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.

class MinStack {

public:

stack<int> stack1;

stack<int> mstack;

void push(int x) {

if (mstack.empty() || x <= mstack.top()) mstack.push(x);

stack1.push(x);

}

void pop() {

if (mstack.top() == stack1.top()) mstack.pop();

stack1.pop();

}

int top() {

if (!stack1.empty()) return stack1.top();

}

int getMin() {

if (!mstack.empty()) return mstack.top();

}

};

/\* Find Minimum in Rotated Sorted Array II

Follow up for "Find Minimum in Rotated Sorted Array":

What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

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.

The array may contain duplicates.

\*/

class Solution {

public:

int findMin(vector<int> &num) {

int n = num.size();

int start = 0, end = n - 1;

while (start < end && num[start] >= num[end]) {

int mid = (start + end ) / 2;

if (num[mid] > num[start]) start = mid + 1;

else if (num[mid] < num[end]) end = mid;

else start++;

}

return num[start];

}

};

class Solution {

public:

int findMin(vector<int> &num) {

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

if (num[0] < num[num.size() - 1]) return num[0];

int start = 0, end = num.size() - 1;

while (start + 1 < end) {

int mid = start + (end - start) / 2;

if (num[mid] > num[start]) start = mid;

else if (num[mid] < num[end]) end = mid;

else {

int Min = INT\_MAX;

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

if (num[i] < Min) Min = num[i];

}

return Min;

}

}

return min(num[start], num[end]);

}

};

/\* Find Minimum in Rotated Sorted Array

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.

\*/

class Solution {

public:

int findMin(vector<int> &num) {

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

if (num[0] < num[num.size() - 1]) return num[0];

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

if (num[i] < num[i - 1]) return num[i];

}

}

};

class Solution {

public:

int findMin(vector<int> &num) {

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

if (num[0] < num[num.size() - 1]) return num[0];

int start = 0, end = num.size() - 1;

while (start + 1 < end) {

int mid = start + (end - start) / 2;

if (num[mid] > num[start]) start = mid;

else end = mid;

}

return min(num[start], num[end]);

}

};

/\* Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4],

the contiguous subarray [2,3] has the largest product = 6.

\*/

class Solution {

public:

int maxProduct(int A[], int n) {

if (n < 1) return 0;

int Max = A[0], Min = A[0], maxRet = A[0];

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

int mx = Max, mn = Min;

Max = max(max(A[i], A[i] \* mx), A[i] \* mn);

Min = min(min(A[i], A[i] \* mx), A[i] \* mn);

maxRet = max(maxRet, Max);

}

return maxRet;

}

};

class Solution {

public:

int maxProduct(int A[], int n) {

if (A == NULL || n == 0) {

return 0;

}

int maxSofar = A[0];

int minSofar = A[0];

int maxRet = A[0];

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

if (A[i] >= 0) {

maxSofar = max(maxSofar \* A[i], A[i]);

minSofar = min(minSofar \* A[i], A[i]);

}

else {

int temp = maxSofar;

maxSofar = max(minSofar \* A[i], A[i]);

minSofar = min(temp \* A[i], A[i]);

}

maxRet = max(maxRet, maxSofar);

}

return maxRet;

}

};

/\* Reverse Words in a String

Given an input string, reverse the string word by word.

For example,

Given s = "the sky is blue",

return "blue is sky the".

click to show clarification.

Clarification:

What constitutes a word?

A sequence of non-space characters constitutes a word.

Could the input string contain leading or trailing spaces?

Yes. However, your reversed string should not contain leading or trailing spaces.

How about multiple spaces between two words?

Reduce them to a single space in the reversed string.

\*/

class Solution {

public:

void reverseWords(string &s) {

if (s.size() == 0) return;

string ret;

int i = s.size() - 1;

while (i >= 0) {

while (i >= 0 && s[i] == ' ') i--;

int end = i;

while (i >= 0 && s[i] != ' ') i--;

if (end > i) ret = ret + " " + s.substr(i + 1, end - i);

}

if (ret.size() == 0) s = "";

else s = ret.substr(1, ret.size() - 1);

}

};

class Solution {

public:

void reverseWords(string &s) {

string result;

int pos = 0;

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

if (s[i] == ' ') {

if (i > pos ) // pos is the first character of each word. i is the next space after each word.

result = s.substr(pos,i-pos)+ " " + result ; // this will generate a space in the end.

pos = i + 1;

}

else if (i == s.size()-1)

result = s.substr(pos,s.size()-pos)+" "+result;

}

s = result.substr(0,result.size()-1) ;

}

};

/\* Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, \*, /. Each operand may be an integer or another expression.

Some examples:

["2", "1", "+", "3", "\*"] -> ((2 + 1) \* 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

\*/

class Solution {

public:

int evalRPN(vector<string> &tokens) {

stack<int> operand;

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

if (tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "\*" || tokens[i] == "/") {

int r = operand.top();

operand.pop();

int l = operand.top();

operand.pop();

operand.push(calculate(l, r, tokens[i]));

} else {

operand.push(atoi(tokens[i].c\_str())); // must use c string for atoi

}

}

return operand.top();

}

int calculate(int l, int r, string opt) {

if (opt == "+") return l + r;

else if (opt == "-") return l - r;

else if (opt == "\*") return l \* r;

else return l / r;

}

};

/\*Max Points on a Line

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

\*/

/\*\*

\* Definition for a point.

\* struct Point {

\* int x;

\* int y;

\* Point() : x(0), y(0) {}

\* Point(int a, int b) : x(a), y(b) {}

\* };

\*/

class Solution {

public:

int maxPoints(vector<Point> &points) {

int n = points.size();

if (n < 3) return n;

unordered\_map<float, int> hmap;

int ret = 0, repeated;

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

int x1 = points[i].x, y1 = points[i].y;

hmap.clear();

repeated = 1;

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

int x2 = points[j].x, y2 = points[j].y;

if (x1 == x2 && y1 == y2) repeated++;

else if (x1 == x2) hmap[INT\_MAX]++;

else hmap[(float)(y1 - y2) / (x1 - x2)]++; // need to add float to force the convertion from int to float. note that only need convertion for dividend or divisor.

}

ret = max(ret, repeated); // when all points are the same, we will not go into loop below. we need to update ret here.

for (unordered\_map<float, int>::iterator iter = hmap.begin(); iter != hmap.end(); iter++) {

ret = max(ret, iter->second + repeated);

}

}

return ret;

}

};

/\*Sort List

Sort a linked list in O(n log n) time using constant space complexity.

\*/

class Solution {

public:

ListNode \*sortList(ListNode \*head) {

if (head == NULL || head->next == NULL) return head;

ListNode \*mid = findMid(head);

ListNode \*right = sortList(mid->next);

mid->next = NULL;

ListNode \*left = sortList(head);

return merge(left, right);

}

ListNode \*merge(ListNode \*l1, ListNode \*l2) {

if (l1 == NULL) return l2;

if (l2 == NULL) return l1;

ListNode \*Dummy = new ListNode(0);

ListNode \*prev = Dummy;

while (l1 != NULL && l2 != NULL) {

if (l1->val < l2->val) {

prev->next = l1;

l1 = l1->next;

} else {

prev->next = l2;

l2 = l2->next;

}

prev = prev->next;

}

if (l1 == NULL) prev->next = l2;

else prev->next = l1;

return Dummy->next;

}

ListNode \*findMid(ListNode \*node) {

if (node == NULL) return NULL;

ListNode \*slow = node;

ListNode \*fast = node;

while (fast->next != NULL && fast->next->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

};

/\* Insertion Sort List

Sort a linked list using insertion sort.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*insertionSortList(ListNode \*head) {

if (head == NULL || head->next == NULL) return head;

ListNode \*Dummy = new ListNode(0);

ListNode \*node = head;

while (node != NULL) {

ListNode \*curr = Dummy->next;

ListNode \*prev = Dummy;

while (curr != NULL) {

if (node->val > curr->val) {

prev = curr;

curr = curr->next;

} else {

break;

}

}

ListNode\* temp = node;

node = node->next;

prev->next = temp;

temp->next = curr;

}

return Dummy->next;

}

};

LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

class LRUCache{

public:

struct dlinkedlist {

int key;

int val;

dlinkedlist \*prev;

dlinkedlist \*next;

dlinkedlist(): prev(NULL), next(NULL) {}

dlinkedlist(int k, int v): key(k), val(v), prev(NULL), next(NULL) {}

};

LRUCache(int capacity) {

size = capacity;

Dummyhead = new dlinkedlist;

Dummytail = new dlinkedlist;

Dummyhead->next = Dummytail;

Dummytail->prev = Dummyhead;

}

int get(int key) {

if (cacheM.count(key)) {

int val = cacheM[key]->val;

deleteNode(cacheM[key]);

cacheM[key] = addtoTail(key, val);

return val;

}

else {

return -1;

}

}

void set(int key, int value) {

if (cacheM.count(key)) {

deleteNode(cacheM[key]);

cacheM[key] = addtoTail(key, value);

}

else {

if (cacheM.size() == size) {

int k = Dummyhead->next->key;

deleteNode(Dummyhead->next);

cacheM.erase(k);

}

cacheM[key] = addtoTail(key, value);

}

}

void deleteNode(dlinkedlist\* curr) {

curr->prev->next = curr->next;

curr->next->prev = curr->prev;

delete curr;

}

dlinkedlist\* addtoTail(int key, int val) {

dlinkedlist \*node = new dlinkedlist(key, val);

Dummytail->prev->next = node;

node->prev = Dummytail->prev;

node->next = Dummytail;

Dummytail->prev = node;

return node;

}

private:

int size;

dlinkedlist \*Dummyhead;

dlinkedlist \*Dummytail;

map<int, dlinkedlist\*> cacheM;

};

/\* Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

1

\

2

/

3

return [3,2,1].

Note: Recursive solution is trivial, could you do it iteratively?

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<int> postorderTraversal(TreeNode \*root) {

vector<int> tree;

helper(root, tree);

return tree;

}

void helper(TreeNode \*root, vector<int> &tree) {

if (root != NULL) {

helper(root->left, tree);

helper(root->right, tree);

tree.push\_back(root->val);

}

}

};

/\* Binary Tree Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

1

\

2

/

3

return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

class Solution {

public:

vector<int> preorderTraversal(TreeNode \*root) {

vector<int> tree;

preOrderTraversalHelper(root, tree);

return tree;

}

void preOrderTraversalHelper(TreeNode \*node, vector<int> &tree) {

if (node != NULL) {

tree.push\_back(node->val);

preOrderTraversalHelper(node->left, tree);

preOrderTraversalHelper(node->right, tree);

}

}

};

class Solution {

public:

vector<int> preorderTraversal(TreeNode \*root) {

vector<int> solution;

if (root == NULL) {

return solution;

}

stack<TreeNode\*> tree;

tree.push(root);

while (!tree.empty()) {

TreeNode \*node = tree.top();

solution.push\_back(node->val);

tree.pop();

if (node->right != NULL) {

tree.push(node->right);

}

if (node->left != NULL) {

tree.push(node->left);

}

}

return solution;

}

};

/\*Reorder List

Given a singly linked list L: L0→L1→…→Ln-1→Ln,

reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…

You must do this in-place without altering the nodes' values.

For example,

Given {1,2,3,4}, reorder it to {1,4,2,3}.

class Solution {

public:

void reorderList(ListNode \*head) {

if (head == NULL || head->next == NULL) return;

ListNode \*mid = findMid(head);

ListNode \*newhead = reverse(mid->next);

mid->next = NULL;

ListNode \*node = head;

while (node->next != NULL && newhead->next != NULL) {

ListNode \*temp = node;

node = node->next;

temp->next = newhead;

temp = newhead;

newhead = newhead->next;

temp->next = node;

}

if (node->next == NULL) node->next = newhead;

else {

ListNode \*temp = node->next;

node->next = newhead;

newhead->next = temp;

}

}

ListNode \*findMid(ListNode \*head) {

ListNode \*slow = head;

ListNode \*fast = head;

while (fast->next != NULL && fast->next->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

ListNode \*reverse(ListNode \*head) {

if (head == NULL || head->next == NULL) return head;

ListNode \*prev = NULL;

ListNode \*curr = head;

while (curr != NULL) {

ListNode \*next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

}

return prev;

}

};

/\* Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Follow up:

Can you solve it without using extra space?

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*detectCycle(ListNode \*head) {

if (head == NULL) {

return NULL;

}

ListNode \*slow = head;

ListNode \*fast = head;

do {

if (fast == NULL || fast->next == NULL) {

return NULL;

}

slow = slow->next;

fast = fast->next->next;

} while (slow != fast);

while (head != slow) {

head = head->next;

slow = slow->next;

}

return head;

}

};

/\* Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:

Can you solve it without using extra space?

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

bool hasCycle(ListNode \*head) {

if (head == NULL) {

return false;

}

set<ListNode\*> node\_set;

node\_set.insert(head);

head = head->next;

while (head != NULL) {

if (node\_set.count(head)) {

return true;

}

else {

node\_set.insert(head);

head = head->next;

}

}

return false;

}

};

class Solution {

public:

bool hasCycle(ListNode \*head) {

if (head == NULL) {

return false;

}

ListNode \*slow = head;

ListNode \*fast = head->next;

while (fast != NULL && fast->next != NULL) {

if (fast == slow) {

return true;

}

else {

slow = slow->next;

fast = fast->next->next;

}

}

return false;

}

};

/\* Word Break II

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.

For example, given

s = "catsanddog",

dict = ["cat", "cats", "and", "sand", "dog"].

A solution is ["cats and dog", "cat sand dog"].

\*/

class Solution {

public:

vector<string> wordBreak(string s, unordered\_set<string> &dict) {

vector<string> result;

bool isSinDict = false;

// O(n) time for a quick check if there is no solution

for (int i = s.size() - 1; i >= 0; --i) {

if (dict.count(s.substr(i, s.size() - i))) {

isSinDict = true;

}

}

if (!isSinDict) {

return result;

}

vector<vector<string> > output;

DFS(s, 0, output, result, dict);

result.clear();

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

string sentence = output[i][0];

for (int j = 1; j < output[i].size(); ++j) {

sentence = sentence + " " + output[i][j];

}

result.push\_back(sentence);

}

return result;

}

void DFS(string &s, int start, vector<vector<string> > &output, vector<string> &result, unordered\_set<string> &dict) {

if (start == s.size()) {

output.push\_back(result);

return;

}

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

string temp = s.substr(start, i -start + 1);

if (dict.count(temp)) {

result.push\_back(temp);

DFS(s, i + 1, output, result, dict);

result.pop\_back();

}

}

}

};

/\* Word Break

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.

For example, given

s = "leetcode",

dict = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

\*/

class Solution {

public:

bool wordBreak(string s, unordered\_set<string> &dict) {

if (dict.count("") && s.size() == 0) {

return true;

}

if (s.size() == 0 || dict.size() == 0) {

return false;

}

vector<bool> isBreak(s.size() + 1, false);

isBreak[0] = true;

isBreak[1] = dict.count(s.substr(0, 1));

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

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

if (isBreak[j] && dict.count(s.substr(j, i - j))) {

isBreak[i] = true;

break;

}

}

}

return isBreak[s.size()];

}

};

/\* Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

\*/

/\*\*

\* Definition for singly-linked list with a random pointer.

\* struct RandomListNode {

\* int label;

\* RandomListNode \*next, \*random;

\* RandomListNode(int x) : label(x), next(NULL), random(NULL) {}

\* };

\*/

class Solution {

public:

RandomListNode \*copyRandomList(RandomListNode \*head) {

if (head == NULL) {

return NULL;

}

map<RandomListNode\*, RandomListNode\*> nodeMap;

RandomListNode \*Dummy = new RandomListNode(0);

RandomListNode \*start = head;

RandomListNode \*node = Dummy;

while (head != NULL) {

RandomListNode \*temp = new RandomListNode(head->label);

nodeMap[head] = temp;

node->next = temp;

node = node->next;

head = head->next;

}

node->next = NULL;

node = Dummy->next;

head = start;

while (node != NULL) {

node->random = nodeMap[head->random];

node = node->next;

head = head->next;

}

return Dummy->next;

}

};

/\* Single Number II

Given an array of integers, every element appears three times except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

\*/

class Solution {

public:

int singleNumber(int A[], int n) {

if (n == 1) return A[0];

int count[32] = {0};

int result = 0;

for (int i = 0; i < 32; ++i) {

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

if ( (A[j] >> i) & 1) {

count[i]++;

}

}

result |= ((count[i] % 3) << i);

}

return result;

}

};

/\* Single Number

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?

\*/

class Solution {

public:

int singleNumber(int A[], int n) {

int sum = A[0];

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

sum = (A[i]^sum);

}

return sum;

}

};

/\* Candy

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

Each child must have at least one candy.

Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

\*/

class Solution {

public:

int candy(vector<int> &ratings) {

if (ratings.size() <= 1) return ratings.size();

vector<int> nums(ratings.size());

nums[0] = 1;

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

if (ratings[i] > ratings[i - 1]) nums[i] = nums[i - 1] + 1;

else nums[i] = 1;

}

for (int i = ratings.size() - 2; i >= 0; i--) {

if (ratings[i] > ratings[i + 1] && nums[i] <= nums[i + 1]) nums[i] = nums[i + 1] + 1;

}

int sum = 0;

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

sum += nums[i];

}

return sum;

}

};

/\* 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, otherwise return -1.

Note:

The solution is guaranteed to be unique.

蛮精巧的一道题。最直白的解法就是从每一个点开始，遍历整个环，然后找出最后剩余油量最大的点。这个是O(n^2)的。但是这题明显不会无聊到让做题人写个两层循环这么简单。

仔细想一下，其实和以前求最大连续子数组和的题很像。

在任何一个节点，其实我们只关心油的损耗，定义：

diff[i] = gas[i] – cost[i] 0<=i <n

那么这题包含两个问题：

1. 能否在环上绕一圈？

2. 如果能，这个起点在哪里？

第一个问题，很简单，我对diff数组做个加和就好了，leftGas = ∑diff[i]， 如果最后leftGas是正值，那么肯定存在这么一个起始点。如果是负值，那说明，油的损耗大于油的供给，不可能有解。得到第一个问题的答案只需要O(n)。

对于第二个问题，起点在哪里？

假设，我们从环上取一个区间[i, j], j>i， 然后对于这个区间的diff加和，定义

sum[i,j] = ∑diff[k] where i<=k<j

如果sum[i,j]小于0，那么这个起点肯定不会在[i,j]这个区间里，跟第一个问题的原理一样。举个例子，假设i是[0,n]的解，那么我们知道 任意sum[k,i-1] (0<=k<i-1) 肯定是小于0的，否则解就应该是k。同理，sum[i,n]一定是大于0的，否则，解就不应该是i，而是i和n之间的某个点。所以第二题的答案，其实就是在0到n之间，找到第一个连续子序列（这个子序列的结尾必然是n）大于0的。

至此，两个问题都可以在一个循环中解决。

\*/

class Solution {

public:

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

int gasTank = 0;

int sum = 0;

int gasIndex = 0;

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

gasTank += gas[i] - cost[i];

sum += gas[i] - cost[i];

if (sum < 0) {

gasIndex = i + 1;

sum = 0;

}

}

if (gasTank < 0) return -1;

else return gasIndex;

}

};

/\* Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization:

Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2.

Second node is labeled as 1. Connect node 1 to node 2.

Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

1

/ \

/ \

0 --- 2

/ \

\\_/

\*/

/\*\*

\* Definition for undirected graph.

\* struct UndirectedGraphNode {

\* int label;

\* vector<UndirectedGraphNode \*> neighbors;

\* UndirectedGraphNode(int x) : label(x) {};

\* };

\*/

class Solution {

public:

UndirectedGraphNode \*cloneGraph(UndirectedGraphNode \*node) {

if (node == NULL) {

return NULL;

}

queue<UndirectedGraphNode\*> que;

map<UndirectedGraphNode\*, UndirectedGraphNode\*> nodeMap;

UndirectedGraphNode \*root = new UndirectedGraphNode(node->label);

nodeMap[node] = root;

que.push(node);

while (!que.empty()) {

UndirectedGraphNode \*temp = que.front();

for (int i = 0; i < temp->neighbors.size(); ++i) {

UndirectedGraphNode \*neighb = temp->neighbors[i];

if (!nodeMap.count(neighb)) {

UndirectedGraphNode \*nodeCopy = new UndirectedGraphNode(neighb->label);

nodeMap[temp]->neighbors.push\_back(nodeCopy);

nodeMap[neighb] = nodeCopy;

que.push(neighb);

}

else {

nodeMap[temp]->neighbors.push\_back(nodeMap[neighb]);

}

}

que.pop();

}

return root;

}

};

/\* Palindrome Partitioning II

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given s = "aab",

Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

\*/

class Solution {

public:

int minCut(string s) {

vector<int> minCut(s.size() + 1, 0); //minCut[i] are minimum cuts we need for the first i characters

vector<vector<int> > isPalindrome(s.size(), vector<int>(s.size(), 0)); //isPalindrome[i][j] indicates that if the substring from index i to j is a palindrome.

minCut[0] = -1; // for later calculation

minCut[1] = 0; // no need to cut if there is only character in a string

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

minCut[i] = i - 1; // worst scenario that we have to cut string to characters. This is also the default value if no longer substring palindrome in the string

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

if (s[j] == s[i - 1]) {

if (i - 1 - j < 3 || isPalindrome[j + 1][i - 2]) {

isPalindrome[j][i - 1] = 1;

minCut[i] = min(minCut[i], minCut[j] + 1);

}

}

}

}

return minCut[s.size()];

}

};

/\* Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab",

Return

[

["aa","b"],

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

]

\*/

class Solution {

public:

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

vector<vector<string> > ret;

vector<string> sol;

dfs(ret, sol, s, 0);

return ret;

}

void dfs(vector<vector<string> > &ret, vector<string> &sol, string &s, int start) {

if (start == s.size()) {

ret.push\_back(sol);

return;

}

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

if (isPalindrome(s, start, i)) {

sol.push\_back(s.substr(start, i - start + 1));

dfs(ret, sol, s, i + 1);

sol.pop\_back();

}

}

}

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

while (start < end) {

if (s[start] != s[end]) return false;

else {

start++;

end--;

}

}

return true;

}

};

/\* Surrounded Regions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

X X X X

X O O X

X X O X

X O X X

After running your function, the board should be:

X X X X

X X X X

X X X X

X O X X

class Solution {

public:

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

if (board.size() == 0) return;

int row = board.size();

int col = board[0].size();

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

bfs(0, i, board);

bfs(row - 1, i, board);

}

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

bfs(i, 0, board);

bfs(i, col - 1, board);

}

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

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

if (board[i][j] == 'A') board[i][j] = 'O';

else if (board[i][j] == 'O') board[i][j] = 'X';

}

}

}

void bfs(int r, int c, vector<vector<char> > &board) {

if (board[r][c] != 'O') return;

board[r][c] = 'A';

queue<pair<int, int> > dots;

dots.push(make\_pair(r, c));

while (!dots.empty()) {

int x = dots.front().first;

int y = dots.front().second;

dots.pop();

if (x + 1 < board.size() && board[x + 1][y] == 'O') {

board[x + 1][y] = 'A';

dots.push(make\_pair(x + 1, y));

}

if (x - 1 >= 0 && board[x - 1][y] == 'O') {

board[x - 1][y] = 'A';

dots.push(make\_pair(x - 1, y));

}

if (y + 1 < board[0].size() && board[x][y + 1] == 'O') {

board[x][y + 1] = 'A';

dots.push(make\_pair(x, y + 1));

}

if (y - 1 >= 0 && board[x][y - 1] == 'O') {

board[x][y - 1] = 'A';

dots.push(make\_pair(x, y - 1));

}

}

}

};

/\* Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

1

/ \

2 3

The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

int sumNumbers(TreeNode \*root) {

int result = 0;

int sum = 0;

dfs(root, sum, result);

return result;

}

void dfs(TreeNode \*node, int sum, int &total) {

if (node == NULL) return;

sum = 10 \* sum + node->val;

if (node->left == NULL && node->right == NULL) {

total += sum;

return;

}

dfs(node->left, sum, total);

dfs(node->right, sum, total);

}

};

/\* Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given [100, 4, 200, 1, 3, 2],

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

Your algorithm should run in O(n) complexity.

\*/

class Solution {

public:

int longestConsecutive(vector<int> &num) {

int ret = 0;

unordered\_set<int> iset(num.begin(), num.end());

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

int n = 1;

int curr = num[i];

while (iset.count(--curr)) { // don't forget iset.count()

n++;

iset.erase(curr);

}

curr = num[i];

while (iset.count(++curr)) {

n++;

iset.erase(curr);

}

ret = max(ret, n);

}

return ret;

}

};

/\* Word Ladder

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that:

Only one letter can be changed at a time

Each intermediate word must exist in the dictionary

For example,

Given:

start = "hit"

end = "cog"

dict = ["hot","dot","dog","lot","log"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",

return its length 5.

Note:

Return 0 if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

\*/

class Solution {

public:

typedef struct node {

string s;

int d;

node(string str, int dis): s(str), d(dis) {}

};

int ladderLength(string start, string end, unordered\_set<string> &dict) {

if (dict.size() == 0 || start.size() != end.size()) {

return 0;

}

if (isTransformable(start, end)) return 2;

int n = start.size();

queue<node\*> que;

node\* source = new node(start, 1);

que.push(source);

while (!que.empty()) {

node \*curr = que.front();

que.pop();

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

string s = curr->s;

for (int c = 'a'; c != 'z' + 1; ++c) {

if (s[i] != c) {

s[i] = c;

if (dict.count(s)) {

if(isTransformable(s, end)) return curr->d + 2;

node\* temp = new node(s, curr->d + 1);

que.push(temp);

dict.erase(s);

}

}

}

}

}

return 0;

}

bool isTransformable(string &s1, string &s2) {

int n = 0;

for (int i = 0; i < s1.size(); i++) {

if (s1[i] != s2[i]) n++;

}

if (n == 1) return true;

else return false;

}

};

/\* Word Ladder II

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

Only one letter can be changed at a time

Each intermediate word must exist in the dictionary

For example,

Given:

start = "hit"

end = "cog"

dict = ["hot","dot","dog","lot","log"]

Return

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

Note:

All words have the same length.

All words contain only lowercase alphabetic characters.

\*/

class Solution {

public:

struct node {

string s;

int d;

vector<node\*> neighbors;

node(string w, int distance) : s(w), d(distance) {}

};

vector<vector<string> > findLadders(string start, string end, unordered\_set<string> &dict) {

vector<vector<string> > ret;

vector<string> sol;

if (dict.size() == 0 || start.size() != end.size()) {

return ret;

}

if (isTransformable(start, end) && dict.count(start) && dict.count(end)) {

sol.push\_back(start);

sol.push\_back(end);

ret.push\_back(sol);

return ret;

}

int l = INT\_MAX;

node\* source = constructGraph(start, end, dict, l);

if (l == INT\_MAX) return ret;

sol.push\_back(start);

dfs(source, end, ret, sol);

return ret;

}

void dfs(node \*head, string end, vector<vector<string> > &ret, vector<string> &sol) {

if (head->s == end) {

ret.push\_back(sol);

return;

}

for (int i = 0; i < head->neighbors.size(); i++) {

sol.push\_back(head->neighbors[i]->s);

dfs(head->neighbors[i], end, ret, sol);

sol.pop\_back();

}

}

node \*constructGraph(string start, string end, unordered\_set<string> &dict, int &len) {

node \*head = new node(start, 1);

queue<node\*> candt;

candt.push(head);

unordered\_map<string, node\*> hmap;

hmap[start] = head;

int level = 1;

while (level < len && !candt.empty()) {

int n = candt.size();

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

node\* curr = candt.front();

candt.pop();

for (int i = 0; i < curr->s.size(); i++) {

string copy = curr->s;

for (int c = 'a'; c <= 'z'; c++) {

if (c == curr->s[i]) continue;

copy[i] = c;

if (copy == end) {

len = min(len, curr->d + 1);

if (!hmap.count(copy)) {

node \*temp = new node(copy, len);

hmap[end] = temp;

curr->neighbors.push\_back(temp);

}

else curr->neighbors.push\_back(hmap[copy]);

}

else if (level + 1 < len && dict.count(copy)) {

if (!hmap.count(copy)) {

node \*temp = new node(copy, curr->d + 1);

hmap[copy] = temp;

candt.push(temp);

curr->neighbors.push\_back(temp);

}

else if (hmap[copy]->d == level + 1) {

//candt.push(hmap[copy]);

curr->neighbors.push\_back(hmap[copy]);

}

}

}

}

}

level++;

}

return head;

}

bool isTransformable(string &s1, string &s2) {

int n = 0;

for (int i = 0; i < s1.size(); i++) {

if (s1[i] != s2[i]) n++;

}

if (n == 1) return true;

else return false;

}

};

/\* Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,

"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is not a palindrome.

Note:

Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

\*/

class Solution {

public:

bool isPalindrome(string s) {

int i = 0, j = s.size() - 1;

while (i < j) {

while (i < j && !isalnum(s[i])) i++;

while (i < j && !isalnum(s[j])) j--;

if (tolower(s[i]) != tolower(s[j])) return false;

i++; j--;

}

return true;

}

};

/\* Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example:

Given the below binary tree,

1

/ \

2 3

Return 6.

class Solution {

public:

int maxPathSum(TreeNode \*root) {

int csum = 0, ret = INT\_MIN;

mpathSum(root, csum, ret);

return ret;

}

void mpathSum(TreeNode \*node, int &cSum, int &maxSum) {

if (node == NULL) {

cSum = 0;

return;

}

int lsum = 0, rsum = 0; //lsum and rsum are the root-node maxSum of left subtree and right subtree

mpathSum(node->left, lsum, maxSum);

mpathSum(node->right, rsum, maxSum);

cSum = max(node->val, max(node->val + lsum, node->val + rsum)); // cSum is the root-node maxSum of current tree

maxSum = max(maxSum, max(cSum, node->val + lsum + rsum)); // maxSum is the max path sum of current tree

}

};

class Solution {

public:

typedef class Resulttype {

public:

int singlePath;

int maxPath;

Resulttype(int singlepath, int maxpath): singlePath(singlepath), maxPath(maxpath) {}

} ResultType;

int maxPathSum(TreeNode \*root) {

return helper(root).maxPath;

}

ResultType helper(TreeNode \*node) {

if (node == NULL) {

return ResultType(0, INT\_MIN);

}

//divide

ResultType left = helper(node->left);

ResultType right = helper(node->right);

//conquer

int singlePath = max(left.singlePath, right.singlePath) + node->val;

singlePath = max(singlePath, 0);

int maxPath = max(left.maxPath, right.maxPath);

maxPath = max(maxPath, left.singlePath + right.singlePath + node->val);

return ResultType(singlePath, maxPath);

}

};

/\* Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

\*/

class Solution {

public:

int maxProfit(vector<int> &prices) {

if (prices.size() < 2) return 0;

vector<int> left(prices.size());

left[0] = 0;

int Min = prices[0];

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

Min = min(Min, prices[i]);

left[i] = max(left[i - 1], prices[i] - Min);

}

vector<int> right(prices.size());

right[prices.size() - 1] = 0;

int Max = prices.back();

for (int i = prices.size() - 2; i >= 0; i--) {

Max = max(Max, prices[i]);

right[i] = max(right[i + 1], Max - prices[i]);

}

int profit = 0;

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

profit = max(profit, left[i] + right[i]); // we can sell and buy at the same day.

}

return profit;

}

};

/\* Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times).

However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

\*/

class Solution {

public:

int maxProfit(vector<int> &prices) {

int n = prices.size();

if (n < 2) return 0;

int maxP = 0;

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

int diff = prices[i] - prices[i - 1];

if (diff > 0) maxP += diff;

}

return maxP;

}

};

/\* Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

\*/

class Solution {

public:

int maxProfit(vector<int> &prices) {

if (prices.size() <= 1) {

return 0;

}

int Min = prices[0];

int maxProfit = 0;

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

Min = min(Min, prices[i]);

maxProfit = max(maxProfit, prices[i] - Min);

}

return maxProfit;

}

};

/\* Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note:

Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

\*/

class Solution {

public:

int minimumTotal(vector<vector<int> > &triangle) {

int m = triangle.size();

vector<int> dp(triangle[m - 1]);

for (int i = m - 2; i >= 0; --i) {

for (int j = 0; j < triangle[i].size(); j++) {

if (dp[j] < dp[j + 1]) dp[j] = dp[j] + triangle[i][j];

else dp[j] = dp[j + 1] + triangle[i][j];

}

}

return dp[0];

}

};

/\* Pascal's Triangle II

\*/

class Solution {

public:

vector<int> getRow(int rowIndex) {

vector<int> ret;

vector<int> temp;

temp.push\_back(1);

if (rowIndex == 0) return temp;

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

ret.clear();

ret.push\_back(1);

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

ret.push\_back(temp[j - 1] + temp[j]);

}

ret.push\_back(1);

temp = ret;

}

return ret;

}

};

/\* Pascal's Triangle

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

For example, given numRows = 5,

Return

[

[1],

[1,1],

[1,2,1],

[1,3,3,1],

[1,4,6,4,1]

]

\*/

class Solution {

public:

vector<vector<int> > generate(int numRows) {

vector<vector<int> > ret(numRows);

if (numRows == 0) return ret;

ret[0].push\_back(1);

if (numRows == 1) return ret;

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

ret[i].push\_back(1);

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

ret[i].push\_back(ret[i - 1][j - 1] + ret[i - 1][j]);

}

ret[i].push\_back(1);

}

return ret;

}

};

/\* Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space.

For example,

Given the following binary tree,

1

/ \

2 3

/ \ \

4 5 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ \

4-> 5 -> 7 -> NULL

\*/

/\*\*

\* Definition for binary tree with next pointer.

\* struct TreeLinkNode {

\* int val;

\* TreeLinkNode \*left, \*right, \*next;

\* TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}

\* };

\*/

class Solution {

public:

void connect(TreeLinkNode \*root) {

if (root == NULL) return;

queue<TreeLinkNode\*> treeQue;

vector<TreeLinkNode\*> level;

treeQue.push(root);

while (!treeQue.empty()) {

int size = treeQue.size();

level.clear();

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

TreeLinkNode\* node = treeQue.front();

treeQue.pop();

level.push\_back(node);

if (node->left != NULL) treeQue.push(node->left);

if (node->right != NULL) treeQue.push(node->right);

}

for (int i = 0; i < level.size() - 1; i++) level[i]->next = level[i + 1];

level[level.size() - 1]->next = NULL;

}

}

};

/\* Populating Next Right Pointers in Each Node

Given a binary tree

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space.

You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

\*/

/\*\*

\* Definition for binary tree with next pointer.

\* struct TreeLinkNode {

\* int val;

\* TreeLinkNode \*left, \*right, \*next;

\* TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}

\* };

\*/

class Solution {

public:

void connect(TreeLinkNode \*root) {

if (root == NULL) return;

queue<TreeLinkNode\*> treeQue;

vector<TreeLinkNode\*> level;

treeQue.push(root);

while (!treeQue.empty()) {

int size = treeQue.size();

level.clear();

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

TreeLinkNode\* node = treeQue.front();

treeQue.pop();

level.push\_back(node);

if (node->left != NULL) treeQue.push(node->left);

if (node->right != NULL) treeQue.push(node->right);

}

for (int i = 0; i < level.size() - 1; i++) level[i]->next = level[i + 1];

level[level.size() - 1]->next = NULL;

}

}

};

/\* Distinct Subsequences @@@@@@@@@@@@@@@@@@@

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Here is an example:

S = "rabbbit", T = "rabbit"

Return 3.

\*/

class Solution {

public:

int numDistinct(string S, string T) {

if (S.size() < T.size() || S.size() == 0) {

return 0;

}

if (S == T) {

return 1;

}

int m = S.size();

int n = T.size();

vector<vector<int> > dp(m + 1, vector<int>(n + 1, 0));

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

dp[i][0] = 1;

}

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

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

dp[i][j] = dp[i - 1][j];

if (S[i - 1] == T[j - 1]) {

dp[i][j] += dp[i - 1][j - 1];

}

}

}

return dp[m][n];

}

};

/\* Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example,

Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

click to show hints.

Hints:

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

void flatten(TreeNode \*root) {

if (root == NULL) return; // why we need this?

vector<TreeNode\*> nodeVector;

preOrder(root, nodeVector);

int i;

for (i = 1; i < nodeVector.size(); i++) {

nodeVector[i - 1]->left = NULL; // do not forget this!!!

nodeVector[i - 1]->right = nodeVector[i];

}

}

void preOrder(TreeNode \*node, vector<TreeNode\*> &nodeV) {

if (node != NULL) {

nodeV.push\_back(node);

preOrder(node->left, nodeV);

preOrder(node->right, nodeV);

}

}

};

/\* Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ / \

7 2 5 1

return

[

[5,4,11,2],

[5,8,4,5]

]

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<vector<int> > pathSum(TreeNode \*root, int sum) {

vector<vector<int> > result;

vector<int> sol;

dfs(root, result, sol, sum, 0);

return result;

}

void dfs(TreeNode \*node, vector<vector<int> > &ret, vector<int> sol, int sum, int path) {

if (node == NULL) return;

sol.push\_back(node->val);

path += node->val;

if (node->left == NULL && node->right == NULL) {

if (path == sum) ret.push\_back(sol);

return;

}

dfs(node->left, ret, sol, sum, path);

dfs(node->right, ret, sol, sum, path);

}

};

/\* Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:

Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

class Solution {

public:

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

bool pathSum = false;

dfs(root, 0, sum, pathSum);

return pathSum;

}

void dfs(TreeNode \*node, int path, int sum, bool &is) {

if (node == NULL) return;

path += node->val;

if (node->left == NULL && node->right == NULL) {

if (path == sum) is = true;

return;

}

dfs(node->left, path, sum, is);

dfs(node->right, path, sum, is);

}

};

class Solution {

public:

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

int pathsum = 0;

return haspathsum(root, pathsum, sum);

}

bool haspathsum(TreeNode \*node, int sum, int target) {

if (node != NULL) {

sum += node->val;

if (node->left == NULL && node->right == NULL) {

if (sum == target) return true;

else return false;

}

return (haspathsum(node->left, 0, target - sum) || haspathsum(node->right, 0, target - sum));

}

else return false;

}

};

/\* Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

int minDepth(TreeNode \*root) {

if (root == NULL) return 0;

if (root->left == NULL && root->right == NULL) return 1;

else if (root->left != NULL && root->right == NULL) return minDepth(root->left) + 1;

else if (root->left == NULL && root->right != NULL) return minDepth(root->right) + 1;

else return min(minDepth(root->left), minDepth(root->right)) + 1;

}

};

class Solution {

public:

int minDepth(TreeNode \*root) {

if (root == NULL) return 0;

int minDep = INT\_MAX;

dfs(root, 0, minDep);

return minDep;

}

void dfs(TreeNode \*node, int depth, int &minDept) {

if (node == NULL) return;

depth++;

if (node->left == NULL && node->right == NULL) {

minDept = min(minDept, depth);

return;

}

dfs(node->left, depth, minDept);

dfs(node->right, depth, minDept);

}

};

/\* Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

bool isBalanced(TreeNode \*root) {

if (root == NULL) {

return true;

}

return maxDepth(root) != -1;

}

int maxDepth(TreeNode \*node) {

if(node == NULL) {

return 0;

}

int left = maxDepth(node->left);

int right = maxDepth(node->right);

if (left == -1 || right == -1 || abs(left - right) > 1) {

return -1;

}

return (left > right ? left : right) + 1;

}

};

/\* Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

TreeNode \*sortedArrayToBST(vector<int> &num) {

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

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

TreeNode\* root = new TreeNode(num[0]);

return root;

}

if (num.size() == 2) {

TreeNode\* root = new TreeNode(num[0]);

TreeNode\* L = new TreeNode(num[1]);

root->right = L;

return root;

}

int mid = (num.size() - 1) / 2;

TreeNode \*node = new TreeNode(num[mid]);

vector<int> left(num.begin(), num.begin() + mid);

vector<int> right(num.begin() + mid + 1, num.end());

TreeNode \*L = sortedArrayToBST(left);

TreeNode \*R = sortedArrayToBST(right);

node->left = L;

node->right = R;

return node;

}

};

class Solution {

public:

TreeNode \*sortedArrayToBST(vector<int> &num) {

return findRoot(num, 0, num.size() - 1);

}

TreeNode \*findRoot(vector<int> &num, int start, int end) {

if (start == end) {

TreeNode \*node = new TreeNode(num[start]);

return node;

}

else if (start > end) return NULL;

int mid = start + (end - start) / 2;

TreeNode \*root = new TreeNode(num[mid]);

TreeNode \*left = findRoot(num, start, mid - 1);

TreeNode \*right = findRoot(num, mid + 1, end);

root->left = left;

root->right = right;

return root;

}

};

/\* Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

TreeNode \*sortedListToBST(ListNode \*head) {

if (head == NULL) return NULL;

if (head->next == NULL) {

TreeNode \*root = new TreeNode(head->val);

return root;

}

ListNode \*mid = findMid(head);

ListNode \*right = mid->next->next;

TreeNode \*root = new TreeNode(mid->next->val);

mid->next = NULL;

root->left = sortedListToBST(head);

root->right = sortedListToBST(right);

return root;

}

ListNode \*findMid(ListNode \*head) {

if (head == NULL) {

return NULL;

}

ListNode \*Dummy = new ListNode(0);;

Dummy->next = head;

ListNode \*slow = Dummy;

ListNode \*fast = Dummy->next;

while (fast != NULL && fast->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

};

/\* Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:

Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

[

[15,7],

[9,20],

[3]

]

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<vector<int> > levelOrderBottom(TreeNode \*root) {

vector<vector<int> > solution;

if (root == NULL) {

return solution;

}

queue<TreeNode\*> temp;

temp.push(root);

vector<int> level;

while (!temp.empty()) {

size\_t n = temp.size();

for (size\_t i = 0; i < n; ++i) {

TreeNode\* node = temp.front();

level.push\_back(node->val);

if (node->left != NULL) {

temp.push(node->left);

}

if (node->right !=NULL) {

temp.push(node->right);

}

temp.pop();

}

solution.push\_back(level);

level.clear();

}

reverse(solution.begin(), solution.end());

}

};

/\* Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

TreeNode \*buildTree(vector<int> &preorder, vector<int> &inorder) {

return findParent(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1);

}

TreeNode \*findParent(vector<int> &preorder, int prestart, int preend, vector<int> &inorder, int instart, int inend) {

if (instart > inend) return NULL;

TreeNode \*root = new TreeNode(preorder[prestart]);

int index = findIndex(inorder, instart, inend, preorder[prestart]);

root->left = findParent(preorder, prestart + 1, prestart + index, inorder, instart, instart + index - 1);

root->right = findParent(preorder, prestart + index + 1, preend, inorder, instart + index + 1, inend);

return root;

}

int findIndex(vector<int> &inorder, int instart, int inend, int target) {

for (int i = instart; i <= inend; i++) {

if (inorder[i] == target) return i - instart;

}

}

};

/\* Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

TreeNode \*buildTree(vector<int> &inorder, vector<int> &postorder) {

return findParent(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() - 1);

}

TreeNode \*findParent(vector<int> &inorder, int instart, int inend, vector<int> &postorder, int posstart, int posend) {

if (instart > inend) return NULL;

TreeNode \*root = new TreeNode(postorder[posend]);

int index = findIndex(inorder, instart, inend, postorder[posend]);

root->left = findParent(inorder, instart, instart + index - 1, postorder, posstart, posstart + index - 1);

root->right = findParent(inorder, instart + index + 1, inend, postorder, posstart + index, posend - 1);

return root;

}

int findIndex(vector<int> &inorder, int start, int end, int target) {

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

if (inorder[i] == target) return i - start;

}

}

};

/\* Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution { // divide and conque

public:

int maxDepth(TreeNode \*root) {

if (root == NULL) {

return 0;

}

int left\_max = maxDepth(root->left);

int right\_max = maxDepth(root->right);

if (left\_max > right\_max) {

return left\_max + 1;

}

else {

return right\_max + 1;

}

}

};

class Solution { // dfs

public:

int maxDepth(TreeNode \*root) {

int ret = 0, d = 0;

dfs(root, d, ret);

return ret;

}

void dfs(TreeNode \*node, int dep, int &mDepth) {

if (node == NULL) return;

dep++;

if (node->left == NULL && node->right == NULL) {

mDepth = max(mDepth, dep);

}

dfs(node->left, dep, mDepth);

dfs(node->right, dep, mDepth);

}

};

/\* Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:

Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its level order traversal as:

[

[3],

[9,20],

[15,7]

]

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<vector<int> > levelOrder(TreeNode \*root) {

vector<vector<int> > solution;

if (root == NULL) {

return solution;

}

queue<TreeNode\*> buffer;

buffer.push(root);

vector<int> level;

while (!buffer.empty()) {

size\_t n = buffer.size();

for (size\_t i = 0; i < n; ++i) {

TreeNode\* node = buffer.front();

level.push\_back(node->val);

if (node->left != NULL) {

buffer.push(node->left);

}

if (node->right != NULL) {

buffer.push(node->right);

}

buffer.pop();

}

solution.push\_back(level);

level.clear();

}

return solution;

}

};

/\* Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<vector<int> > zigzagLevelOrder(TreeNode \*root) {

vector<vector<int> > solution;

if (root == NULL) {

return solution;

}

queue<TreeNode\*> temp;

temp.push(root);

vector<int> level;

while (!temp.empty()) {

size\_t n = temp.size();

for (size\_t i = 0; i < n; ++i) {

TreeNode \*node = temp.front();

level.push\_back(node->val);

if (node->left != NULL) {

temp.push(node->left);

}

if (node->right != NULL) {

temp.push(node->right);

}

temp.pop();

}

solution.push\_back(level);

level.clear();

}

size\_t n = solution.size();

for (size\_t i = 1; i < n; i += 2) {

reverse(solution[i].begin(), solution[i].end());

}

}

};

/\* Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

class Solution {

public:

bool isSymmetric(TreeNode \*root) {

if (root == NULL) return true;

return isSymmetricRecursive(root->left, root->right);

}

bool isSymmetricRecursive(TreeNode \*node1, TreeNode \*node2) {

if (node1 != NULL && node2 != NULL) {

return (node1->val == node2->val) && isSymmetricRecursive(node1->left, node2->right)

&& isSymmetricRecursive(node1->right, node2->left);

} else if (node1 != NULL || node2 != NULL) return false;

else if (node1 == NULL && node2 == NULL) return true;

}

};

class Solution {

public:

bool isSymmetric(TreeNode \*root) {

if (root == NULL) return true;

if (root->left == NULL && root->right == NULL) return true;

if (root->left != NULL && root->right != NULL) {

queue<TreeNode\*> left;

queue<TreeNode\*> right;

left.push(root->left);

right.push(root->right);

while (!left.empty() && !right.empty()) {

int l = left.size(), r = right.size();

if (l != r) return false;

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

TreeNode \*L = left.front();

TreeNode \*R = right.front();

if (L->val != R->val) return false;

left.pop();

right.pop();

if (L->left != NULL && R->right != NULL) {

left.push(L->left);

right.push(R->right);

} else if (L->left != NULL || R->right != NULL) return false;

if (L->right != NULL && R->left != NULL) {

left.push(L->right);

right.push(R->left);

} else if (L->right != NULL || R->left != NULL) return false;

}

}

return true;

}

else return false;

}

};

/\* Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

bool isSameTree(TreeNode \*p, TreeNode \*q) {

if (p == nullptr && q == nullptr)

{

return true;

}

else

{

if (p != nullptr && q != nullptr)

{

if (p->val == q->val)

{

return isSameTree(p->left, q->left) && isSameTree(p->right,q->right);

}

else

{

return false;

}

}

else

{

return false;

}

}

}

};

/\* Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

void recoverTree(TreeNode \*root) {

vector<TreeNode\*> tree;

inorder(root, tree);

int i, j;

for (i = 0; i < tree.size() - 1; i++) {

if (tree[i]->val > tree[i + 1]->val) break;

}

for (j = tree.size() - 1; j > i; j--) {

if (tree[j]->val < tree[j - 1]->val) break;

}

int temp = tree[i]->val;

tree[i]->val = tree[j]->val;

tree[j]->val = temp;

}

void inorder(TreeNode \*root, vector<TreeNode\*> &iv) {

if (root != NULL) {

inorder(root->left, iv);

iv.push\_back(root);

inorder(root->right, iv);

}

}

};

class Solution {

public:

void recoverTree(TreeNode \*root) {

queue<TreeNode\*> tree;

int count = 0;

TreeNode \*L;

TreeNode \*R;

inorder(root, tree, L, R, count);

int temp = L->val;

L->val = R->val;

R->val = temp;

}

void inorder(TreeNode \*root, queue<TreeNode\*> &treeque, TreeNode\* &left, TreeNode\* &right, int &n) {

if (root != NULL) {

inorder(root->left, treeque, left, right, n);

treeque.push(root);

if (treeque.size() == 3) treeque.pop();

if (treeque.size() == 2) {

if (root->val < treeque.front()->val && n == 0) {

left = treeque.front();

n++;

}

if (root->val < treeque.front()->val) right = root; // this leave the possibility that right can be replaced later.

}

inorder(root->right, treeque, left, right, n);

}

}

};

class Solution {

public:

void recoverTree(TreeNode \*root) {

TreeNode \*node1;

TreeNode \*node2;

TreeNode \*prev = new TreeNode(INT\_MIN); // make sure it will not interfere the proceed of the recursion

int n = 0;

inOrder(root, node1, node2, prev, n);

int temp = node1->val;

node1->val = node2->val;

node2->val = temp;

}

void inOrder(TreeNode \*root, TreeNode\* &n1, TreeNode\* &n2, TreeNode\* &prev, int &n) {

if (root != NULL) {

inOrder(root->left, n1, n2, prev, n);

if (root->val < prev->val && n == 0) {

n1 = prev;

n++;

}

if (root->val < prev->val) n2 = root;

prev = root;

inOrder(root->right, n1, n2, prev, n);

}

}

};

/\* Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution { // can't use a primitive type to represent previous value because the special case.

public:

bool isValidBST(TreeNode \*root) {

if (root == NULL || (root->left == NULL && root->right == NULL)) return true;

bool validTree = true;

queue<int> treeVal;

inOrder(root, validTree, treeVal);

return validTree;

}

void inOrder(TreeNode \*node, bool &valid, queue<int> &que) {

if (valid && node != NULL) {

inOrder(node->left, valid, que);

if (que.empty()) que.push(node->val);

else {

if (node->val <= que.front()) valid = false;

else {

que.pop();

que.push(node->val);

}

}

inOrder(node->right, valid, que);

}

}

};

/\* Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example,

Given:

s1 = "aabcc",

s2 = "dbbca",

When s3 = "aadbbcbcac", return true.

When s3 = "aadbbbaccc", return false.

\*/

class Solution {

public:

bool isInterleave(string s1, string s2, string s3) {

if (s1.size() + s2.size() != s3.size()) return false;

int l1 = s1.size(), l2 = s2.size();

vector<vector<bool> > dp(l1 + 1, vector<bool>(l2 +1, false));

dp[0][0] = true;

for (int i = 0; i < l1; i++) {

if (s1[i] == s3[i]) dp[i + 1][0] = true;

else break;

}

for (int i = 0; i < l2; i++) {

if (s2[i] == s3[i]) dp[0][i + 1] = true;

else break;

}

for (int i = 0; i < l1; i++) {

for (int j = 0; j < l2; j++) {

if ((dp[i][j + 1] && s3[i + j +1] == s1[i]) || (dp[i + 1][j] && s3[i + j + 1] == s2[j])) dp[i + 1][j + 1] = true;

}

}

return dp[l1][l2];

}

};

/\* Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

For example:

Given binary tree {1,#,2,3},

1

\

2

/

3

return [1,3,2].

Note: Recursive solution is trivial, could you do it iteratively?

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> tree;

helper(root, tree);

return tree;

}

void helper(TreeNode \*node, vector<int> &tree) {

if (node != NULL) {

helper(node->left, tree);

tree.push\_back(node->val);

helper(node->right, tree);

}

}

};

/\* Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,

Given n = 3, there are a total of 5 unique BST's.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

\*/

class Solution {

public:

int numTrees(int n) {

if (n == 0) return 1;

else if (n == 1) return 1;

else if (n == 2) return 2;

else {

int sum = 0;

for (int i = 1; i <=n; i++) { // i is the root of the tree

sum += numTrees(i - 1) \* numTrees(n - i);

}

return sum;

}

}

};

Try dp verstion!!!

/\* Unique Binary Search Trees II

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example,

Given n = 3, your program should return all 5 unique BST's shown below.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

\*/

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

vector<TreeNode \*> generateTrees(int n) {

return helper(1, n);

}

vector<TreeNode \*> helper(int start, int end) { // vector<TreeNode\*> stores all root nodes that can be generated from integer array start-end

vector<TreeNode \*> ret;

if (start > end) {

ret.push\_back(NULL);

return ret;

}

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

vector<TreeNode \*> left = helper(start, i - 1);

vector<TreeNode \*> right = helper(i + 1, end);

for (int j = 0; j < left.size(); j++) {

for (int k = 0; k < right.size(); k++) {

TreeNode \*root = new TreeNode(i);

root->left = left[j];

root->right = right[k];

ret.push\_back(root);

}

}

}

return ret;

}

};

/\* Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

\*/

class Solution {

public:

vector<string> restoreIpAddresses(string s) {

vector<string> ret;

if (s.size() <= 3 || s.size() > 12) return ret;

vector<vector<string> > result;

vector<string> solution;

helper(result, solution, s, 0);

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

string ip;

for (int j = 0; j < 4; j++) {

ip += result[i][j] +".";

}

ip = ip.substr(0, ip.size() - 1);

ret.push\_back(ip);

}

return ret;

}

void helper(vector<vector<string> > &ret, vector<string> &sol, string &s, int i) {

if (sol.size() == 4) {

if (i == s.size()) ret.push\_back(sol);

return;

}

if (i < s.size()) {

sol.push\_back(s.substr(i, 1));

helper(ret, sol, s, i + 1);

sol.pop\_back();

}

if (i < s.size() - 1 && s[i] != '0') {

sol.push\_back(s.substr(i, 2));

helper(ret, sol, s, i + 2);

sol.pop\_back();

}

if (i < s.size() - 2 && s[i] != '0' && atoi(s.substr(i, 3).c\_str()) < 256) {

sol.push\_back(s.substr(i, 3));

helper(ret, sol, s, i + 3);

sol.pop\_back();

}

}

};

/\* Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:

Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note:

Given m, n satisfy the following condition:

1 ≤ m ≤ n ≤ length of list.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*reverseBetween(ListNode \*head, int m, int n) {

if (m == n) return head;

ListNode \*Dummy = new ListNode(0);

Dummy->next = head;

ListNode \*node = Dummy;

ListNode \*prevTail = Dummy;

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

prevTail = node;

node = node->next;

}

ListNode \*midhead = node;

ListNode \*prev = NULL;

ListNode \*curr = node;

ListNode \*next;

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

next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

}

prevTail->next = prev;

midhead->next = curr;

return Dummy->next;

}

};

/\* Subsets II

Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

For example,

If S = [1,2,2], a solution is:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

\*/

class Solution {

public:

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

vector<int> empty;

vector< vector<int> > subsets;

subsets.push\_back(empty);

if(S.size() == 0) return subsets;

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

subset\_helper(S, 0, 0, subsets);

return subsets;

}

void subset\_helper(vector<int> &nums, size\_t n, size\_t pos, vector< vector<int> > &subsets)

{

if(n == nums.size()) return;

vector<int> solution;

size\_t length = subsets.size();

size\_t i;

if(nums.size() == 1) i = 0;

else

{

if(n > 0 && nums[n] == nums[n - 1]) i = pos;

else i = 0;

}

for(; i < length; ++i)

{

solution = subsets[i];

solution.push\_back(nums[n]);

subsets.push\_back(solution);

}

subset\_helper(nums, n + 1, length, subsets);

}

};

class Solution {

public:

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

vector<vector<int> > ret;

vector<int> sol;

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

ret.push\_back(sol);

int n = 0, m = 0, preSize = 0;

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

preSize = n;

n = ret.size();

if (i != 0 && S[i] == S[i - 1]) m = preSize;

else m = 0;

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

vector<int> temp = ret[j];

temp.push\_back(S[i]);

ret.push\_back(temp);

}

}

return ret;

}

};

/\* Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,

Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

\*/

class Solution {

public:

int numDecodings(string s) {

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

return 0;

}

vector<int> nums(s.size() + 1, 0);

nums[0] = 1;

nums[1] = (s[0] == '0') ? 0 : 1;

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

if (s[i - 1] != '0') {

nums[i] = nums[i - 1];

}

int twodigit = (s[i - 2] - '0') \* 10 + (s[i - 1] - '0');

if (twodigit <= 26 && twodigit >= 10) {

nums[i] += nums[i - 2];

}

}

return nums[s.size()];

}

};

/\* Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note:

You may assume that A has enough space (size that is greater or equal to m + n) to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

\*/

class Solution {

public:

void merge(int A[], int m, int B[], int n) {

int a = m - 1;

int b = n - 1;

int c = m + n -1;

while(a >= 0 && b >= 0) {

if(A[a] >= B[b]) { A[c--] = A[a--]; }

else { A[c--] = B[b--]; }

}

if(a < 0) {

while(b >= 0) { A[c--] = B[b--];}

}

}

};

/\* Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:

00 - 0

01 - 1

11 - 3

10 - 2

Note:

For a given n, a gray code sequence is not uniquely defined.

For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

\*/

class Solution {

public:

vector<int> grayCode(int n) {

vector<int> ret;

ret.push\_back(0);

if (n == 0) return ret; // don't forget this case.

stack<int> istack;

ret.push\_back(1);

istack.push(0);

istack.push(1);

int i = 2;

while (i <= n) {

while (!istack.empty()) {

int num = istack.top();

int add = pow(2, i - 1);

istack.pop();

ret.push\_back(num + add);

}

for (int j = 0; j < ret.size(); j++) istack.push(ret[j]);

i++;

}

return ret;

}

};

/\* Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given 1->4->3->2->5->2 and x = 3,

return 1->2->2->4->3->5.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*partition(ListNode \*head, int x) {

ListNode \*Dummy = new ListNode(0);

ListNode \*Dummy2 = new ListNode(0);

Dummy2->next = head;

ListNode \*prevtail = Dummy;

ListNode \*prev = Dummy2;

while (head != NULL) {

if (head->val < x) {

prev->next = head->next;

prevtail->next = head;

head = head->next;

prevtail = prevtail->next;

} else {

prev = head;

head = head->next;

}

}

prevtail->next = Dummy2->next;

return Dummy->next;

}

};

/\* Scramble String

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":

great

/ \

gr eat

/ \ / \

g r e at

/ \

a t

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

rgeat

/ \

rg eat

/ \ / \

r g e at

/ \

a t

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

rgtae

/ \

rg tae

/ \ / \

r g ta e

/ \

t a

We say that "rgtae" is a scrambled string of "great".

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

Class Solution {

public:

bool isScramble(string s1, string s2) {

if (s1 == s2) return true;

string s11 = s1;

sort(s11.begin(), s11.end());

string s22 = s2;

sort(s22.begin(), s22.end());

if (s11 != s22) return false;

for (int i = 1; i < s1.size(); i++) {

if (isScramble(s1.substr(0, i), s2.substr(0, i))

&& isScramble(s1.substr(i, s1.size() - i), s2.substr(i, s2.size() - i))) {

return true;

}

else if (isScramble(s1.substr(0, i), s2.substr(s2.size() - i, i))

&& isScramble(s1.substr(i, s1.size() - i), s2.substr(0, s2.size() - i))) {

return true;

}

}

return false;

}

};

/\* Maximal Rectangle

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

\*/

class Solution {

public:

int maximalRectangle(vector<vector<char> > &matrix) {

if (matrix.size() == 0 || matrix[0].size() == 0) return 0;

int row = matrix.size();

int col = matrix[0].size();

int maxRecArea = 0;

vector<vector<int> > height(row, vector<int>(col));

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

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

if (matrix[i][j] == '0') height[i][j] = 0;

else height[i][j] = (i == 0) ? 1 : height[i - 1][j] + 1;

}

maxRecArea = max(maxRecArea, maxArea(height[i]));

}

return maxRecArea;

}

int maxArea(vector<int> hight) {

stack<int> stack;

int maxRec = 0;

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

int currH = (i == hight.size()) ? -1 : hight[i];

while (!stack.empty() && currH < hight[stack.top()]) {

int h = hight[stack.top()];

stack.pop();

int w = stack.empty() ? i : i - stack.top() - 1;

maxRec = max(maxRec, h \* w);

}

stack.push(i);

}

return maxRec;

}

};

/\* Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

Given 1->1->2, return 1->2.

Given 1->1->2->3->3, return 1->2->3.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

if (head == NULL || head->next == NULL) return head;

ListNode \*Dummy = new ListNode(0);

Dummy->next = head;

ListNode \*prev = head;

ListNode \*curr = head->next;

while (curr != NULL) {

if (curr->val == prev->val) prev->next = curr->next;

else prev = curr;

curr = curr->next;

}

return Dummy->next;

}

};

/\* Largest Rectangle in Histogram

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.

Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example,

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

return 10.

\*/

class Solution {

public:

int largestRectangleArea(vector<int> &height) {

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

return 0;

}

stack<int> stack;

int maxi = 0;

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

int curt = (i == height.size()) ? -1 : height[i];

while(!stack.empty() && curt <= height[stack.top()]) {

int h = height[stack.top()];

stack.pop();

int w = stack.empty() ? i : i - stack.top() - 1;

maxi = max(maxi, h \* w);

}

stack.push(i);

}

return maxi;

}

};

/\* Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,

Given 1->2->3->3->4->4->5, return 1->2->5.

Given 1->1->1->2->3, return 2->3.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*deleteDuplicates(ListNode \*head) {

if (head == NULL) {

return NULL;

}

ListNode \*Dummy = new ListNode(0);

Dummy->next = head;

head = Dummy;

while (head->next != NULL && head->next->next != NULL) {

if (head->next->val == head->next->next->val) {

int val = head->next->val;

while (head->next != NULL && head->next->val == val) {

head->next = head->next->next; // very tricky !!!

}

}

else {

head = head->next; // the val head points at is not a replicate and it is valid

}

}

return Dummy->next;

}

};

/\* Combinations

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example,

If n = 4 and k = 2, a solution is:

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

\*/

class Solution {

public:

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

vector<vector<int> > ret;

if (n == 0 || k == 0 || n < k) return ret;

vector<int> sol;

helper(ret, sol, n, k, 1);

return ret;

}

void helper(vector<vector<int> > &result, vector<int> &sol, int n, int k, int pos) {

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

result.push\_back(sol);

return;

}

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

sol.push\_back(i);

helper(result, sol, n, k, i + 1);

sol.pop\_back();

}

}

};

/\* Subsets

Given a set of distinct integers, S, return all possible subsets.

Note:

Elements in a subset must be in non-descending order.

The solution set must not contain duplicate subsets.

For example,

If S = [1,2,3], a solution is:

[

[3],

[1],

[2],

[1,2,3],

[1,3],

[2,3],

[1,2],

[]

]

\*/

class Solution {

public:

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

vector<int> empty;

vector< vector<int> > subsets;

subsets.push\_back(empty);

if(S.size() == 0) return subsets;

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

subset\_helper(S, 0, subsets);

return subsets;

}

void subset\_helper(vector<int> &nums, size\_t n, vector< vector<int> > &subsets)

{

if(n == nums.size()) return;

vector<int> solution;

size\_t length = subsets.size();

for(size\_t i = 0; i < length; ++i)

{

solution = subsets[i];

solution.push\_back(nums[n]);

subsets.push\_back(solution);

}

subset\_helper(nums, n + 1, subsets);

}

};

/\* Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,

Given board =

[

["ABCE"],

["SFCS"],

["ADEE"]

]

word = "ABCCED", -> returns true,

word = "SEE", -> returns true,

word = "ABCB", -> returns false.

\*/

class Solution {

public:

bool exist(vector<vector<char> > &board, string word) {

if (board.size() == 0 || board[0].size() == 0 || word.size() == 0) return false;

vector<vector<char> > visit(board.size(), vector<char>(board[0].size(), 'U'));

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

for (int j = 0; j < board[0].size(); j++) {

if (board[i][j] == word[0]) {

if (dfs(board, visit, i, j, word, 0)) return true;

}

}

}

return false;

}

bool dfs(vector<vector<char> > &board, vector<vector<char> > &visited, int row, int col, string word, int n) {

if (n == word.size()) {

return true;

}

if (row >= 0 && row < board.size() && col >= 0 && col < board[0].size() && visited[row][col] != 'V') {

if (board[row][col] == word[n]) {

visited[row][col] = 'V';

if (dfs(board, visited, row + 1, col, word, n + 1)) return true;

else if (dfs(board, visited, row - 1, col, word, n + 1)) return true;

else if (dfs(board, visited, row, col + 1, word, n + 1)) return true;

else if (dfs(board, visited, row, col - 1, word, n + 1)) return true;

visited[row][col] = 'U';

return false;

}

}

return false;

}

};

/\* Minimum Window Substring

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).

For example,

S = "ADOBECODEBANC"

T = "ABC"

Minimum window is "BANC".

Note:

If there is no such window in S that covers all characters in T, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

\*/

class Solution {

public:

string minWindow(string S, string T) {

int lens = S.size(), lent = T.size();

queue<int> Q;

int srcCnt[128] = {0};//T中每个字母的个数

int foundCnt[128] = {0};//当前找到T中每个字母的个数

for(int i = 0; i < lent; i++) srcCnt[T[i]]++;

int hasFound = 0;//已经找到的字母数目

int winStart = -1, winEnd = lens;//窗口的左右边界

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

if(srcCnt[S[i]]) { // 找到一个在T中的字母

Q.push(i);

foundCnt[S[i]]++;

if(foundCnt[S[i]] <= srcCnt[S[i]]) hasFound++;

if(hasFound == lent) {//找到了一个满足的窗口, 右边界是i

int k;

do {//缩减左窗口到最小

k = Q.front();

Q.pop();

foundCnt[S[k]]--;

} while(srcCnt[S[k]] <= foundCnt[S[k]]); //判断当前字母是不是必须的，如果不是必需的，左窗口右移

if(winEnd - winStart > i - k) {

winStart = k;

winEnd = i;

}

hasFound--; // 当前有效窗口最左边的一个字母被去掉了

}

}

return winStart != -1 ? S.substr(winStart, winEnd - winStart +1) : "";

}

};

/\* Sort Colors

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.

Note:

You are not suppose to use the library's sort function for this problem.

click to show follow up.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

\*/

class Solution {

public:

void sortColors(int A[], int n) {

int i = 0, j = n - 1;

while (i < j) {

if (A[i] > 0 && A[j] == 0) {

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

i++;

j--;

} else {

if (A[i] == 0) i++;

if (A[j] > 0) j--;

}

}

if (A[i] == 0) i++; // in case the input is [0,0,1].

//i points to the first non-zero number in the array or out of array.

if (i < n - 1) {

j = n - 1;

while (i < j) {

if (A[i] == 2 && A[j] == 1) {

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

i++;

j--;

} else {

if (A[i] == 1) i++;

if (A[j] == 2) j--;

}

}

}

}

};

/\* Search a 2D Matrix

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.

For example,

Consider the following matrix:

[

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

Given target = 3, return true.

\*/

class Solution {

public:

bool searchMatrix(vector<vector<int> > &matrix, int target) {

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

return false;

}

int row = matrix.size();

int col = matrix[0].size();

int start = 0;

int mid;

int end = row \* col -1;

while(start + 1 < end) {

mid = start + (end - start) / 2;

int mid\_val = matrix[mid / col][mid % col];

if(mid\_val == target) {

return true;

}

else if(mid\_val > target) {

end = mid;

}

else {

start = mid;

}

}

if(matrix[start / col][start % col] == target || matrix[end / col][end % col] == target) {

return true;

}

return false;

}

};

/\* Set Matrix Zeroes

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

click to show follow up.

Follow up:

Did you use extra space?

A straight forward solution using O(mn) space is probably a bad idea.

A simple improvement uses O(m + n) space, but still not the best solution.

Could you devise a constant space solution?

\*/

class Solution {

public:

void setZeroes(vector<vector<int> > &matrix) {

int m = matrix.size();

int n = matrix[0].size();

bool r = false, c = false;

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

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

if (matrix[i][j] == 0) {

if (i == 0) r = true;

if (j == 0) c = true;

matrix[i][0] = 0;

matrix[0][j] = 0;

}

}

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

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

if (matrix[0][j] == 0 || matrix[i][0] == 0) matrix[i][j] = 0;

}

if (r) for (int i = 0; i < n; i++) matrix[0][i] = 0;

if (c) for (int i = 0; i < m; i++) matrix[i][0] = 0;

}

};

/\* Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character

b) Delete a character

c) Replace a character

\*/

class Solution {

public:

int minDistance(string word1, string word2) {

int n1 = word1.size();

int n2 = word2.size();

vector<vector<int> > steps(n1 + 1, vector<int>(n2 +1));

for (int i = 0; i <= n1; ++i) {

steps[i][0] = i;

}

for (int j = 0; j <= n2; ++j) {

steps[0][j] = j;

}

for (int i = 1; i <= n1; ++i) {

for (int j = 1; j <= n2; ++j) {

if (word1[i - 1] == word2[j - 1]) {

steps[i][j] = steps[i - 1][j - 1];

}

else {

steps[i][j] = min(steps[i - 1][j - 1], min(steps[i-1][j], steps[i][j - 1])) + 1;

}

}

}

return steps[n1][n2];

}

};

/\* Sqrt(x)

Implement int sqrt(int x).

Compute and return the square root of x.

\*/

class Solution {

public:

int sqrt(int x) {

unsigned int res = 0;

for (int i = 15; i >= 0; --i) {

if ((res + (1 << i)) \* (res + (1 << i)) <= x) {

res = res + (1 << i);

}

}

return res;

}

};

/\* Simplify Path

1) skip consecutive '/'

2) record every word between two '/'s

3) if the word is "..", pop the last word in vector.

4) if the word is ".", skip it.

5) if the word is others, push it into vector.

6) put '/' back between words except the last position.

Given an absolute path for a file (Unix-style), simplify it.

For example,

path = "/home/", => "/home"

path = "/a/./b/../../c/", => "/c"

click to show corner cases.

Corner Cases:

Did you consider the case where path = "/../"?

In this case, you should return "/".

Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/".

In this case, you should ignore redundant slashes and return "/home/foo".

\*/

class Solution {

public:

string simplifyPath(string path) {

string simplePath;

if (path.size() == 0 || path[0] != '/') return simplePath;

vector<string> elements;

int i = 0;

while (i < path.size()) {

while (path[i] == '/' && i < path.size()) i++;

if (i == path.size()) break;

int start = i;

while (path[i] != '/' && i < path.size()) i++;

string element = path.substr(start, i - start);

if (element == "..") {

if (elements.size() > 0) elements.pop\_back();

} else if (element != ".") {

elements.push\_back(element);

}

}

if (elements.size() == 0) return "/";

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

simplePath += "/" + elements[i];

}

return simplePath;

}

};

/\* Text Justification

class Solution {

public:

vector<string> fullJustify(vector<string> &words, int L) {

vector<string> ret;

if (words.size() == 0) return ret;

int i = 0;

while (i < words.size()) {

int start = i, end;

int len = words[i].size();

while (i < words.size() - 1 && len <= L) {

i++;

len += words[i].size() + 1;

}

if (i == words.size() - 1 && len <= L) end = i; // the last line

else end = i - 1;

// organize words in one line.

justifyWords(words, ret, L, start, end);

i = end + 1;

}

return ret;

}

void justifyWords(vector<string> &words, vector<string> &ret, int L, int s, int e) {

string sol(L, ' ');

if (s == e) { // only one word in this line

sol.replace(0, words[s].size(), words[s]);

ret.push\_back(sol);

return;

}

else if (e == words.size() - 1) { // the last line

int pos = 0;

for (int i = s; i <= e; i++) {

sol.replace(pos, words[i].size(), words[i]);

pos += words[i].size() + 1;

}

ret.push\_back(sol);

return;

}

int num\_slot = e - s;

int num\_space = L;

for (int i = s; i <= e; i++) num\_space -= words[i].size();

int quotient = num\_space / num\_slot;

int mode = num\_space % num\_slot;

int pos = 0, space;

for (int i = s; i <= e; i++) {

sol.replace(pos, words[i].size(), words[i]);

if (mode != 0) {

space = quotient + 1;

mode--;

} else space = quotient;

pos += words[i].size() + space;

}

ret.push\_back(sol);

return;

}

};

/\* Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

\*/

class Solution {

public:

int climbStairs(int n) {

if (n == 0) {

return 0;

}

vector<int> ways(n, 0);

ways[0] = 1;

ways[1] = 2;

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

ways[i] = ways[i - 1] + ways[i - 2];

}

return ways[n - 1];

}

};

/\* Add Binary

Given two binary strings, return their sum (also a binary string).

For example,

a = "11"

b = "1"

Return "100".

\*/

class Solution {

public:

string addBinary(string a, string b) {

if (a.size() == 0) return b;

if (b.size() == 0) return a;

int l1 = a.size() - 1, l2 = b.size() - 1;

string ret(max(l1, l2) + 2, 0);

int size = ret.size() - 1;

int carry = 0;

while (l1 >= 0 && l2 >= 0) {

int sum = a[l1] - '0' + b[l2] - '0' + carry;

if (sum == 0) {

carry = 0;

ret[size] = '0';

} else if (sum == 1) {

carry = 0;

ret[size] = '1';

} else if (sum == 2) {

carry = 1;

ret[size] = '0';

} else if (sum == 3) {

carry = 1;

ret[size] = '1';

}

l1--; l2--; size--;

}

if (l1 == -1) {

while (l2 >= 0) {

int sum = b[l2] - '0' + carry;

if (sum == 0) {

carry = 0;

ret[size] = '0';

} else if (sum == 1) {

carry = 0;

ret[size] = '1';

} else if (sum == 2) {

carry = 1;

ret[size] = '0';

}

l2--; size--;

}

}

if (l2 == -1) {

while (l1 >= 0) {

int sum = a[l1] - '0' + carry;

if (sum == 0) {

carry = 0;

ret[size] = '0';

} else if (sum == 1) {

carry = 0;

ret[size] = '1';

} else if (sum == 2) {

carry = 1;

ret[size] = '0';

}

l1--; size--;

}

}

if (carry == 1) ret[0] = '1';

else ret = ret.substr(1, ret.size() - 1);

return ret;

}

};

/\* Plus One

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.

\*/

class Solution {

public:

vector<int> plusOne(vector<int> &digits) {

for (int i = digits.size() - 1; i >= 0; i--) {

if (digits[i] != 9) {

digits[i]++;

return digits;

}

else {

if (i == 0) {

digits[0] = 0;

digits.insert(digits.begin(), 1);

return digits;

} else digits[i] = 0;

}

}

}

};

class Solution {

public:

vector<int> plusOne(vector<int> &digits) {

int carry = 1;

int i = digits.size() - 1;

while (i >= 0) {

int sum = digits[i] + carry;

if (sum > 9) {

carry = 1;

digits[i--] = 0;

}

else {

digits[i] = sum;

return digits;

}

}

if (i == -1 && carry == 1) digits.insert(digits.begin(), 1);

return digits;

}

};

/\* Valid Number

Validate if a given string is numeric.

Some examples:

"0" => true

" 0.1 " => true

"abc" => false

"1 a" => false

"2e10" => true

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

\*/

class Solution {

public:

bool isNumber(const char \*s) {

enum inputType {INVALID, SPACE, SIGN, DIGIT, DOT, EXPONENT};

int transitionTable[9][6] = {

//0INVA,1SPA,2SIG,3DI,4DO,5E

-1, 0, 3, 1, 2, -1,//0初始无输入或者只有space的状态

-1, 8, -1, 1, 4, 5,//1输入了数字之后的状态

-1, -1, -1, 4, -1, -1,//2前面无数字，只输入了Dot的状态

-1, -1, -1, 1, 2, -1,//3输入了符号状态

-1, 8, -1, 4, -1, 5,//4前面有数字和有dot的状态

-1, -1, 6, 7, -1, -1,//5'e' or 'E'输入后的状态

-1, -1, -1, 7, -1, -1,//6输入e之后输入Sign的状态

-1, 8, -1, 7, -1, -1,//7输入e后输入数字的状态

-1, 8, -1, -1, -1, -1,//8前面有有效数输入之后，输入space的状态

}; // do not forget this ;

int state = 0;

while (\*s) {

inputType input = INVALID;

if (\*s == ' ') input = SPACE;

else if (\*s == '-' || \*s == '+') input = SIGN;

else if (isdigit(\*s)) input = DIGIT;

else if (\*s == '.') input = DOT;

else if (\*s == 'e' || \*s == 'E') input = EXPONENT;

state = transitionTable[state][input];

if (state == -1) return false;

++s; // do not forget this

}

return state == 1 || state == 4 || state == 7 || state == 8;

}

};

/\* Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

if (l1 == NULL) {

return l2;

}

if (l2 == NULL) {

return l1;

}

ListNode \*Dummy = new ListNode(0);

ListNode \*node = Dummy;

while (l1 != NULL && l2 != NULL) {

if (l1->val < l2->val) {

node->next = l1;

l1 = l1->next;

}

else {

node->next = l2;

l2 = l2->next;

}

node = node->next;

}

if (l1 == NULL) {

node->next = l2;

}

else {

node->next = l1;

}

return Dummy->next;

}

};

/\* Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

\*/

class Solution {

public:

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

if (grid.size() == 0 || grid[0].size() == 0) return 0;

int m = grid.size();

int n = grid[0].size();

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

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

for (int i = 1; i < n; i++) dp[0][i] = dp[0][i - 1] + grid[0][i];

for (int i = 1; i < m; i++) dp[i][0] = dp[i - 1][0] + grid[i][0];

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

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

dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];

}

}

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

}

};

/\* Unique Paths

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

\*/

class Solution {

public:

int uniquePaths(int m, int n) {

if (m == 0 || n == 0) return 0;

vector<vector<int> > dp(m, vector<int>(n));

for (int i = 0; i < m; i++) dp[i][0] = 1;

for (int i = 0; i < n; i++) dp[0][i] = 1;

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

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

dp[i][j] = dp[i - 1][j] + dp[i][j - 1];

}

}

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

}

};

class Solution {

public:

int uniquePaths(int m, int n) {

vector<int> paths(n, 0);

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

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

if (i == 0) {

paths[j] = 1;

}

else {

if (j == 0) {

paths[j] = 1;

}

else {

paths[j] = paths[j - 1] + paths[j];

}

}

}

}

return paths[n - 1];

}

};

/\* Unique Paths II

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

[

[0,0,0],

[0,1,0],

[0,0,0]

]

The total number of unique paths is 2.

Note: m and n will be at most 100.

\*/

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {

if (obstacleGrid.size() == 0) return 0;

int m = obstacleGrid.size(), n = obstacleGrid[0].size();

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

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

if (obstacleGrid[i][0] == 1) break;

else dp[i][0] = 1;

}

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

if (obstacleGrid[0][i] == 1) break;

else dp[0][i] = 1;

}

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

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

if (obstacleGrid[i][j] == 1) dp[i][j] = 0;

else dp[i][j] = dp[i - 1][j] + dp[i][j - 1];

}

}

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

}

};

/\* Permutation Sequence

The set [1,2,3,…,n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order,

We get the following sequence (ie, for n = 3):

"123"

"132"

"213"

"231"

"312"

"321"

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

\*/

class Solution {

public:

string getPermutation(int n, int k) {

string ret;

vector<int> digits;

for (int i = 1; i <= n; i++) digits.push\_back(i);

while (digits.size()) {

int rank = (k - 1) / factorial(n - 1) + 1;

int digt = digits[rank - 1];

ret.push\_back(digt + '0');

n--;

k -= (rank - 1) \* factorial(n);

digits.erase(digits.begin() + rank - 1);

}

return ret;

}

int factorial(int x) {

if (x == 1 || x == 0) return 1;

else return x \* factorial(x - 1);

}

};

/\* Spiral Matrix II

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

For example,

Given n = 3,

You should return the following matrix:

[

[ 1, 2, 3 ],

[ 8, 9, 4 ],

[ 7, 6, 5 ]

]

\*/

class Solution {

public:

vector<vector<int> > generateMatrix(int n) {

vector<vector<int> > ret(n, vector<int>(n));

int x1 = 0, y1 = 0, x2 = n - 1, y2 = n - 1;

int num = 1;

while (x1 <= x2) {

if (x1 + 1 == x2) {

ret[x1][y1] = num++;

ret[x1][y2] = num++;

ret[x2][y2] = num++;

ret[x2][y1] = num++;

} else if (x1 == x2) {

ret[x1][x1] = num;

} else {

for (int i = y1; i <= y2; i++) ret[x1][i] = num++;

for (int i = x1 + 1; i <= x2; i++) ret[i][y2] = num++;

for (int i = y2 - 1; i >= y1; i--) ret[x2][i] = num++;

for (int i = x2 - 1; i >= x1 + 1; i--) ret[i][y1] = num++;

}

x1++; y1++; x2--; y2--;

}

return ret;

}

};

class Solution {

public:

vector<vector<int> > generateMatrix(int n) {

vector<vector<int> > result;

if (n == 0) return result;

vector<vector<int> > rest(n, vector<int>(n));

int c = 1;

spiral(rest, c, n, 0, 0, n - 1, n - 1);

return rest;

}

void spiral(vector<vector<int> > &ret, int &count, int n, int x1, int y1, int x2, int y2) {

if (count == n \* n + 1) return;

for (int i = y1; i <= y2; i++) {

ret[x1][i] = count;

count++;

}

for (int i = x1 + 1; i <= x2; i++) {

ret[i][y2] = count;

count++;

}

if (x1 != x2) {

for (int i = y2 - 1; i >= y1; i--) {

ret[x2][i] = count;

count++;

}

}

if (y1 != y2) {

for (int i = x2 - 1; i >= x1 +1; i--) {

ret[i][y1] = count;

count++;

}

}

spiral(ret, count, n, x1 + 1, y1 + 1, x2 - 1, y2 - 1);

}

};

/\* Length of Last Word

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,

Given s = "Hello World",

return 5.

\*/

class Solution {

public:

int lengthOfLastWord(const char \*s) {

if (s == NULL || s[0] == '\0') return 0;

int len = 0;

int i = strlen(s) - 1;

while (i >= 0 && s[i] == ' ') {

i--;

}

while (i >= 0 && s[i] != ' ') {

len++;

i--;

}

return len;

}

};

/\* Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:

Given 1->2->3->4->5->NULL and k = 2,

return 4->5->1->2->3->NULL.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*rotateRight(ListNode \*head, int k) {

if (head == NULL || head->next == NULL || k == 0) return head;

int nums = 1;

ListNode \*node = head;

while (node->next != NULL) {

nums++;

node = node->next;

}

if (k % nums == 0) return head;

k = k % nums;

ListNode \*tail = node;

int rotateNum = nums - k;

int i = 1;

node = head;

while (i < rotateNum) {

node = node->next;

i++;

}

tail->next = head;

head = node->next;

node->next = NULL;

return head;

}

};

/\* Insert Interval

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.

Example 1:

Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2:

Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

\*/

/\*\*

\* Definition for an interval.

\* struct Interval {

\* int start;

\* int end;

\* Interval() : start(0), end(0) {}

\* Interval(int s, int e) : start(s), end(e) {}

\* };

\*/

class Solution {

public:

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

vector<Interval> result;

int i = 0;

while (i < intervals.size() && intervals[i].end < newInterval.start)

result.push\_back(intervals[i++]);

while (i < intervals.size() && intervals[i].start <= newInterval.end){

if (intervals[i].start < newInterval.start)

newInterval.start = intervals[i].start;

if (intervals[i].end > newInterval.end)

newInterval.end = intervals[i].end;

i++;

}

result.push\_back(newInterval);

while (i < intervals.size() && intervals[i].start > newInterval.end)

result.push\_back(intervals[i++]);

return result;

}

};

/\* Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,

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

return [1,6],[8,10],[15,18].

\*/

/\*\*

\* Definition for an interval.

\* struct Interval {

\* int start;

\* int end;

\* Interval() : start(0), end(0) {}

\* Interval(int s, int e) : start(s), end(e) {}

\* };

\*/

class Solution {

public:

vector<Interval> merge(vector<Interval> &intervals) {

vector<Interval> ret;

vector<Interval> tempv;

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

tempv.clear();

Interval temp = intervals[i];

int j = 0;

while (j < ret.size() && temp.start > ret[j].end) tempv.push\_back(ret[j++]);

while (j < ret.size() && temp.end >= ret[j].start) {

temp.start = min(temp.start, ret[j].start);

temp.end = max(temp.end, ret[j].end);

j++;

}

tempv.push\_back(temp);

while (j < ret.size()) tempv.push\_back(ret[j++]);

ret = tempv;

}

return ret;

}

};

/\* Spiral Matrix

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

For example,

Given the following matrix:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

You should return [1,2,3,6,9,8,7,4,5].

\*/

class Solution {

public:

vector<int> spiralOrder(vector<vector<int> > &matrix) {

vector<int> result;

if (matrix.size() == 0 || matrix[0].size() == 0) return result;

spiral(result, matrix, 0, 0, matrix.size() - 1, matrix[0].size() - 1);

return result;

}

void spiral(vector<int> &ret, vector<vector<int> > &matrix, int x1, int y1, int x2, int y2) {

if (x1 > x2 || y1 > y2) return;

for (int i = y1; i <= y2; i++) {

ret.push\_back(matrix[x1][i]);

}

for (int i = x1 + 1; i <= x2; i++) {

ret.push\_back(matrix[i][y2]);

}

if (x1 != x2) {

for (int i = y2 - 1; i >= y1; i--) {

ret.push\_back(matrix[x2][i]);

}

}

if (y1 != y2) {

for (int i = x2 - 1; i >= x1 + 1; i--) {

ret.push\_back(matrix[i][y1]);

}

}

spiral(ret, matrix, x1 + 1, y1 + 1, x2 - 1, y2 - 1);

}

};

/\* Jump Game

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.

For example:

A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.

\*/

class Solution {

public:

bool canJump(int A[], int n) {

if (n == 0) {

return false;

}

vector<bool> isPossible(n, false);

isPossible[0] = true;

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

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

if (isPossible[j] && (j + A[j]) >= i) {

isPossible[i] = true;

break;

}

}

}

return isPossible[n - 1];

}

};

/\* Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4],

the contiguous subarray [4,−1,2,1] has the largest sum = 6.

click to show more practice.

More practice:

If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

\*/

class Solution {

public:

int maxSubArray(int A[], int n) {

if (A == NULL || n < 1) {

return 0;

}

int Max = INT\_MIN, sum = 0;

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

sum += A[i];

Max = (Max > sum ? Max : sum);

sum = (sum > 0 ? sum : 0);

}

return Max;

}

};

class Solution {

public:

int maxSubArray(int A[], int n) {

if (A == NULL || n < 1) {

return 0;

}

return Maxsubarray(A, 0, n - 1);

}

int Maxsubarray(int A[], int low, int high) {

if (low == high) {

return A[low];

}

else {

int mid = (low + high) / 2;

int left\_sum = Maxsubarray(A, low, mid); // not mid - 1 because this will lose one member

int right\_sum = Maxsubarray(A, mid + 1, high);

int crossing\_sum = maxCrossingSubArray(A, low, mid, high);

if (left\_sum >= right\_sum && left\_sum >= crossing\_sum) {

return left\_sum;

}

else if (crossing\_sum >= left\_sum && crossing\_sum >= right\_sum) {

return crossing\_sum;

}

else {

return right\_sum;

}

}

}

int maxCrossingSubArray(int A[], int low, int mid, int high) {

int left\_sum = INT\_MIN;

int sum = 0;

for(int i = mid; i >= low; --i) {

sum += A[i];

if (sum > left\_sum) {

left\_sum = sum;

}

}

int right\_sum = INT\_MIN;

sum = 0;

for (int i = mid + 1; i <= high; ++i) {

sum += A[i];

if (sum > right\_sum) {

right\_sum = sum;

}

}

return left\_sum + right\_sum;

}

};

class Solution {

public:

int maxSubArray(int A[], int n) {

if (A == NULL || n == 0) {

return 0;

}

int\* sum = new int[n + 1];

sum[0] = 0;

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

sum[i] = sum[i - 1] + A[i - 1];

}

int result = sum[1];

int Min = sum[0];

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

result = max(result, sum[i] - Min);

Min = min(Min, sum[i]);

}

delete[] sum;

return result;

}

};

/\* N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

\*/

class Solution {

public:

int totalNQueens(int n) {

vector<vector<int> > board(n, vector<int>(n, 0));

int ret = 0;

dfs(board, 0, ret);

return ret;

}

void dfs(vector<vector<int> > &board, int level, int &ret) {

if (level == board.size()) ret++;

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

if (isValid(board, level, i)) {

board[level][i] = 1;

dfs(board, level + 1, ret);

board[level][i] = 0;

}

}

}

bool isValid(vector<vector<int> > &board, int x, int y) {

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

if (board[i][y] == 1) return false;

}

for (int i = x - 1, j = y - 1; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == 1) return false;

}

for (int i = x - 1, j = y + 1; i >= 0 && j < board.size(); i--, j++) {

if (board[i][j] == 1) return false;

}

return true;

}

};

/\* The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

For example,

There exist two distinct solutions to the 4-queens puzzle:

[

[".Q..", // Solution 1

"...Q",

"Q...",

"..Q."],

["..Q.", // Solution 2

"Q...",

"...Q",

".Q.."]

]

\*/

class Solution {

public:

vector<vector<string> > solveNQueens(int n) {

vector<vector<string> > result;

vector<string> solution(n, string(n, '.'));

helper(result, solution, 0);

return result;

}

void helper(vector<vector<string> > &result, vector<string> &solution, int row) {

if (row == solution.size()) {

result.push\_back(solution);

return;

}

for (vector<string>::size\_type i = 0; i < solution.size(); ++i) {

if (isValid(solution, row, i)) {

solution[row][i] = 'Q';

helper(result, solution, row + 1);

solution[row][i] = '.';

}

}

}

bool isValid(vector<string> &solution, int row, int col) {

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

if (solution[i][col] == 'Q') {

return false;

}

}

for (int i = row - 1, j = col - 1; i >=0 && j >= 0; --i, --j) {

if (solution[i][j] == 'Q'){

return false;

}

}

for (int i = row - 1, j = col + 1; i >=0 && j < solution.size(); --i, ++j) {

if (solution[i][j] == 'Q'){

return false;

}

}

return true;

}

};

/\* Anagrams

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

\*/

class Solution {

public:

vector<string> anagrams(vector<string> &strs) {

vector<string> ret;

unordered\_map<string,int> words\_count;

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

string str = strs[i];

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

words\_count[str]++;

}

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

string str = strs[i];

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

if (words\_count[str] >= 2) ret.push\_back(strs[i]);

}

return ret;

}

};

/\* Pow(x, n)

Implement pow(x, n).

\*/

class Solution {

public:

double pow(double x, int n) {

int m = abs(n);

if (n >= 0) {

return helper(x, n);

} else {

return 1 / helper(x, m);

}

}

double helper(double x, int n) {

if (n == 0) return 1;

if (n % 2 == 0) {

double ret = pow(x, n / 2);

return ret \* ret;

} else {

double ret = pow(x, (n - 1) / 2);

return ret \* ret \* x;

}

}

};

Find better version!!!

/\* Rotate Image

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:

Could you do this in-place?

\*/

class Solution {

public:

void rotate(vector<vector<int> > &matrix) {

reverse(matrix.begin(), matrix.end());

for (unsigned i = 0; i < matrix.size(); ++i)

for (unsigned j = i+1; j < matrix[i].size(); ++j)

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

}

};

/\* Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

[1,1,2] have the following unique permutations:

[1,1,2], [1,2,1], and [2,1,1].

\*/

class Solution {

public:

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

vector<vector<int> > coll;

vector<int> solution;

if(num.size() ==0) return coll;

vector<int> visited(num.size(), 0);

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

GeneratePermute(num, visited, solution, coll);

return coll;

}

void GeneratePermute(vector<int> & num, vector<int>& visited, vector<int>& solution, vector<vector<int> >& coll)

{

if(solution.size() == num.size())

{

coll.push\_back(solution);

return;

}

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

{

if(visited[i] == 0)

{

if(i > 0 && num[i] == num[i-1] && visited[i-1] == 0)

continue;

visited[i] = 1;

solution.push\_back(num[i]);

GeneratePermute(num, visited, solution, coll);

solution.pop\_back();

visited[i] =0;

}

}

}

};

/\* Permutations

Given a collection of numbers, return all possible permutations.

For example,

[1,2,3] have the following permutations:

[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

\*/

class Solution {

public:

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

vector< vector<int> > ret;

vector<int> solution;

if(num.size() == 0) return ret;

recursionHelper(ret, solution, num);

return ret;

}

void recursionHelper(vector< vector<int> > &ret, vector<int> &solution, const vector<int> &num)

{

if(solution.size() == num.size())

{

ret.push\_back(solution);

return;

}

for(size\_t i = 0; i < num.size(); ++i )

{

if(find(solution.begin(),solution.end(),num[i]) != solution.end())

continue;

solution.push\_back(num[i]);

recursionHelper(ret, solution, num);

solution.pop\_back();

}

}

};

class Solution {

public:

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

vector< vector<int> > ret;

if(num.size() == 0) return ret;

recursionHelper(ret, 0, num);

return ret;

}

void recursionHelper(vector< vector<int> > &ret, int cur, vector<int> &num)

{

if(cur == num.size())

{

ret.push\_back(num);

return;

}

for(size\_t i = cur; i < num.size(); ++i )

{

swap(num[cur], num[i]);

recursionHelper(ret, cur + 1, num);

swap(num[cur], num[i]);

}

}

};

class Solution {

public:

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

vector<vector<int> > result;

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

return result;

}

vector<int> solution;

helper(result, solution, nums);

return result;

}

void helper(vector<vector<int> > &ret, vector<int> &sol, const vector<int> &input) {

if (sol.size() == input.size()) {

ret.push\_back(sol);

return;

}

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

if (find(sol.begin(), sol.end(), input[i]) != sol.end()) {

continue;

}

sol.push\_back(input[i]);

helper(ret, sol, input);

sol.pop\_back();

}

}

};

class Solution {

public:

vector<vector<int> > permute(vector<int> num) {

vector<vector<int> > result;

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

helper(result, 0, num);

return result;

}

void helper(vector<vector<int> > &ret, int curr, vector<int> &num) {

if (curr == num.size()) {

ret.push\_back(num);

return;

}

for (int i = curr; i < num.size(); i++) {

swap(num[curr], num[i]);

helper(ret, curr + 1, num);

swap(num[curr], num[i]);

}

}

};

/\* Jump Game II

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.

For 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.)

\*/

class Solution {

public:

int jump(int A[], int n) {

if (n == 1) return 0;

vector<int> minStep(n, -1);

minStep[0] = 0;

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

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

if (j + A[j] >= i && minStep[j] != -1) {

minStep[i] = minStep[j] + 1;

break;

}

}

}

return minStep[n - 1];

}

};

/\* Wildcard Matching

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).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

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

\*/

class Solution {

public:

bool isMatch(const char \*s, const char \*p) {

const char\* star=NULL;

const char\* ss=s;

while (\*s){

if ((\*p=='?')||(\*p==\*s)){s++;p++;continue;}

if (\*p=='\*'){star=p++; ss=s;continue;}

if (star){ p = star+1; s=++ss;continue;}

return false;

}

while (\*p=='\*'){p++;}

return !\*p;

}

};

/\* Multiply Strings

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

\*/

class Solution {

public:

string multiply(string num1, string num2) {

int n1 = num1.size(), n2 = num2.size();

vector<int> tmpres(n1+n2, 0);

int k = n1 + n2 - 2;

for(int i = 0; i < n1; i++)

for(int j = 0; j < n2; j++)

tmpres[k-i-j] += (num1[i]-'0')\*(num2[j]-'0');

int carryBit = 0;

for(int i = 0; i < n1+n2; i++)//处理进位

{

tmpres[i] += carryBit;

carryBit = tmpres[i] / 10;

tmpres[i] %= 10;

}

int i = k+1;

while(tmpres[i] == 0)i--;//去掉乘积的前导0

if(i < 0)return "0"; //注意乘积为0的特殊情况

string res;

for(; i >= 0; i--)

res.push\_back(tmpres[i] + '0');

return res;

}

};

/\* Trapping Rain Water

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.

For example,

Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. Thanks Marcos for contributing this image!

\*/

class Solution {

public:

int trap(int A[], int n) {

if (A == NULL || n < 3) return 0;

//find the left largest

int max = A[0];

int\* lmax = new int[n];

lmax[0] = 0;

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

lmax[i] = max;

if (A[i] > max) max = A[i];

}

//find the right largest and calculate trap rain for each column

int rmax = A[n - 1];

int trap = 0;

for (int i = n - 2; i > 0; --i) {

if (min(rmax, lmax[i]) > A[i]) {

trap += min(rmax, lmax[i]) - A[i];

}

if (A[i] > rmax) rmax = A[i];

}

delete lmax;

return trap;

}

};

/\* First Missing Positive

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

For example,

Given [1,2,0] return 3,

and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

\*/

class Solution {

public:

int firstMissingPositive(int A[], int n) {

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

int num = A[i];

while (num <= n && num > 0 && A[num - 1] != num) {

swap(A[num - 1], A[i]); // after swap, the number at i may violate "the rule", we have to check it again until everything follows the rule

num = A[i];

}

}

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

if (A[i] != i + 1){

return i + 1;

}

}

return n + 1;

}

};

/\* Combination Sum II

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.

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]

\*/

class Solution {

public:

vector<vector<int> > combinationSum2(vector<int> &num, int target) {

vector<vector<int> > ret;

vector<int> sol;

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

int i;

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

if (num[i] > target) break;

}

dfs(ret, sol, num, target, 0, 0, i);

return ret;

}

void dfs(vector<vector<int> > &ret, vector<int> &sol, vector<int> &num, int target, int sum, int pos, int end) {

if (sum > target) return;

if (sum == target) {

ret.push\_back(sol);

return;

}

for (int i = pos; i < end; i++) {

sol.push\_back(num[i]);

dfs(ret, sol, num, target, sum + num[i], i + 1, end);

sol.pop\_back();

while (i < end - 1 && num[i] == num[i + 1]) i++; //this is to avoid duplicate combination

}

}

};

/\* Combination Sum @@@@@@@@@@@@@@@@@@@@@@@ special example of dfs

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 ≤ a2 ≤ … ≤ ak).

The solution set must not contain duplicate combinations.

For example, given candidate set 2,3,6,7 and target 7,

A solution set is:

[7]

[2, 2, 3]

\*/

class Solution {

public:

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

vector<vector<int> > result;

vector<int> solution;

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

int end;

for (end = 0; end < candidates.size(); ++end) {

if (candidates[end] > target) {

break;

}

}

int sum = 0;

helper(candidates, result, solution, 0, sum, target, end);

return result;

}

void helper(vector<int> &candidate, vector<vector<int> > &result, vector<int> &sol, int level, int &sum, int target, int size) {

if (sum == target) {

result.push\_back(sol);

return;

}

if (sum > target) {

return;

}

for (int i = level; i < size; ++i) {

sum += candidate[i];

sol.push\_back(candidate[i]);

helper(candidate, result, sol, i, sum, target, size);

sum -= candidate[i];

sol.pop\_back();

}

}

};

/\* Count and Say

The count-and-say sequence is the sequence of integers beginning as follows:

1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.

11 is read off as "two 1s" or 21.

21 is read off as "one 2, then one 1" or 1211.

Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

\*/

class Solution {

public:

string countAndSay(int n) {

if (n == 1) return "1";

string prev = "1";

string curr;

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

curr = "";

int j = 0;

while (j < prev.size()) {

char c = prev[j];

int count = 1;

while (++j < prev.size() && prev[j] == c) count++;

curr += to\_string(count) + c;

}

prev = curr;

}

return curr;

}

};

/\* Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.

\*/

class Solution {

public:

typedef map<pair<int, int>, unordered\_set<char> > sMap;

typedef map<pair<int, int>, unordered\_set<char> >::iterator iter\_M;

typedef unordered\_set<char>::iterator iter\_S;

int flag;

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

sMap map;

unordered\_set<char> set;

for (int i = '1'; i <= '9'; i++) set.insert(i);

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

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

if (board[i][j] == '.') map[make\_pair(i, j)] = set;

}

// eleminate selection

vector<pair<int, int> > pair;

iter\_M iter;

bool reduced = true;

while (reduced) {

reduced = false;

pair.clear();

for (iter = map.begin(); iter != map.end(); iter++) {

int row = iter->first.first;

int col = iter->first.second;

//scan corresponding column

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

if (board[i][col] != '.') iter->second.erase(board[i][col]);

}

//scan corresponding row

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

if (board[row][i] != '.') iter->second.erase(board[row][i]);

}

row = (row / 3) \* 3;

col = (col / 3) \* 3;

//check corresponding box

for (int i = row; i < row + 3; i++) {

for (int j = col; j < col + 3; j++)

if (board[i][j] != '.') iter->second.erase(board[i][j]);

}

//determine if the cell has only one choice, fill the cells with correct number

if (iter->second.size() == 1) {

reduced = true;

pair.push\_back(iter->first);

board[iter->first.first][iter->first.second] = \*iter->second.begin();

}

}

//delete map element with correct number

for (size\_t i = 0; i < pair.size(); i++) map.erase(pair[i]);

}

flag = false;

dfs(board, map.begin(), map.size());

}

void dfs(vector<vector<char> > &board, iter\_M iter1, int n) {

if (n == 0) {

flag = true;

return;

}

int row = iter1->first.first, col = iter1->first.second;

for (iter\_S iters = iter1->second.begin(); iters != iter1->second.end(); iters++) {

if (isValid(board, row, col, \*iters)) {

board[row][col] = \*iters;

dfs(board, ++iter1, n - 1);

if (flag) return;

--iter1;

board[row][col] = '.';

}

}

}

bool isValid(vector<vector<char> > &board, int x, int y, char c) {

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

if (board[i][y] == c) return false;

}

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

if (board[x][i] == c) return false;

}

x = (x / 3) \* 3;

y = (y / 3) \* 3;

for (int i = x; i < x + 3; i++) {

for (int j = y; j < y + 3; j++)

if (board[i][j] == c) return false;

}

return true;

}

};

class Solution {

public:

typedef map<pair<int, int>, unordered\_set<char> > sMap;

typedef map<pair<int, int>, unordered\_set<char> >::iterator iter\_M;

typedef unordered\_set<char>::iterator iter\_S;

int flag;

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

sMap map;

unordered\_set<char> set;

for (int i = '1'; i <= '9'; i++) set.insert(i);

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

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

if (board[i][j] == '.') map[make\_pair(i, j)] = set;

}

flag = false;

dfs(board, map.begin(), map.size());

}

void dfs(vector<vector<char> > &board, iter\_M iter1, int n) {

if (n == 0) {

flag = true;

return;

}

int row = iter1->first.first, col = iter1->first.second;

for (iter\_S iters = iter1->second.begin(); iters != iter1->second.end(); iters++) {

if (isValid(board, row, col, \*iters)) {

board[row][col] = \*iters;

dfs(board, ++iter1, n - 1);

if (flag) return;

--iter1;

board[row][col] = '.';

}

}

}

bool isValid(vector<vector<char> > &board, int x, int y, char c) {

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

if (board[i][y] == c) return false;

}

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

if (board[x][i] == c) return false;

}

x = (x / 3) \* 3;

y = (y / 3) \* 3;

for (int i = x; i < x + 3; i++) {

for (int j = y; j < y + 3; j++)

if (board[i][j] == c) return false;

}

return true;

}

};

/\* Search Insert Position

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.

Here are few examples.

[1,3,5,6], 5 → 2

[1,3,5,6], 2 → 1

[1,3,5,6], 7 → 4

[1,3,5,6], 0 → 0

\*/

class Solution {

public:

int searchInsert(int A[], int n, int target) {

int count = 0;

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

if (A[i] < target) count++;

else break;

}

return count;

}

};

/\* Valid Sudoku

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

A partially filled sudoku which is valid.

Note:

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

\*/

class Solution {

public:

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

unordered\_map<char, int> map;

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

map.clear();

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

if (board[i][j] != '.' && ++map[board[i][j]] > 1) return false;

}

}

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

map.clear();

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

if (board[j][i] != '.' && ++map[board[j][i]] > 1) return false;

}

}

for (int i = 0; i < 9; i += 3) {

for (int j = 0; j < 9; j += 3) {

map.clear();

if (!checkBox(board, i, j)) return false;

}

}

return true;

}

bool checkBox(vector<vector<char> > &board, int row, int col) {

unordered\_map<char, int> map;

for (int i = row; i < row + 3; i++) {

for (int j = col; j < col + 3; j++) {

if (board[i][j] != '.' && ++map[board[i][j]] > 1) return false;

}

}

return true;

}

};

class Solution {

public:

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

unordered\_map<char, int> map;

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

map.clear();

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

if (board[i][j] != '.' && ++map[board[i][j]] > 1) return false;

}

}

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

map.clear();

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

if (board[j][i] != '.' && ++map[board[j][i]] > 1) return false;

}

}

for (int i = 0; i < 9; i += 3) {

for (int j = 0; j < 9; j += 3) {

if (!checkBox(board, i, j)) return false;

}

}

return true;

}

bool checkBox(vector<vector<char> > &board, int row, int col) {

unordered\_map<char, int> map;

for (int i = row; i < row + 3; i++) {

for (int j = col; j < col + 3; j++) {

if (board[i][j] != '.' && ++map[board[i][j]] > 1) return false;

}

}

return true;

}

};

/\* Search in Rotated Sorted Array

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.

\*/

class Solution {

public:

int search(int A[], int n, int target) {

if (A == NULL || n == 0) return -1;

int start = 0, end = n - 1;

while (start + 1 < end) {

int mid = start + (end - start) / 2;

if (A[mid] == target) return mid;

if (A[mid] > A[start]) {

if (target < A[mid] && target > A[start]) end = mid;

else start = mid;

} else {

if (target > A[mid] && target < A[end]) start = mid;

else end = mid;

}

}

if (A[start] == target) return start;

if (A[end] == target) return end;

return -1;

}

};

/\* Search for a Range

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].

For example,

Given [5, 7, 7, 8, 8, 10] and target value 8,

return [3, 4].

\*/

class Solution {

public:

vector<int> searchRange(int A[], int n, int target) {

vector<int> ret;

int start = 0, end = n - 1;

while (start + 1 < end) { // find the first target

int mid = start + (end - start) / 2;

if (A[mid] >= target) end = mid;

else start = mid;

}

if (A[start] == target) ret.push\_back(start);

else if (A[end] == target) ret.push\_back(end);

else {

ret.push\_back(-1);

ret.push\_back(-1);

return ret;

}

start = 0;

end = n - 1;

while (start + 1 < end) { // find the second target

int mid = start + (end - start) / 2;

if (A[mid] <= target) start = mid;

else end = mid;

}

if (A[end] == target) ret.push\_back(end);

else if (A[start] == target) ret.push\_back(start);

return ret;

}

};

/\* Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

\*/

class Solution {

public:

int longestValidParentheses(string s) {

int ret = 0;

int n = s.size();

vector<int> dp(n + 1);

dp[n] = 0; // dp[i] indicates the length of longest valid parentheses start at i.

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

if (s[i] == ')') dp[i] = 0;

else {

int j = i + dp[i + 1] + 1;

if (j < n && s[j] == ')') dp[i] = dp[i + 1] + 2 + dp[j + 1];

else dp[i] = 0;

}

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

}

return ret;

}

};

class Solution {

public:

int longestValidParentheses(string s) {

vector<int>v;

int max = 0, start = 0;

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

if (s[i] == '(')

v.push\_back(i);

else{

if (v.empty()) // invalid

start = i + 1;

else{

v.pop\_back();

max = std::max(max, v.empty() ? i - start + 1 : i - v.back());

}

}

}

return max;

}

};;

/\* 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 the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

\*/

class Solution {

public:

void nextPermutation(vector<int> &num) {

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

int i, j;

for (i = num.size() - 2; i >= 0; i--) {

if (num[i] < num[i + 1]) break;

}

for (j = num.size() - 1; j > i; j--) {

if (num[j] > num[i]) break;

}

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

i = i + 1;

j = num.size() - 1;

while (i < j) {

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

i++;

j--;

}

}

};

/\* Substring with Concatenation of All Words

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

For example, given:

S: "barfoothefoobarman"

L: ["foo", "bar"]

You should return the indices: [0,9].

(order does not matter).

\*/

class Solution {

public:

vector<int> findSubstring(string S, vector<string> &L) {

vector<int> result;

if (S.size() == 0 || L.size() == 0 || S.size() < L.size() \* L[0].size()) return result;

int len = L[0].size();

unordered\_map<string, int> map1;

for (int i = 0; i < L.size(); i++) map1[L[i]]++;

for (int i = 0; i <= S.size() - L.size() \* len; i++) {

unordered\_map<string, int> map = map1;

int j = i;

while (!map.empty()) { // this is faster since the numbero elements in L is usually

string word = S.substr(j, len);

if (map.count(word)) {

if (--map[word] == 0) map.erase(word);

j += len;

} else break;

}

if (map.empty()) result.push\_back(i);

}

return result;

}

};

/\* Divide Two Integers @@@@@@@@@@@@@@@

Divide two integers without using multiplication, division and mod operator.

\*/

class Solution {

public:

int divide(int dividend, int divisor) {

unsigned long absDvd = (dividend > 0) ? dividend : -dividend;

unsigned long absDiv = (divisor > 0) ? divisor : -divisor;

if (absDvd == 0 || absDvd < absDiv) return 0;

char sign;

if ((dividend > 0 && divisor > 0) || (dividend < 0 && divisor < 0)) sign = '+';

else sign = '-';

unsigned long Div\_copy = absDiv;

int leftShift = 0;

while (absDvd > Div\_copy) {

Div\_copy = Div\_copy << 1; // double divisor until it is greater than dividend.

leftShift++; // how many times we doubled

}

int result = 0;

while (absDvd >= absDiv) {

if (Div\_copy <= absDvd) { //

absDvd -= Div\_copy;

result += 1 << leftShift;

}

Div\_copy = Div\_copy >> 1;

leftShift--;

}

if (sign == '+') return result;

else return -result;

}

};

/\* Implement strStr()

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Update (2014-11-02):

The signature of the function had been updated to return the index instead of the pointer. If you still see your function signature returns a char \* or String, please click the reload button to reset your code definition.

\*/

class Solution {

public:

char \*strStr(char \*haystack, char \*needle)

{

if(haystack == NULL || needle == NULL || strlen(haystack) < strlen(needle))

{

return NULL;

}

if(strlen(needle) == 0)

{

return haystack;

}

char \*p = haystack;

while(p <= (haystack + strlen(haystack) - strlen(needle)))

{

if(\*p == \*needle)

{

int i;

for(i = 0; i < strlen(needle); ++i)

{

if(\*(p + i) != \*(needle + i)) break;

}

if(i == strlen(needle)) return p;

}

++p;

}

return NULL;

}

};

/\* Remove Duplicates from Sorted Array

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.

For example,

Given input array A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

\*/

class Solution {

public:

int removeDuplicates(int A[], int n) {

if(A == NULL || n == 0) return 0;

int size = 0;

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

if(A[i] != A[size]) {

A[++size] = A[i];

}

}

return size + 1;

}

};

/\* Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":

What if duplicates are allowed at most twice?

For example,

Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].

\*/

class Solution {

public:

int removeDuplicates(int A[], int n) {

int size = 0;

bool twice = false;

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

if (A[i] != A[size]) {

A[++size] = A[i];

twice = false;

}

else if (!twice) {

A[++size] = A[i];

twice = true;

}

}

return n == 0 ? 0 : size + 1;

}

};

/\* Remove Element

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.

\*/

class Solution {

public:

int removeElement(int A[], int n, int elem) {

int num = 0, i;

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

if(A[i] != elem) A[num++] = A[i];

return num;

}

};

/\* Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,

Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*reverseKGroup(ListNode \*head, int k) {

ListNode \*Dummy = new ListNode(0);

Dummy->next = head;

ListNode \*prevTail = Dummy;

ListNode \*currHead = head;

ListNode \*node = head;

int count = 0;

while (node != NULL) {

count++;

if (count == k) {

ListNode \*temp = node->next;

node->next = NULL;

prevTail->next = reverse(currHead);

prevTail = currHead;

prevTail->next = temp;

currHead = temp;

node = prevTail;

count = 0;

}

node = node->next;

}

return Dummy->next;

}

ListNode \*reverse(ListNode \*head) {

ListNode \*curr = head;

ListNode \*prev = NULL;

while (curr != NULL) {

ListNode \*temp = curr->next;

curr->next = prev;

prev = curr;

curr = temp;

}

return prev;

}

};

/\* Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

For example,

Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*swapPairs(ListNode \*head) {

ListNode \*Dummy = new ListNode(0);

Dummy->next = head;

ListNode \*prevTail = Dummy;

ListNode \*currHead = head;

ListNode \*node = head;

while (node != NULL && node->next != NULL) {

node = node->next->next;

prevTail->next = reversepair(currHead);

prevTail = currHead;

currHead = node;

}

prevTail->next = node;

return Dummy->next;

}

ListNode \*reversepair(ListNode \*start) {

ListNode \*temp = start->next;

temp->next = start;

start->next = NULL;

return temp;

}

};

/\* Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

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

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

if (lists.size() == 2) return merge2Lists(lists[0], lists[1]);

int mid = lists.size() / 2;

vector<ListNode \*> left(lists.begin(), lists.begin() + mid);

vector<ListNode \*> right(lists.begin() + mid, lists.end());

return merge2Lists(mergeKLists(left), mergeKLists(right));

}

ListNode \*merge2Lists(ListNode \*l1, ListNode \*l2) {

if (l1 == NULL) return l2;

if (l2 == NULL) return l1;

ListNode \*Dummy = new ListNode(0);

ListNode \*prev = Dummy;

while (l1 != NULL && l2 != NULL) {

if (l1->val < l2->val) {

prev->next = l1;

prev = l1;

l1 = l1->next;

} else {

prev->next = l2;

prev = l2;

l2 = l2->next;

}

}

if (l1 == NULL) prev->next = l2;

else prev->next = l1;

return Dummy->next;

}

};

class Solution {

private:

struct cmp {

bool operator ()(const ListNode \*a, const ListNode \*b)

{

return a->val > b->val;

}

};

public:

ListNode \*mergeKLists(vector<ListNode \*> &lists) {

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

return NULL;

}

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

return lists[0];

}

ListNode \*Dummy = new ListNode(0);

ListNode \*node = Dummy;

priority\_queue<ListNode\*, vector<ListNode\*>, cmp> node\_queue;

for (size\_t i = 0; i < lists.size(); ++i) {

if (lists[i] != NULL) {

node\_queue.push(lists[i]);

}

}

while (!node\_queue.empty()) {

ListNode \*temp = node\_queue.top();

node->next = temp;

node\_queue.pop();

if (temp->next != NULL) {

node\_queue.push(temp->next);

}

node = node->next;

}

node->next = NULL;

return Dummy->next;

}

};

/\* Generate Parentheses @@@@@@@@@@@@@@@@@@@@@@@@@

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

"((()))", "(()())", "(())()", "()(())", "()()()"

\*/

class Solution {

public:

vector<string> generateParenthesis(int n) {

vector<string> res;

if(n == 0) return res;

string s;

getParenthesis(n, n, s, res);

return res;

}

void getParenthesis(int rr, int lr, string s, vector<string> &res) {

if(rr == 0 && lr == 0) {

res.push\_back(s);

return;

}

if(lr > 0) {

getParenthesis(rr, lr - 1, s+'(', res);

}

if(rr > lr) {

getParenthesis(rr - 1, lr, s+')', res);

}

}

};

/\* Valid Parentheses

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

\*/

class Solution {

public:

bool isValid(string s) {

stack<char> stack;

map<char, char> map;

map[')'] = '(';

map[']'] = '[';

map['}'] = '{';

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

if (s[i] == '(' || s[i] == '[' || s[i] == '{') stack.push(s[i]);

else {

if (stack.empty()) return false;

else if (stack.top() != map[s[i]]) return false;

else stack.pop();

}

}

if (!stack.empty()) return false;

return true;

}

};

/\* Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.

For example,

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.

Note:

Given n will always be valid.

Try to do this in one pass.

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*removeNthFromEnd(ListNode \*head, int n) {

ListNode \*front = head;

ListNode \*behind = head;

ListNode \*Dummy = new ListNode(0);

Dummy->next = head;

ListNode \*prev = Dummy;

while (n != 0) {

front = front->next;

--n;

}

while (front != NULL) {

prev = behind;

front = front->next;

behind = behind->next;

}

prev->next = behind->next;

delete behind;

return Dummy->next;

}

};

/\* 4Sum @@@@@@@@@@@@@@@@

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.

Note:

Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d)

The solution set must not contain duplicate quadruplets.

For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:

(-1, 0, 0, 1)

(-2, -1, 1, 2)

(-2, 0, 0, 2)

\*/

class Solution {

public:

vector<vector<int> > fourSum(vector<int> &num, int target) {

vector<vector<int> > result;

if (num.size() < 4) {

return result;

}

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

for (int i = 0; i < num.size() - 3; i++) {

if (i != 0 && num[i] == num[i - 1]) {

continue;

}

int a = num[i];

for (int j = i + 1; j < num.size() - 2; j++) {

if (j != i + 1 && num[j] == num[j - 1]) {

continue;

}

int b = num[j];

int left = j + 1, right = num.size() - 1;

while (left < right) {

int sum = num[left] + num[right] + a + b;

if (sum == target) {

vector<int> sol;

sol.push\_back(a);

sol.push\_back(b);

sol.push\_back(num[left]);

sol.push\_back(num[right]);

result.push\_back(sol);

left++;

right--;

// these two while loop is inside of if statement

while (left < right && num[left] == num[left - 1]) {

left++;

}

while (left < right && num[right] == num[right + 1]) {

right--;

}

} else if (sum < target) {

left++;

} else {

right--;

}

}

}

}

return result;

}

};

/\* Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

Input:Digit string "23"

Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

\*/

class Solution {

public:

vector<string> letterCombinations(string digits) {

unordered\_map<char, string> hmap;

hmap['0'] = "";

hmap['1'] = "";

hmap['2'] = "abc";

hmap['3'] = "def";

hmap['4'] = "ghi";

hmap['5'] = "jkl";

hmap['6'] = "mno";

hmap['7'] = "pqrs";

hmap['8'] = "tuv";

hmap['9'] = "wxyz";

vector<string> ret;

string sol;

dfs(ret, sol, digits, hmap, 0);

return ret;

}

void dfs(vector<string> &ret, string &sol, string &source, unordered\_map<char, string> &hmap, int pos) {

if (sol.size() == source.size()) {

ret.push\_back(sol);

return;

}

for (int i = 0; i < hmap[source[pos]].size(); i++) {

sol.push\_back(hmap[source[pos]][i]);

dfs(ret, sol, source, hmap, pos + 1);

sol.pop\_back();

}

}

};

/\* 3Sum Closest

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.

For 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).

\*/

class Solution {

public:

int threeSumClosest(vector<int> &num, int target) {

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

int cSum;

int Min = INT\_MAX;

for (int i = 0; i < num.size() - 2; i++) {

if ( i != 0 && num[i] == num[i - 1]) {

continue;

}

int a = num[i];

int left = i + 1, right = num.size() - 1;

while (left < right) {

int sum = a + num[left] + num[right];

if (abs(sum - target) < Min) {

Min = abs(sum - target);

cSum = sum;

}

if (sum < target) {

left++;

while (left < right && num[left] == num[left - 1]) {

left++;

}

}

else if (sum > target) {

right--;

while (left < right && num[right] == num[right + 1]) {

right--;

}

} else {

return sum;

}

}

}

return cSum;

}

};

/\* 3Sum

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.

Note:

Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)

The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:

(-1, 0, 1)

(-1, -1, 2)

\*/

class Solution {

public:

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

vector<vector<int> > result;

if (num.size() < 3) {

return result;

}

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

for (int i = 0; i < num.size() - 2; i++) {

if (i != 0 && num[i] == num[i - 1]) {

continue;

}

int a = num[i];

int l = i + 1, r = num.size() - 1;

while (l < r) {

if (num[l] + num[r] < -a) {

l++;

} else if (num[l] + num[r] > -a) {

r--;

} else {

vector<int> sol;

sol.push\_back(a);

sol.push\_back(num[l]);

sol.push\_back(num[r]);

result.push\_back(sol);

l++;

r--;

while (l < r && num[l] == num[l - 1]) {

l++;

}

while (l < r && num[r] == num[r + 1]) {

r--;

}

}

}

}

return result;

}

};

/\* Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

\*/

class Solution {

public:

string longestCommonPrefix(vector<string> &strs) {

string lcp;

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

return lcp;

}

return lcp\_helper(strs, 0, strs.size() - 1);

}

string lcp\_helper(vector<string> &strs, int start, int end) {

if (start == end) {

return strs[start];

}

else if (start == (end - 1)) {

return lcpOftwo(strs[start], strs[end]);

}

else {

int mid = start + (end - start) / 2;

string left = lcp\_helper(strs, start, mid);

string right = lcp\_helper(strs, mid + 1, end);

return lcpOftwo(left, right);

}

}

string lcpOftwo(string &s1, string &s2) {

if (s1.size() == 0 || s2.size() == 0) {

return "";

}

for (int i = 0; i < min(s1.size(), s2.size()); ++i) {

if (s1[i] != s2[i]) {

return s1.substr(0, i);

}

}

return s1.substr(0, min(s1.size(), s2.size()));

}

};

/\* Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

\*/

class Solution {

public:

int romanToInt(string s) {

unordered\_map<char, int> rtoi = { { 'I', 1 }, { 'V', 5 }, { 'X', 10 }, { 'L', 50 }, { 'C', 100 }, { 'D', 500 }, { 'M', 1000 } };;

int i = s.size() - 1;

int ret = 0;

int prev = 0;

while (i >= 0) {

int curr = rtoi[s[i]];

if(curr < prev) ret -= curr;

else ret += curr;

prev = curr;

i--;

}

return ret;

}

};

/\* Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

\*/

class Solution {

public:

string intToRoman(int num) {

string M[] = {"", "M", "MM", "MMM"};

string C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

string X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};

string I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];

}

};

/\* Container With Most Water

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.

Note: You may not slant the container.

\*/

class Solution { // it is not what I thougt like largest rectangle problem

public:

int maxArea(vector<int> &height) {

if (height.size() < 2) return 0;

int i = 0, j = height.size() - 1;

int maxarea = 0;

while(i < j) {

int area = 0;

if(height[i] < height[j]) {

area = height[i] \* (j-i);

//Since i is lower than j,

//so there will be no jj < j that make the area from i,jj

//is greater than area from i,j

//so the maximum area that can benefit from i is already recorded.

//thus, we move i forward.

//因为i是短板，所以如果无论j往前移动到什么位置，都不可能产生比area更大的面积

//换句话所，i能形成的最大面积已经找到了，所以可以将i向前移。

++i;

} else {

area = height[j] \* (j-i);

//the same reason as above

//同理

--j;

}

if(maxarea < area) maxarea = area;

}

return maxarea;

}

};

/\* Regular Expression Matching @@@@@@@@@@@@@@@@@

Implement regular expression matching with support for '.' and '\*'.

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "a\*") → true

isMatch("aa", ".\*") → true

isMatch("ab", ".\*") → true

isMatch("aab", "c\*a\*b") → true

\*/

class Solution {

public:

bool matchFirst(const char \*s, const char \*p) {

return (\*p == \*s || (\*p == '.' && \*s != '\0'));

}

bool isMatch(const char \*s, const char \*p) {

if (\*p == '\0') return \*s == '\0'; //empty

if (\*(p + 1) != '\*') {//without \*

if(!matchFirst(s,p)) return false;

return isMatch(s + 1, p + 1);

} else { //next: with a \*

if(isMatch(s, p + 2)) return true; //try the length of 0

while ( matchFirst(s,p) ) //try all possible lengths

if (isMatch(++s, p + 2))return true;

}

}

};

/\* Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

click to show spoilers.

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

\*/

class Solution {

public:

bool isPalindrome(int x) {

if (x < 0) return false;

int l = 0;

int divisor = 10;

int x1 = x;

while (x1 / divisor > 0) { // find the length of x

l++;

x1 /= 10;

}

int largeDivisor = pow(10, l), smallDivisor = 10;

while (largeDivisor >= smallDivisor) {

if (x / largeDivisor != ((x % smallDivisor) / (smallDivisor / 10))) return false;

else {

x %= largeDivisor;

largeDivisor /= 10;

smallDivisor \*= 10;

}

}

return true;

}

};

/\* String to Integer (atoi)

Implement atoi to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

spoilers alert... click to show requirements for atoi.

Requirements for atoi:

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT\_MAX (2147483647) or INT\_MIN (-2147483648) is returned.

\*/

class Solution {

public:

int atoi(const char \*str) {

int len = strlen(str);

int i = 0;

while (i < len && str[i] == ' ') i++;

if (i == len) return 0;

int sign = 1;

if (str[i] == '+') i++;

else if (str[i] == '-') {

sign = -1;

i++;

}

int sum = 0;

while (i < len && isdigit(str[i])) {

int digit = str[i] - '0';

if (sum > INT\_MAX / 10 || (sum == INT\_MAX / 10 && digit >= 8)) {

return sign == 1 ? INT\_MAX : INT\_MIN;

}

sum = sum \* 10 + digit;

i++;

}

return sign \* sum;

}

};

/\* Reverse Integer

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

click to show spoilers.

Have you thought about this?

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

Throw an exception? Good, but what if throwing an exception is not an option? You would then have to re-design the function (ie, add an extra parameter).

\*/

class Solution {

public:

int reverse(int x) {

bool neg = false;

if (x < 0) {

neg = true;

x = - x;

}

long long ret = 0;

while (x / 10) {

ret = ret \* 10 + x % 10; // very neat!!!

x /= 10;

}

ret = ret \* 10 + x;

if (ret > INT\_MAX) return 0;

return neg ? -(int)ret : (int)ret;

}

};

class Solution {

public:

int reverse(int x) {

int ret = 0;

while (x != 0) {

if (abs(ret) > 214748364) return 0; // why this number?

ret = ret \* 10 + x % 10;

x /= 10;

}

return ret;

}

};

/\* ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

\*/

class Solution {

public:

string convert(string s, int nRows) {

if (nRows == 1) return s;

vector<string> strs(nRows);

int row = 0;

int change;

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

if (row == 0) change = 1;

if (row == nRows - 1) change = -1;

strs[row] += s[i];

row += change;

}

string ret;

for (int i = 0; i < nRows; i++) ret += strs[i];

return ret;

}

};

/\* Longest Palindromic Substring

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

\*/

class Solution {

public:

string longestPalindrome(string s) {

int n = s.size();

if (n < 2) return s;

int len = 0;

int start = 0;

bool dp[1000][1000]; // must use stack array to avoid time limit exceed.

memset(dp, false, sizeof(dp)); // initialization.

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

dp[i][i] = true;

if (i < n - 1 && s[i] == s[i + 1]) {

len = 2;

start = i;

dp[i][i + 1] = true;

}

}

for (int l = 3; l <= n; l++) {

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

int j = i + l - 1;

if (s[i] == s[j] && dp[i + 1][j - 1]) {

dp[i][j] = true;

if (l > len) {

start = i;

len = l;

}

}

}

}

return s.substr(start, len);

}

};

/\* Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 0 -> 8

\*/

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*addTwoNumbers(ListNode \*l1, ListNode \*l2) {

if (l1 == NULL) return l2;

if (l2 == NULL) return l1;

int carry = 0;

ListNode \*Dummy = new ListNode(0);

ListNode \*prev = Dummy;

while (l1 != NULL && l2 != NULL) {

int sum = l1->val + l2->val + carry;

carry = sum / 10;

sum %= 10;

ListNode \*temp = new ListNode(sum);

prev->next = temp;

prev = temp;

l1 = l1->next;

l2 = l2->next;

}

while (l1 != NULL) {

int sum = l1->val + carry;

carry = sum / 10;

sum %= 10;

ListNode \*temp = new ListNode(sum);

prev->next = temp;

prev = temp;

l1 = l1->next;

}

while (l2 != NULL) {

int sum = l2->val + carry;

carry = sum / 10;

sum %= 10;

ListNode \*temp = new ListNode(sum);

prev->next = temp;

prev = temp;

l2 = l2->next;

}

if (carry != 0) {

ListNode \*temp = new ListNode(carry);

prev->next = temp;

prev = temp;

}

prev->next = NULL;

return Dummy->next;

}

};

/\* Longest Substring Without Repeating Characters

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.

\*/

class Solution {

public:

int lengthOfLongestSubstring(string s) {

unordered\_map<char,int> ht;

int n=s.size(), longest=0, prev=0;

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

if(ht.count(s[i])>0) { // find duplicate

longest = max(longest,i-prev);

while(prev<=ht[s[i]])

ht.erase(s[prev++]);

}

ht[s[i]]=i;

}

return max(longest,n-prev);

}

};

/\* Median of Two Sorted Arrays

There are two sorted arrays A and B 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)).

\*/

class Solution {

public:

double findMedianSortedArrays(int A[], int m, int B[], int n) {

int len = m + n;

if (len % 2 == 0) return (findKth(A, 0, m, B, 0, n, len / 2) + findKth(A, 0, m, B, 0, n, len / 2 + 1)) / 2.0;

else return findKth(A, 0, m, B, 0, n, len / 2 + 1);

}

int findKth(int A[], int A\_start, int m, int B[], int B\_start, int n, int k) {

if (A\_start >= m) return B[B\_start + k - 1];

if (B\_start >= n) return A[A\_start + k - 1];

if (k == 1) return A[A\_start] < B[B\_start] ? A[A\_start] : B[B\_start];

int a\_key = (A\_start + k / 2 - 1 < m) ? A[A\_start + k / 2 - 1] : INT\_MAX;

int b\_key = (B\_start + k / 2 - 1 < m) ? B[B\_start + k / 2 - 1] : INT\_MAX;

if (a\_key < b\_key) findKth(A, A\_start + k / 2, m, B, B\_start, n, k - k / 2);

else findKth(A, A\_start, m, B, B\_start + k / 2, n, k - k / 2);

}

};

/\* Two Sum

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.

Input: numbers={2, 7, 11, 15}, target=9

Output: index1=1, index2=2

\*/

class Solution {

public:

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

vector<int> ret;

if (numbers.size() < 2) return ret;

unordered\_map<int, int> hmap;

hmap[target - numbers[0]] = 1;

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

if (hmap.count(numbers[i])) {

ret.push\_back(hmap[numbers[i]]);

ret.push\_back(i + 1);

return ret;

}

hmap[target - numbers[i]] = i + 1;

}

}

};

/\* Search in Rotated Sorted Array II

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.

\*/

class Solution {

public:

bool search(int A[], int n, int target) {

if (n == 0) return false;

int start = 0, end = n - 1;

while (start + 1 < end) {

int mid = start + (end - start) / 2;

if (A[mid] == target) return true;

if (A[mid] > A[start]) {

if (target < A[mid] && target >= A[start]) end = mid; // >= is critical!!

else start = mid;

}

else if (A[mid] < A[end]) {

if (target > A[mid] && target <= A[end]) start = mid; // <= is critical!!

else end = mid;

}

else {

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

if (A[i] == target) return true;

}

return false;

}

}

if (A[start] == target || A[end] == target) return true;

return false;

}

};