# 二叉树专题（labuladong）

## 144、94、145二叉树的前中后序遍历

递归写法好说

注意掌握迭代写法

基本思路是写一个栈，把根节点入栈，然后循环遍历取栈顶，注意几点：

1. 子节点入栈的顺序是先右后左
2. 根节点进入ans vector
3. 前中后顺序不同
4. 中序和后序遍历要加flag，只有回溯的时候才把根节点加入ans vector

有一种方式是压空包，我实现的另一种方式是搞一个std::pair，都行

## 102 二叉树的层序遍历

While(q)里面做一些细的操作，先取得目前队列的长度，然后for循环，这样可以一层一层地分开遍历

## 104 二叉树最大深度

自顶向下和自底向上，两种思路

// LMB : 自顶向下

class Solution {

public:

int maxDepth(TreeNode\* root) {

int max\_depth = 0;

int depth = 0;

depthHelper(root, depth, max\_depth);

return max\_depth;

}

void depthHelper(TreeNode\* root, int depth, int& max\_depth) {

if(!root) return;

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

max\_depth = max(max\_depth, depth+1);

}

depthHelper(root->left, depth+1, max\_depth);

depthHelper(root->right, depth+1, max\_depth);

}

};

 // LMB : 自底向上

class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if(!root) return 0;

        return 1 + max(maxDepth(root->left), maxDepth(root->right));

    }

};

或者

class Solution {

public:

int maxDepth(TreeNode\* root) {

return root ? 1 + max(maxDepth(root->left), maxDepth(root->right)) : 0;

}

};

## 101 对称二叉树

注意不能从根节点写一个函数来决定，比如下面这样是错的！

class Solution {

public:

    bool isSymmetric(TreeNode\* root) {

        if(!root) return true;

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

        if(!root->left || !root->right) return false;

        if(root->left->val != root->right->val) return false;

        return isSymmetric(root->left) && isSymmetric(root->right);

    }

};

必须写一个双节点递归函数才行：

class Solution {

public:

bool isSymmetric(TreeNode\* root) {

return root ? isSymmetric(root->left, root->right) : true;

}

bool isSymmetric(TreeNode\* left, TreeNode\* right) {

if( !left && !right ) return true;

if( !left || !right ) return false;

if(left->val != right->val) return false;

return isSymmetric(left->left, right->right) && isSymmetric(left->right, right->left);

}

};

## 112、路径总和

class Solution {

public:

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

        if(!root) return false;

        if(!root->left && !root->right) return root->val == sum;

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

    }

}

注意分析根节点为空 与 叶节点的情形能不能合并。本题不能合并，所以要分开写！

## 116. 填充每个节点的下一个右侧节点指针

以前，这道题我用层序遍历来生硬解决：

class Solution {

public:

Node\* connect(Node\* root) {

if(!root) return nullptr;

queue<Node\*> q;

q.push(root);

while(!q.empty()) {

int count = q.size();

Node\* leftest = q.front(), \*prev = leftest;

q.pop();

if(leftest->left) q.push(leftest->left);

if(leftest->right) q.push(leftest->right);

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

Node\* node = q.front();

q.pop();

prev->next = node;

prev = node;

if(node->left) q.push(node->left);

if(node->right) q.push(node->right);

}

prev->next = nullptr;

}

return root;

}

};

现在，我用递归来解决。注意，要启用双节点递归：

class Solution {

public:

    Node\* connect(Node\* root) {

        if(root) connect(root->left, root->right);

        return root;

    }

    void connect(Node\* left, Node\* right) {

        if(!left || !right) return;

        left->next = right;

        connect(left->left, left->right);

        connect(right->left, right->right);

        connect(left->right, right->left);

    }

};

## 226、翻转二叉树

class Solution {

public:

    TreeNode\* invertTree(TreeNode\* root) {

        helper(root);

        return root;

    }

    void helper(TreeNode\* root) {

        if(!root) return;

        swap(root->left, root->right);

        helper(root->left);

        helper(root->right);

    }

};

## 114、二叉树展开为链表

class Solution {

public:

    void flatten(TreeNode\* root) {

        if(!root) return;

        // if(!root->left && !root->right) return;

        flatten(root->left);

        flatten(root->right);

        TreeNode\* left\_raw = root->left, \*right\_raw = root->right;

        root->left = nullptr;

        root->right = left\_raw;

        TreeNode\* n=root;

        while(n->right != nullptr) {

            n = n->right;

        }

        n->right = right\_raw;

    }

};

思路：典型的后续遍历

# 翻转链表（反转链表）专题

## 206 反转链表

迭代写法：

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

ListNode \*prev = nullptr, \*next;

while(head) {

next = head->next;

head->next = prev;

prev = head;

head = next;

}

return prev;

}

};

后序遍历递归写法（回溯写法）

// 回溯写法

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if( !head || !head->next ) {

return head;

}

ListNode\* ret = reverseList(head->next);

head->next->next = head;

head->next = nullptr;

return ret;

}

};

先序遍历，双参数写法：

class Solution {

public:

// ListNode\* reverseList(ListNode\* head) {

// return reverseListRecursive(head);

// }

ListNode\* reverseList(ListNode\* head, ListNode\* prev = nullptr) {

if(!head) {

return prev;

}

ListNode\* next = head->next;

head->next = prev;

return reverseList(next, head);

}

};

## 92 反转链表 II

// 方法源自labuladong

/\*

https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247484467&idx=1&sn=beb3ae89993b812eeaa6bbdeda63c494&chksm=9bd7fa3baca0732dc3f9ae9202ecaf5c925b4048514eeca6ac81bc340930a82fc62bb67681fa&scene=21#wechat\_redirect

\*/

class Solution {

public:

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

        if( m == 1 ) {

            ListNode\* successor = nullptr;

            return reverseN(head, n, successor);

        }

        head->next = reverseBetween(head->next, m-1, n-1);

        return head;

    }

    ListNode\* reverseN(ListNode\* head, int n, ListNode\*& successor) {

        if(n==0) return head;

        if(n==1) {

            successor = head->next;

            return head;

        }

        ListNode\* ret = reverseN(head->next, n-1, successor);

        head->next->next = head;

        head->next = successor;

        return ret;

    }

};

关键：学会用一句或两句白话来说清楚函数的作用，并且坚决相信函数可以实现它。

值得一提的是，**递归操作链表并不高效。**和迭代解法相比，虽然时间复杂度都是 O(N)，但是迭代解法的空间复杂度是 O(1)，而递归解法需要堆栈，空间复杂度是 O(N)。所以递归操作链表可以作为对递归算法的练习或者拿去和小伙伴装逼，但是考虑效率的话还是使用迭代算法更好。

## 25 K个一组翻转链表

class Solution {

public:

    ListNode\* reverseN(ListNode\* a, ListNode\* b) {

        ListNode \*prev = nullptr, \*next;

        while(a!=b) {

            next = a->next;

            a->next = prev;

            prev = a;

            a = next;

        }

        return prev;

    }

    ListNode\* reverseKGroup(ListNode\* head, int k) {

        ListNode \*a = head, \*b = head;

        for(int i=0; i<k; ++i) {

            if( !b ) return head;

            b = b->next;

        }

        ListNode\* ret = reverseN(a, b);

        a->next = reverseKGroup(b, k);

        return ret;

    }

};

/\*

参考labuladong

https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247484597&idx=1&sn=c603f1752e33cb2701e371d84254aee2&chksm=9bd7fabdaca073abd512d8fff18016c9092ede45fed65c307852c65a2026d8568ee294563c78&scene=21#wechat\_redirect

思路 ：外层函数定义，K个一组翻转以head为头结点的链表，返回翻转后的头结点

一般情形：

1、翻转前k个节点

2、K个一组翻转以head往右k个结点为头结点的链表，返回翻转后的头结点（子问题）

3、两者接起来

base case：

不足k个结点，直接返回head

\*/

心得：没想清楚思路的时候，要做的就是：

甩开编程语言，用白话去分析问题

怎么用白话分析问题？清晰地用语言描述函数的功能。

怎么清晰地用语言描述函数的功能？输入什么，做什么事，返回什么

其实不只是编程，生活中的任何事都是两点：

1. 想清楚；2、干明白

想清楚是干明白的前提

事情进展不顺的时候，要像打篮球那样，叫个暂停

分析下究竟是没想清楚，还是没干明白

# 回文小专题

回文是什么？是正着读反着读都一样的字符串。解决回文问题的核心是双指针

英文名：**Palindrome**

## 25. 最长回文子串

思路：

**for** 0 <= i < len(s):  
    找到以 s[i] 为中心的回文串  
    找到以 s[i] 和 s[i+1] 为中心的回文串  
    更新答案

class Solution {

public:

    string longestPalindrome(string s) {

        string res;

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

            string s1 = palindrome(s, i, i);

            string s2 = palindrome(s, i, i+1);

            res = s1.length() > res.length() ? s1 : res;

            res = s2.length() > res.length() ? s2 : res;

        }

        return res;

    }

    string palindrome(string s, int l, int r) {

        while(l>=0 && r<s.length() && s[l]==s[r]) {

            --l;

            ++r;

        }

        return s.substr(l+1, (r-1)-(l+1)+1);

    }

};

/\*

来自labuladong

https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247484471&idx=1&sn=7c26d04a1f035770920d31377a1ebd42&chksm=9bd7fa3faca07329189e9e8b51e1a665166946b66b8e8978299ba96d5f2c0d3eafa7db08b681&scene=21#wechat\_redirect

思路：

int longestPalindrome(string s) {

    for i 0~len

        pnlindrome(s, i, i)

        palindrome(s, i, i+1)

        更新

}

\*/

## 回文链表

方法一：后序遍历

class Solution {public:

bool isPalindrome(ListNode\* head) {

ListNode\* left = head;

return traverse(left, head);

}

bool traverse(ListNode\*& left, ListNode\* right) {

if(!right) return true;

bool ret = traverse(left, right->next);

ret = ret && (left->val == right->val);

left = left->next;

return ret;

}

};

/\*

后续遍历

过程略有冗余，来自labuladong

https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247484822&idx=1&sn=44742c9a3557038c8da7150100d94db9&chksm=9bd7fb9eaca0728876e1146306a09f5453bcd5c35c4a264304ea6189faa83ec12a00322f0246&scene=21#wechat\_redirect

\*/

方法二：快慢指针找中点 + 反转

class Solution {

public:

    bool isPalindrome(ListNode\* head) {

        ListNode \*fast = head, \*slow = head;

        while(fast && fast->next) {

            fast = fast->next->next;

            slow = slow->next;

        }

        if(fast) slow = slow->next;

        ListNode \*left = head, \*right = reverse(slow);

        while(right) {

            if(left->val != right->val) return false;

            left = left->next;

            right = right->next;

        }

        return true;

    }

    ListNode\* reverse(ListNode\* head) {

        ListNode \*prev = nullptr, \*next;

        while(head) {

            next = head->next;

            head->next = prev;

            prev = head;

            head = next;

        }

        return prev;

    }

};

/\*

先用快慢指针找中点的方法

来自labuladong

https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247484822&idx=1&sn=44742c9a3557038c8da7150100d94db9&chksm=9bd7fb9eaca0728876e1146306a09f5453bcd5c35c4a264304ea6189faa83ec12a00322f0246&scene=21#wechat\_redirect

\*/

# 回到二叉树

## 二叉树的最近公共祖先

这道题代码看着简单，其实非常不好理解。但它却是理解二叉树遍历解法的一道非常好的题目。

class Solution {

public:

    TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

        if(!root) return nullptr;

        if(p==root || q==root) return root;

        TreeNode\* left = lowestCommonAncestor(root->left, p, q);

        TreeNode\* right = lowestCommonAncestor(root->right, p, q);

        if(left != nullptr && right != nullptr) {

            return root;

        }

        if(left == nullptr && right == nullptr) {

            return nullptr;

        }

        return !left ? right : left;

    }

};

/\*

来自labuladong

https://mp.weixin.qq.com/s?\_\_biz=MzAxODQxMDM0Mw==&mid=2247485561&idx=1&sn=a394ba978283819da1eb34a256f6915b&chksm=9bd7f671aca07f6722f0bc1e946ca771a0a40fd8173cc1227a7e0eabfe4e2fcc57b9ba464547&scene=21#wechat\_redirect

这是一道很经典的二叉树递归的题目，非常非常经典。

\*/

## 654 最大二叉树

比较简单：

class Solution {

public:

    TreeNode\* constructMaximumBinaryTree(vector<int>& nums) {

        return build(nums, 0, nums.size()-1);

    }

    TreeNode\* build(vector<int>& nums, int l, int r) {

        if(l>r) return nullptr;

        int max\_idx = -1;

        int max\_val = INT\_MIN;

        for(int i=l; i<=r; ++i) {

            if(nums[i] > max\_val) {

                max\_idx = i;

                max\_val = nums[i];

            }

        }

        TreeNode\* root = new TreeNode(max\_val);

        root->left = build(nums, l, max\_idx-1);

        root->right = build(nums, max\_idx+1, r);

        return root;

    }

};

## 从前序与中序遍历序列构造二叉树

class Solution {

public:

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

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

    }

    TreeNode\* build(vector<int>& preorder, int pre\_l, int pre\_r,

                    vector<int>& inorder, int in\_l, int in\_r) {

        if(pre\_l > pre\_r) return nullptr;

        int root\_val = preorder[pre\_l];

        int root\_idx\_inorder = -1;

        for(int i=in\_l; i<=in\_r; ++i) {

            if(inorder[i] == root\_val) {

                root\_idx\_inorder = i;

            }

        }

        int left\_num = root\_idx\_inorder - in\_l;

        // int right\_num = in\_r - root\_idx\_inorder;

        TreeNode\* root = new TreeNode(root\_val);

        root->left = build(preorder, pre\_l+1, pre\_l+left\_num, inorder, in\_l, root\_idx\_inorder-1);

        root->right = build(preorder, pre\_l+left\_num+1, pre\_r, inorder, root\_idx\_inorder+1, in\_r);

        return root;

    }

};

// 要完全理解才行

## 从中序与后序遍历序列构造二叉树

class Solution {

public:

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

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

}

TreeNode\* build(vector<int>& inorder, int in\_l, int in\_r,

vector<int>& postorder, int post\_l, int post\_r) {

if(post\_l > post\_r) return nullptr;

int root\_val = postorder[post\_r];

int root\_index;

TreeNode\* root = new TreeNode(root\_val);

for(int i=in\_l; i<=in\_r; ++i) {

if(inorder[i] == root\_val) {

root\_index = i;

break;

}

}

int right\_num = in\_r - root\_index;

root->right = build(inorder, root\_index+1, in\_r, postorder, post\_r-1-right\_num+1, post\_r-1);

root->left = build(inorder, in\_l, root\_index-1, postorder, post\_l, post\_r-1-right\_num);

return root;

}

};

这个注意子树是先右后左即可。其他的就是计算index的细节。

## 297 二叉树的序列化与反序列化

前序遍历、后序遍历、层序遍历三种解法，看提交记录即可

// 前序遍历法

class Codec {

public:

// Encodes a tree to a single string.

string serialize(TreeNode\* root) {

string ret = "[";

ret += serializeHelper(root);

ret.back() = ']';

// cout << ret << endl;

return ret;

}

string serializeHelper(TreeNode\* root) {

if(!root) return "null,";

string ret = to\_string(root->val) + ",";

ret += serializeHelper(root->left);

ret += serializeHelper(root->right);

return ret;

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

queue<int> nodes;

decode(data, nodes);

return deserializeHelper(nodes);

}

TreeNode\* deserializeHelper(queue<int>& nodes) {

if(nodes.empty()) return nullptr;

int root\_val = nodes.front();

nodes.pop();

if(root\_val == INT\_MIN) return nullptr;

TreeNode\* root = new TreeNode(root\_val);

root->left = deserializeHelper(nodes);

root->right = deserializeHelper(nodes);

return root;

}

void decode(const string& code, queue<int>& nodes) {

string \_code = code.substr(1, code.length()-2);

// cout << \_code << endl;

istringstream iss(\_code);

string line;

while(getline(iss, line, ',')) {

int val = line == "null" ? INT\_MIN : stod(line);

nodes.push(val);

}

}

};

// 后序遍历法

class Codec {

public:

// Encodes a tree to a single string.

string serialize(TreeNode\* root) {

string ret = "[";

ret += serializeHelper(root);

ret.back() = ']';

return ret;

}

string serializeHelper(TreeNode\* root) {

if(!root) return "null,";

string ret;

ret += serializeHelper(root->left);

ret += serializeHelper(root->right);

ret += to\_string(root->val) + ",";

return ret;

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

stack<int> nodes;

decode(data, nodes);

return deserializeHelper(nodes);

}

TreeNode\* deserializeHelper(stack<int>& nodes) {

if(nodes.empty()) return nullptr;

int root\_val = nodes.top();

nodes.pop();

if(root\_val == INT\_MIN) return nullptr;

TreeNode\* root = new TreeNode(root\_val);

root->right = deserializeHelper(nodes);

root->left = deserializeHelper(nodes);

return root;

}

void decode(const string& code, stack<int>& nodes) {

string \_code = code.substr(1, code.length()-2);

istringstream iss(\_code);

string line;

while(getline(iss, line, ',')) {

int val = line == "null" ? INT\_MIN : stod(line);

nodes.push(val);

}

}

};

// 层序遍历法

// 方法三 ： 层序遍历

class Codec {

public:

// Encodes a tree to a single string.

string serialize(TreeNode\* root) {

string ret = "[";

ret += serializeHelper(root);

ret.back() = ']';

return ret;

}

string serializeHelper(TreeNode\* root) {

if(!root) return "null,";

string ret;

queue<TreeNode\*> q;

q.push(root);

while(!q.empty()) {

TreeNode\* node = q.front();

q.pop();

if(!node) {

ret += "null,";

continue;

}

ret += to\_string(node->val) + ",";

q.push(node->left);

q.push(node->right);

}

cout << ret << endl;

return ret;

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

queue<int> nodes;

decode(data, nodes);

return deserializeHelper(nodes);

}

TreeNode\* deserializeHelper(queue<int>& nodes) {

if(nodes.empty()) return nullptr;

if(nodes.front() == INT\_MIN) return nullptr;

TreeNode\* root = new TreeNode(nodes.front());

nodes.pop();

queue<TreeNode\*> q;

q.push(root);

while(!nodes.empty()) {

TreeNode\* farther = q.front();

q.pop();

if(!farther) continue;

int left\_val = nodes.front();

nodes.pop();

if(left\_val != INT\_MIN) {

TreeNode\* left\_child = new TreeNode(left\_val);

farther->left = left\_child;

q.push(left\_child);

} else {

farther->left = nullptr;

}

int right\_val = nodes.front();

nodes.pop();

if(right\_val != INT\_MIN) {

TreeNode\* right\_child = new TreeNode(right\_val);

farther->right = right\_child;

q.push(right\_child);

} else {

farther->right = nullptr;

}

}

return root;

}

void decode(const string& code, queue<int>& nodes) {

string \_code = code.substr(1, code.length()-2);

istringstream iss(\_code);

string line;

while(getline(iss, line, ',')) {

int val = line == "null" ? INT\_MIN : stod(line);

nodes.push(val);

}

}

};

## 652. 寻找重复的子树

难度2

class Solution {

public:

    vector<TreeNode\*> findDuplicateSubtrees(TreeNode\* root) {

        vector<TreeNode\*> result;

        map<string, int> subtree\_counter;

        traverse(root, subtree\_counter, result);

        return result;

    }

    // 序列化函数

    // traverse函数

    string traverse(TreeNode\* root, map<string, int>& subtree\_counter, vector<TreeNode\*>& result) {

        if(!root) return "";

        string left = traverse(root->left, subtree\_counter, result);

        string right = traverse(root->right, subtree\_counter, result);

        string subtree = left + "," + right + "," + to\_string(root->val);

        ++subtree\_counter[subtree];

        if(subtree\_counter[subtree] == 2) {

            result.push\_back(root);

        }

        return subtree;

    }

};

## 230. 二叉搜索树中第k小的元素

难度1

class Solution {

public:

    int kthSmallest(TreeNode\* root, int k) {

        int rank = 0, result = 0;

        traverse(root, k, rank, result);

        return result;

    }

    void traverse(TreeNode\* root, int k, int& rank, int& result) {

        if(!root) return;

        traverse(root->left, k, rank, result);

        ++rank;

        if(rank == k) {

            result = root->val;

            return;

        }

        traverse(root->right, k, rank, result);

    }

};

## 538. 把二叉搜索树转换为累加树

难度2

class Solution {

public:

    TreeNode\* convertBST(TreeNode\* root) {

        int sum = 0;

        traverseBST(root, sum);

        return root;

    }

    void traverseBST(TreeNode\* root, int& sum) {

        if(!root) return;

        traverseBST(root->right, sum);

        root->val += sum;

        sum = root->val;

        traverseBST(root->left, sum);

    }

};

## 98. 验证二叉搜索树

难度3

class Solution {

public:

    bool isValidBST(TreeNode\* root) {

        return isValidBST(root, nullptr, nullptr);

    }

    bool isValidBST(TreeNode\* root, TreeNode\* min, TreeNode\* max) {

        if(!root) return true;

        if(min != nullptr && root->val <= min->val) return false;

        if(max != nullptr && root->val >= max->val) return false;

        return isValidBST(root->left, min, root) &&

                isValidBST(root->right, root, max);

    }

};

# 动态规划

## 322. 零钱兑换

class Solution {

public:

    int coinChange(vector<int>& coins, int amount) {

        vector<int> dp(amount+1, amount+1);

        dp[0] = 0;

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

            for(auto coin : coins) {

                if(i-coin < 0) continue;

                dp[i] = min(dp[i], dp[i-coin]+1);

            }

        }

        return (dp[amount] == amount+1) ? -1 : dp[amount];

    }

};

## 72. 编辑距离

class Solution {

public:

    int minDistance(string word1, string word2) {

        int m = word1.length(), n = word2.length();

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

        for(int i=0; i<=m; ++i) dp[i][0] = i;

        for(int j=0; j<=n; ++j) dp[0][j] = j;

        for(int i=1; i<=m; ++i) {

            for(int j=1; j<=n; ++j) {

                if(word1[i-1] == word2[j-1]) {   // 这里一定要注意！！

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

                }

                else {

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

                }

            }

        }

        return dp[m][n];

    }

};

/\*

状态，也就是决定问题规模的量是：i,j，即游标

选择，插入、删除、替换，或者啥也不做

定义，word1[0…i]，word2[0…j]的最小编辑距离：dp[i][j]

转移方程:

if(w1[i]==w2[j]) dp[i][j] = dp[i-1][j-1]

else dp[i][j] = min(选择) + 1

Base case ：dp[0][j] = j    dp[i][0] = i

返回值 dp[m][n]

\*/

## 1143. 最长公共子序列

class Solution {

public:

    int longestCommonSubsequence(string text1, string text2) {

        int m = text1.length(), n = text2.length();

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

        for(int i=1; i<=m; ++i) {

            for(int j=1; j<=n; ++j) {

                if(text1[i-1] == text2[j-1]) {

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

                }

                else {

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

                }

            }

        }

        return dp[m][n];

    }

};

## 516. 最长回文子序列

class Solution {

public:

    int longestPalindromeSubseq(string s) {

        int n = s.length();

        vector<vector<int>> dp(n, vector<int>(n, 0));

        for(int i=0; i<n; ++i) dp[i][i] = 1;

        for(int i=n-1; i>=0; --i) {

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

                if(s[i] == s[j]) dp[i][j] = dp[i+1][j-1] + 2;

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

            }

        }

        return dp[0][n-1];

    }

};

/\*

采用双游标i，j

最优子结构，可以动态规划

状态：i，j

选择：s[i-1]是否等于s[j+1]

定义：dp[i][j]  i-j的长度

方程：等于，则加2；不等，则取个最大

接下来是难点：

Base case：i==j

遍历方向：反着或斜着遍历。见下页

\*/

拓展 ：

1. 能否压缩空间
2. 斜向遍历怎么写
3. 路径如何记录

class Solution {

public:

    int findTargetSumWays(vector<int>& nums, int S) {

        int sum = 0, method\_num = 0;

        backtrack(nums, 0, S, sum, method\_num);

        return method\_num;

    }

    void backtrack(vector<int>& nums, int idx, int S, int& sum, int& method\_num) {

        if(idx == nums.size()) {

            if(sum == S) {

                ++method\_num;

            }

            return;

        }

        sum += nums[idx];

        backtrack(nums, idx+1, S, sum, method\_num);

        sum -= nums[idx];

        sum -= nums[idx];

        backtrack(nums, idx+1, S, sum, method\_num);

        sum += nums[idx];

    }

};

## 300. 最长递增子序列

标准解法：

class Solution {

public:

    int lengthOfLIS(vector<int>& nums) {

        int n = nums.size();

        int max\_length = 0;

        vector<int> dp(n,1);

        for(int i=0; i<n; ++i) {

            for(int j=0; j<i; ++j) {

                if(nums[i] > nums[j])

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

            }

            max\_length = max(max\_length, dp[i]);

        }

        return max\_length;

    }

};

class Solution {

public:

    int lengthOfLIS(vector<int>& nums) {

        int n = nums.size();

        if(n<=1) return n;

        vector<int> obj;

        for(int i=0; i<n; ++i) {

            if(obj.empty() || nums[i] > obj.back())

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

            else {

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

                \*iter = nums[i];

            }

        }

        return obj.size();

    }

};

// 标准解法 - 二分查找 + 贪心法

class Solution {

public:

    int lengthOfLIS(vector<int>& nums) {

        int n = nums.size();

        if(n<=1) return n;

        vector<int> obj;

        for(int i=0; i<n; ++i) {

            if(obj.empty() || nums[i] > obj.back())

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

            else {

                int l=0, r=obj.size();

                while(l<r) {

                    int mid = l + (r-l)/2;

                    if(nums[i] > obj[mid])

                        l = mid+1;

                    else if (nums[i] < obj[mid])

                        r = mid;

                    else if (nums[i] == obj[mid])

                        r = mid;

                }

                obj[l] = nums[i];

            }

        }

        return obj.size();

    }

};

// 标准解法 – 自研二分查找 + 贪心法

class Solution {

public:

    int lengthOfLIS(vector<int>& nums) {

        int n = nums.size();

        if(n<=1) return n;

        int piles = 0;

        vector<int> top(n);   // 维护每个堆顶

        for(int i=0; i<n; ++i) {

            int poker = nums[i];

            int l=0, r=piles;

            while(l<r) {

                int mid = l + (r-l)/2;

                if(top[mid] > nums[i])

                    r = mid;

                if(top[mid] < nums[i])

                    l = mid + 1;

                if(top[mid] == nums[i])

                    r = mid;

            }

            if(l == piles) ++piles;

            top[l] = poker;

        }

        return piles;

    }

};

// 标准解法 - 二分查找 + 扑克堆

## 673. 最长递增子序列的个数

class Solution {

public:

    int findNumberOfLIS(vector<int>& nums) {

        int n = nums.size();

        if(n<=1) return n;

        vector<int> dp\_length(n,1);

        vector<int> dp\_count(n,1);

        int max\_length = 0;

        for(int i=0; i<n; ++i) {

            for(int j=0; j<i; ++j) {

                if(nums[i] > nums[j]) {

                    if(dp\_length[j] >= dp\_length[i]) {

                        dp\_length[i] = dp\_length[j] + 1;

                        dp\_count[i] = dp\_count[j];

                    }

                    else if (dp\_length[j]+1 == dp\_length[i]) {

                        dp\_count[i] += dp\_count[j];

                    }

                }

            }

            if(dp\_length[i] > max\_length) {

                max\_length = dp\_length[i];

            }

        }

        int ret = 0;

        for(int i=0; i<n; ++i) {

            if(dp\_length[i] == max\_length) {

                ret += dp\_count[i];

            }

        }

        return ret;

    }

};

## 494. 目标和

class Solution {

public:

    int findTargetSumWays(vector<int>& nums, int S) {

        int sum = accumulate(nums.begin(), nums.end(), 0);

        if(sum < S || (sum + S) % 2 == 1) return 0;

        return subset(nums, (sum + S)/2);

    }

    int subset(vector<int>& nums, int sum) {

        int n = nums.size();

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

        for(int i=0; i<=n; ++i)

            dp[i][0] = 1;

        for(int i=1; i<=n; ++i) {

            for(int j=0; j<=sum; ++j) {   // 细节，j从0开始

                if(j<nums[i-1]) {

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

                }

                else {

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

                }

            }

        }

        return dp[n][sum];

    }

};

压缩空间版本：

class Solution {

public:

    int findTargetSumWays(vector<int>& nums, int S) {

        int sum = accumulate(nums.begin(), nums.end(), 0);

        if(sum < S || (sum + S) % 2 == 1) return 0;

        return subset(nums, (sum + S)/2);

    }

    int subset(vector<int>& nums, int sum) {

        int n = nums.size();

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

        vector<int> dp(sum+1, 0);

        dp[0] = 1;

        for(int i=1; i<=n; ++i) {

            for(int j=sum; j>=0; --j) {   // 细节，j从0开始

                if(j>=nums[i-1]) {

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

                }

            }

        }

        return dp[sum];

    }

};

// 压缩空间版本

## 416. 分割等和子集

class Solution {

public:

    bool canPartition(vector<int>& nums) {

        int sum = accumulate(nums.begin(), nums.end(), 0);

        if(sum%2 == 1) return false;

        int n = nums.size();

        int target = sum/2;

        vector<vector<bool>> dp(n+1, vector<bool>(target + 1, false));

        for(int i=0; i<=n; ++i) {

            dp[i][0] = true;

        }

        for(int i=1; i<=n; ++i) {

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

                if(j<nums[i-1]) {

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

                } else {

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

                }

            }

        }

        return dp[n][target];

    }

};

压缩空间 ：

class Solution {

public:

    bool canPartition(vector<int>& nums) {

        int sum = accumulate(nums.begin(), nums.end(), 0);

        if(sum%2 == 1) return false;

        int n = nums.size();

        int target = sum/2;

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

        dp[0] = true;

        for(int i=1; i<=n; ++i) {

            for(int j=target; j>=0; --j) {

                if(j>=nums[i-1]) {

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

                }

            }

        }

        return dp[target];

    }

};