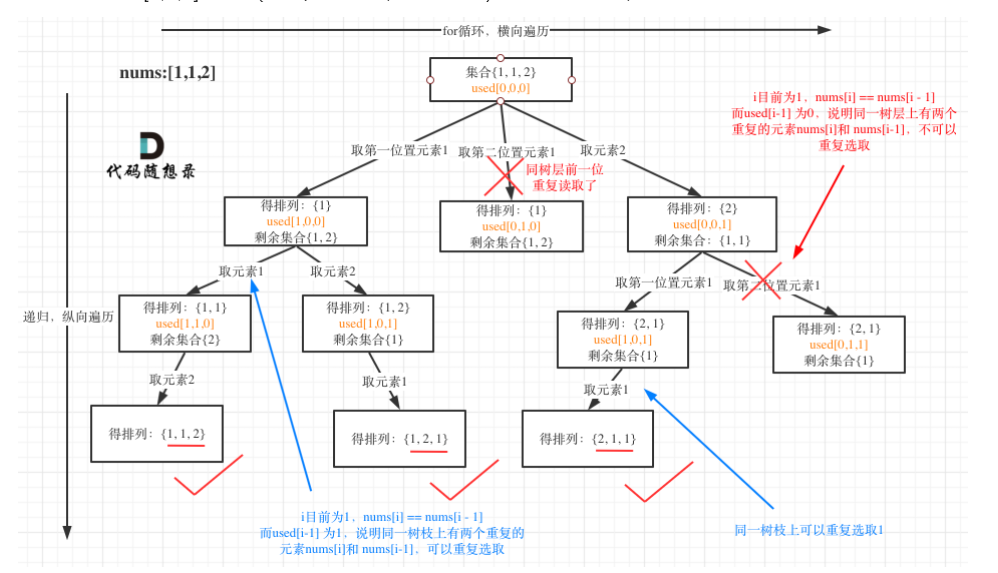
//每次填入数字时，可以检测相同集里手否已经有数字被填入过，如果有则可以剪枝。

//例如有序列：1 1 2；

//在解空间树的第一层第一个结点，我们填1；

//该层的第二个结点应该填i[1]，发现i[1]=i[0]且vis[0]=0,则进行剪枝，不填！

图解：



//由此可以得到代码：

class Solution {

vector<int> vis;

public:

void backtrack(vector<int>& nums, vector<vector<int>>& anslist, int idx, vector<int>& ans) {

if (idx == nums.size()) {

anslist.push\_back(ans);

return;

}

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

if (vis[i] || (i > 0 && nums[i] == nums[i - 1] && !vis[i - 1])) { //和前一个相等且前一个没被填过

continue;

}

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

vis[i] = 1;

backtrack(nums, anslist, idx + 1, ans);

vis[i] = 0;

ans.pop\_back();

}

}

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

vector<vector<int>> ans;

vector<int> perm;

vis.resize(nums.size());

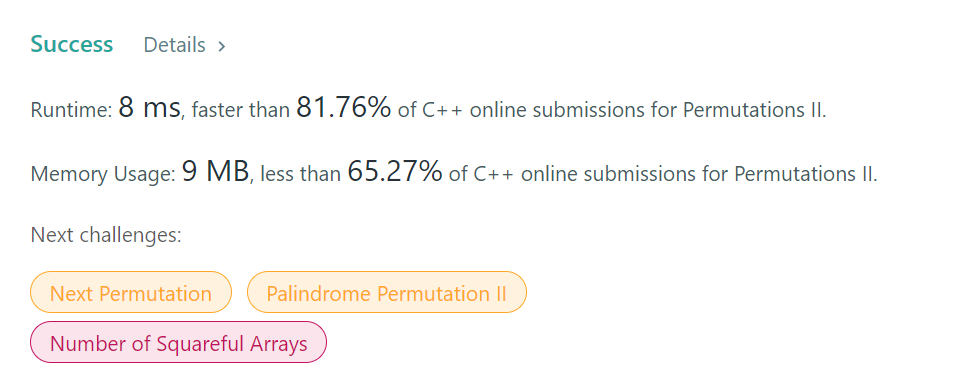
sort(nums.begin(), nums.end());

backtrack(nums, ans, 0, perm);

return ans;

}

};



**实验11**：45.跳跃游戏II

//给定一个非负整数数组，你最初位于数组的第一个位置。

//

//数组中的每个元素代表你在该位置可以跳跃的最大长度。

//

//你的目标是使用最少的跳跃次数到达数组的最后一个位置。

//示例 :

//

//输入: [2, 3, 1, 1, 4]

//输出 : 2

//解释 : 跳到最后一个位置的最小跳跃数是 2。

//从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

//

//分析：当前处在位置i，那么可以到达的最远位置为i+a[i]；

//所以只要搜索[i+1..i+a[i]]中的max（最远到达位置）,下一步跳到那里即可；

//什么时候可以跳完：若i+a[i]>=n,则i为最后一跳！步数+1算法结束即可

//这居然是难度hard的题目……一定是leetcode为我们找信心特意改成hard呜呜

代码如下：

class Solution {

private:

int sum = 0;

public:

int jump(vector<int>& nums) {

int cur = 0;

int max = 0;

int next = 0;

if (nums.size() == 1) { //一开始就在最后一个位置

return 0;

}

while (cur < nums.size()) {

//已经可以跳完了

if (cur + nums[cur] >= nums.size() - 1) { sum += 1; break; }

//还不能跳完

for (int i = cur + 1; i <= cur + nums[cur]; i++)

{

if (i + nums[i] > max) { max = i + nums[i]; next = i; }

}

//max作为下一步起跳点

sum += 1;

cur = next;

max = 0;

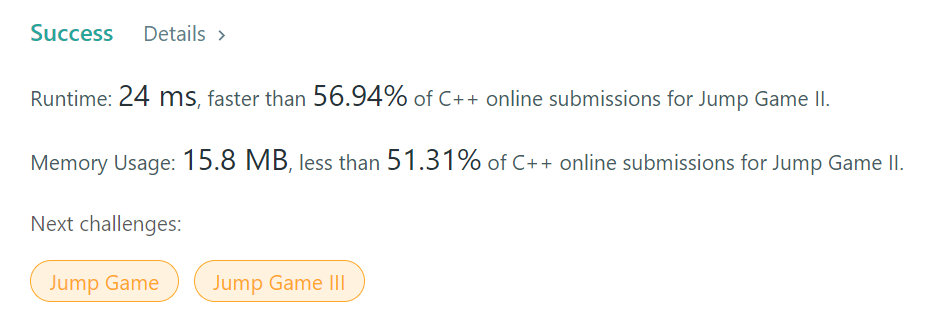
}

return sum;

}

};

运行截图：



**实验12**：122.买卖股票的最佳时机II

//给定一个数组，它的第 i 个元素是【一支】给定股票第 i 天的价格。

//

//设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

//

//注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

//

//示例 1:

//

//输入: [7, 1, 5, 3, 6, 4]

//输出 : 7

//解释 : 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5 - 1 = 4 。

//随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6 - 3 = 3 。

//

//分析：我们可以这样想这个问题：我们一直在买卖同一只股票，这只股票的价格是不断变动的；

//那么我们可能的最大获利是多少？显然是：价格波动图（横轴为时间纵轴为价格）上所有递增序列的落差的和！

//因此，我们只要不断计算序列中相邻两个价格的差值，若为正就将其加入maxprofit，若为负则不加入。代码如下：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int n = prices.size();

int profit = 0;

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

if (prices[i] - prices[i - 1] > 0) { profit += prices[i] - prices[i - 1]; }

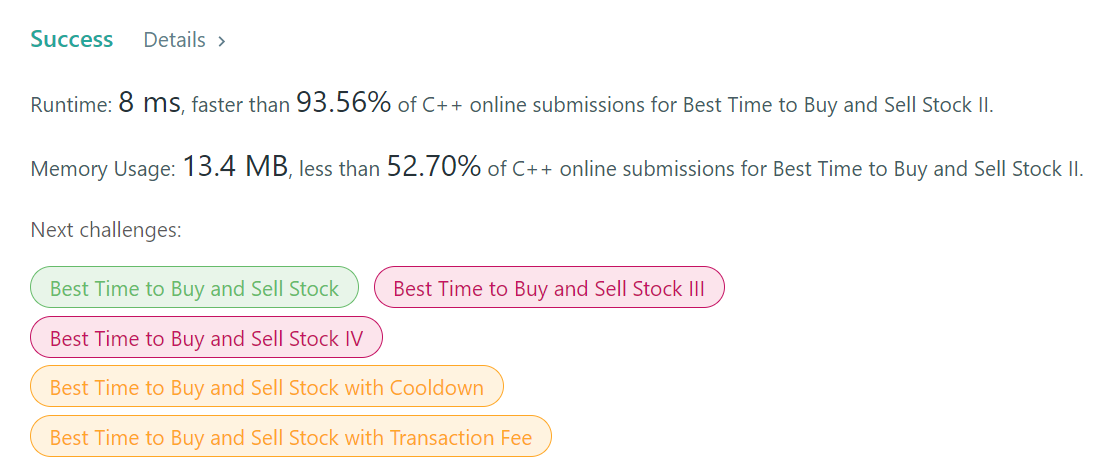
}

return profit;

}

};

运行截图：



**实验13**：300.最长上升子序列

//给定一个无序的整数数组，找到其中最长上升子序列的长度。

//

//示例 :

//

//输入: [10, 9, 2, 5, 3, 7, 101, 18]

//输出 : 4

//解释 : 最长的上升子序列是 [2, 3, 7, 101]，它的长度是 4。

//

//使用动态规划，设dp[i]为：以a[i]结尾的最长子序列长度;

//那么在填入dp[i]时，我们只需要扫描j∈[0..i-1]的每一个数字，若a[j]<a[i],

//那么a[i]就加入了以a[j]做结尾的子序列中！

//dp[i]=Max(dp[j]+1,dp[i])

其实这个也是老师讲过的经典例题……找找自信……

代码：

class Solution {

public:

int lengthOfLIS(vector<int>& nums) {

int n = nums.size();

vector<int> dp;

dp.resize(n);

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

dp[i] = 1;

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

if (nums[i] > nums[j]) { dp[i] = max(dp[j] + 1, dp[i]); }

}

}

int ans = 0;

vector<int>::iterator it = dp.begin();

for (it = dp.begin(); it < dp.end(); it++) {

if (\*it > ans) ans = \*it;

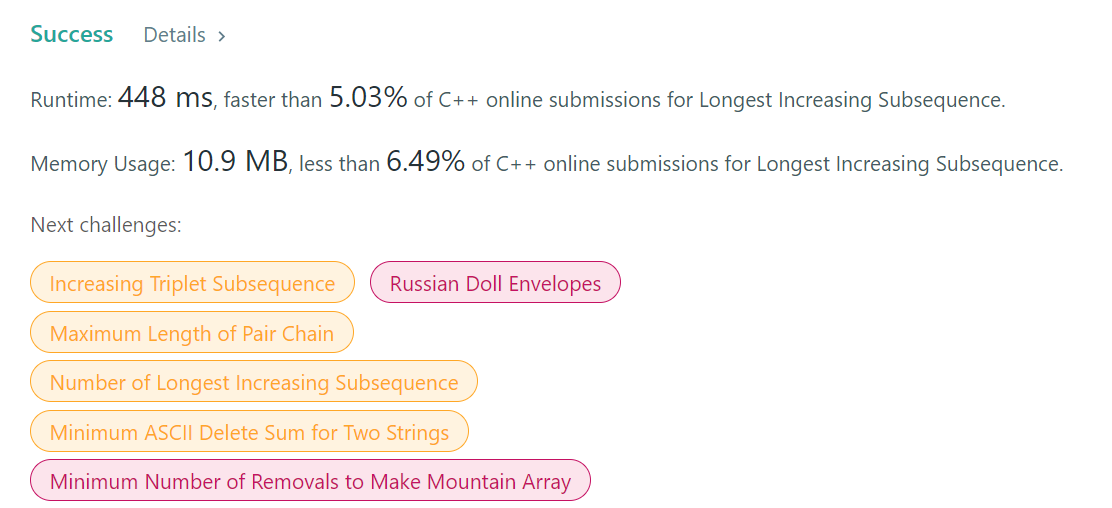
}

return ans;

}

};

运行截图：

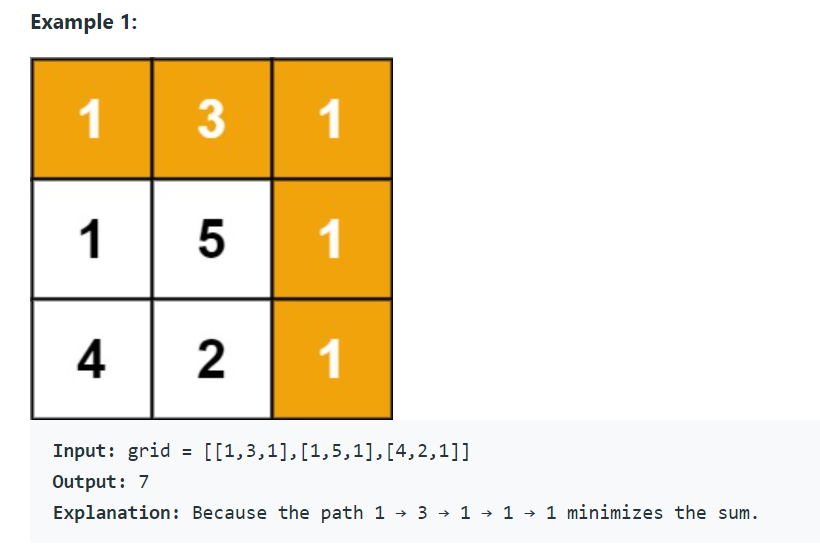


**实验14**：64.最小路径和

//给定一个包含非负整数的 m x n 网格 grid ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

//

//说明：每次只能向下或者向右移动一步。



//分析：这是一个典型的动态规划问题。设二维数组dp[i][j]:到达该格子的最小路径和。

//第一行的格子：dp[i][j]= dp[i][j-1]+grid[i][j](i=0)

//第一列的格子：dp[i][j]= dp[i-1][j]+grid[i][j](j=0)

//其它格子：dp[i][j]=Max(dp[i][j - 1] + grid[i][j]，dp[i - 1][j] + grid[i][j])

class Solution {

public:

int minPathSum(vector<vector<int>>& grid) {

vector<vector<int>> dp;

dp.resize(grid.size());

int n = grid.size();// 行数

int m = grid[0].size();// 列数

for (int k = 0; k < n; k++) { //每一行resize

dp[k].resize(m);

}

dp[0][0] = grid[0][0];

for(int i=1;i<n;i++){//第一列格子

dp[i][0] = dp[i - 1][0] + grid[i][0];

}

for (int j = 1; j < m; j++) {//第一行格子

dp[0][j] = dp[0][j - 1] + grid[0][j];

}

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

for (int b = 1; b < m; b++) {

dp[a][b] = min(dp[a][b - 1] + grid[a][b], dp[a - 1][b] + grid[a][b]);

}

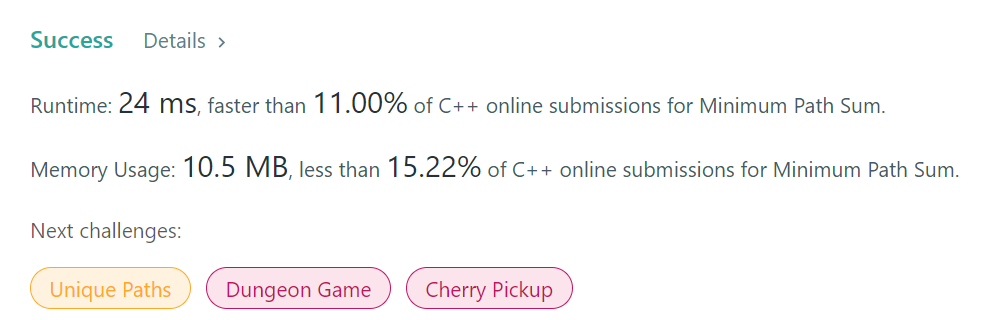
}

return dp[n - 1][m - 1];

}

};

运行截图：



**实验15**：1314.矩阵区域和

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

//

//i - K <= r <= i + K, j - K <= c <= j + K

//(r, c) 在矩阵内。

//

//

//示例 1：

//

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

//输出： [[12, 21, 16], [27, 45, 33], [24, 39, 28]]

//

//这简直是一个把动态规划拍在脸上的题，其实解题思路和上一题【最小路径和】相似，使用dp[][]来逐步记录，最后得到整个矩阵即可

若使用和上题类似的动态规划，需要四层循环，时间复杂度很高（可解决问题但超时了）：

class Solution {

private:

vector<vector<int>> ans;

public:

vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {

int n = mat.size();//行数

int m = mat[0].size();//列数

ans.resize(mat.size());

for (int k = 0; k < n; k++) { //每一行resize

ans[k].resize(m);

}

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

for (int j = 0; j < m; j++) {

ans[i][j] = 0;

//计算ans[i][j]

for (int k = i - K; k <= i + K; k++) {

for (int p = j - K; p <= j + k; p++) {

k = k < 0 ? 0 : k;

p = p < 0 ? 0 : p;

k = k >= n ? n - 1 : k;

p = p >= m ? m - 1 : p;

ans[i][j] += mat[k][p];

}

}

}

}

return ans;

}

};

这里将求前缀和的部分单独提出，将时间复杂度将为n^2：

class Solution {

public:

vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int K) {

int n = mat.size() + 1, m = mat[0].size() + 1;

vector<vector<int>> sumvec(n, vector<int>(m, 0));

vector<vector<int>> ans(n - 1, vector<int>(m - 1, 0));

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

for (int j = 1; j < m; j++) { //求前缀和

sumvec[i][j] = sumvec[i][j - 1] + sumvec[i - 1][j] - sumvec[i - 1][j - 1] + mat[i - 1][j - 1];

}

}

n--; m--;

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

for (int j = 0; j < m; j++) {

int L = (j - K) < 0 ? 0 : j - K;

int R = (j + K) >= m ? m - 1 : j + K;

int U = (i - K) < 0 ? 0 : i - K;

int D = (i + K) >= n ? n - 1 : i + K;

ans[i][j] = sumvec[D + 1][R + 1] - sumvec[D + 1][j - K] - sumvec[U][R + 1] + sumvec[U][L];

}

}

return ans;

}

};

运行截图：

