155.Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

* push(x) -- Push element x onto stack.
* pop() -- Removes the element on top of the stack.
* top() -- Get the top element.
* getMin() -- Retrieve the minimum element in the stack.

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;

}

};

1497. Check If Array Pairs Are Divisible by k

Given an array of integers arr of even length n and an integer k.

We want to divide the array into exactly n / 2 pairs such that the sum of each pair is divisible by k.

Return *True* If you can find a way to do that or *False* otherwise.

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;

}

};

134.Gas Station

There are *N* gas stations along a circular route, where the amount of gas at station *i* is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -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];

}

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;

}

1. 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;

}

};

1. 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);

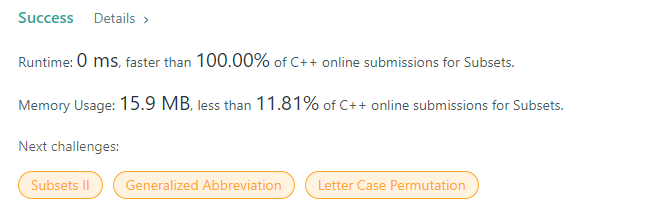
dfs(nums, 0, x);

return reslist;

}

};

第一次faster than 100%，截图纪念下：



1. 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：

