LeetCode题

一、栈

1.1 栈的应用

> 简化路径

```
string simplifyPath(string path) {
    string res;
    stack<string> stks;
    int n = path.size();
    for (int i = 0, j = 0; i < n;) {
        if (path[j] == '/') {
            while (j < n && path[j] == '/') j++;
            i = j;
        }
        while (j < n && path[j] != '/') j++;
        if (i >= n)     break;
        string nw = path.substr(i, j - i);
        if (nw == ".");
        else if (nw == "..") {
            if (stks.size()) stks.pop();
        }
        else {
            stks.push(nw);
        }
        i = j + 1;
    }
    if (stks.empty())    return "/";
    while (stks.size()) {
        string tmp = stks.top();
        reverse(tmp.begin(), tmp.end());
        res += tmp + '/';
        stks.pop();
    }
    reverse(res.begin(), res.end());
    return res;
}
```

> 验证二叉树的前序序列化

```
// 核心思路：  二叉树正确的条件就是每个节点有两个子节点（null）
bool isValidSerialization(string preorder) {
    stack<char> stk;
    stk.push(1);
    for(int i = 0; i < preorder.size(); i++){
        if(stk.empty()) return false;
        if(preorder[i] == '#'){
            stk.top() -= 1;
            if(stk.top() == 0)  stk.pop();
            i++;
            continue;
```

```
        }
        int j = i;
        int num = 0;
        while(j < preorder.size() && preorder[j] != ','){
            num = preorder[j] - '0' + num*10;
            j++;
        }
        stk.top() -= 1;
        if(stk.top() == 0)  stk.pop();
        stk.push(2);
        i = j;
    }
    return stk.empty();
}
```

> 函数的独占时间

```
vector<int> exclusiveTime(int n, vector<string>& logs) {
    vector<int> res;
    res.resize(n, 0);
    stack<pair<int, int>> stk;
    for (int i = 0; i < logs.size(); i++) {
        int logsize = logs[i].size();
        int pro = 0;
        int time = 0;
        string sta;
        //第一段
        int prepos = 0;
        int pos = logs[i].find(':', prepos);
        pro = atoi(logs[i].substr(prepos, pos - prepos).c_str());
        //第二段
        prepos = pos + 1;
        pos = logs[i].find(':', prepos);
        sta = logs[i].substr(prepos, pos - prepos);
        //第三段
        time = atoi(logs[i].substr(pos + 1).c_str());

        if (sta == "start") {
            if (stk.size()) {
                res[stk.top().first] += time - stk.top().second;
            }
            stk.push({ pro, time });
        }
        else {
            auto nw = stk.top();
            stk.pop();
            res[nw.first] += time + 1- nw.second;
            if (stk.size()) {
                stk.top().second = time + 1;
            }
        }
    }
    return res;
}
```

> 最短无序连续子数组

```
int findUnsortedSubarray(vector<int>& nums) {
    int len = nums.size();
    int begin = -1;
    int end = -1;
    int minv = nums[len-1];
    int maxv = nums[0];
    for(int i = 0; i < len; i++){
        if(nums[i] >= maxv){
            maxv = nums[i];
        }
        else{
            end = i;
        }
        if(nums[len-i-1] <= minv){
            minv = nums[len-i-1];
        }
        else{
            begin = len-i-1;
        }
    }
    return begin == -1 ? 0 : end-begin+1;
}
```

> 移除无效的括号

```
string smallestSubsequence(string s) {
    vector<int> cnt(26);
    stack<int> stk;
    int n = s.size();
    unordered_map<int, int> um;
    for (int i = 0; i < n; i++) {
        int u = s[i] - 'a';
        cnt[u]++;
        um[u] = 0;
    }
    for (int i = 0; i < n; i++) {
        int u = s[i] - 'a';
        if(um[u] == 0){
            while (stk.size() && cnt[stk.top()] > 0 && stk.top() > u) {
                um[stk.top()]--;
                stk.pop();
            }
            stk.push(u);
            um[u]++;
        }
        cnt[u]--;
    }
    string res;
    while (stk.size()) {
        res += (char)(stk.top() + 'a');
        stk.pop();
    }
    reverse(res.begin(), res.end());
```

```
    return res;
}
```

> 去除重复字母

```
string smallestSubsequence(string s) {
    vector<int> cnt(26);
    stack<int> stk;
    int n = s.size();
    unordered_map<int, int> um;
    for (int i = 0; i < n; i++) {
        int u = s[i] - 'a';
        cnt[u]++;
        um[u] = 0;
    }
    for (int i = 0; i < n; i++) {
        int u = s[i] - 'a';
        if(um[u] == 0){
            while (stk.size() && cnt[stk.top()] > 0 && stk.top() > u) {
                um[stk.top()]--;
                stk.pop();
            }
            stk.push(u);
            um[u]++;
        }
        cnt[u]--;
    }
    string res;
    while (stk.size()) {
        res += (char)(stk.top() + 'a');
        stk.pop();
    }
    reverse(res.begin(), res.end());
    return res;
}
```

1.2 单调栈

> 接雨水、每日温度、下一个更大元素I、去除重复字母、股票价格跨度、移掉K位数字、最短无序连续子数组、柱状图中最大的矩形

- 当元素出栈时，说明**新元素**是出栈元素**向后**找第一个比其小（大）的元素
- 当元素出栈后，说明**新栈顶元素**是出栈元素**向前**找第一个比其小（大）的元素

```
//判断什么时候应该用单调递增的栈，什么时候应该用单调递减的栈：
//往前走找第一个比自己 大 的元素，用单调递减的栈，1. 也就是 decStack.top() <= price,
decStack.pop()
//往前走找第一个比自己 小 的元素，用单调递增的栈，1. 也就是 incStack.top() >= price,
incStack.pop()
vector<int> vi{100, 80, 60, 70, 60, 75, 85}
stack<int> stk;
for(int i = 0; i < n; i++){
    // 单调增
    while(stk.size() && stk.top() >= vi[i]){
        stk.pop();
    }
    stk.push(vi[i]);
}
```

> 股票价格跨度

```
class StockSpanner {
public:
    stack<int> stk;
    stack<int> weights;
    StockSpanner() {
    }
    int next(int price) {
        int res = 1;
        // 维护了一个单调递减栈
        while(stk.size() && stk.top() <= price){
            stk.pop();
            // 不符合单调递减的就抹除掉，记下它的日子就行。
            res += weights.top();
            weights.pop();
        }
        stk.push(price);
        weights.push(res);
        return res;
    }
};
// 普通解法
class StockSpanner {
public:
    vector<int> vi;
    vector<int> nextv;
    StockSpanner() {
    }
    int next(int price) {
        int res = 1;
        int n = vi.size() - 1;
        while(n >= 0){
            if(vi[n] <= price){
                res += nextv[n];
                n -= nextv[n];
            }
            else{
                break;
            }
```

```
        }
        vi.push_back(price);
        nextv.push_back(res);
        return res;
    }
};
```

> 柱状图中最大的矩形

```
// 没懂 待研究
//高固定下来，求宽
int largestRectangleArea(vector<int>& heights) {
    int n = heights.size();
    int maxr = 0;
    // 往前大于的个数
    stack<int> incStk;
    stack<int> numStk;
    vector<int> numv;
    for(int i = 0; i < n; i++){
        int res = 1;
        // 找之前第一个小于的 要单调递增栈
        while(incStk.size() && incStk.top() >= heights[i]){
            incStk.pop();
            res += numStk.top();
            numStk.pop();
        }
        incStk.push(heights[i]);
        numStk.push(res);
        numv.push_back(res);
    }
    // 往后大于的个数
    stack<int> incStk2;
    stack<int> numStk2;
    vector<int> numv2;
    for (int i = n - 1; i >= 0; i--) {
        int res = 1;
        // 找之前第一个小于的 要单调递增栈
        while (incStk2.size() && incStk2.top() >= heights[i]) {
            incStk2.pop();
            res += numStk2.top();
            numStk2.pop();
        }
        incStk2.push(heights[i]);
        numStk2.push(res);
        numv2.push_back(res);
    }
    for(int i = 0; i < n; i++){
        maxr = max(maxr, (numv[i]+numv2[n - 1 - i]-1) * heights[i]);    //注意索引
的正反
    }
    return maxr;
}
//
int largestRectangleArea(vector<int>& heights) {
    stack<int> stki;
```

```
    heights.push_back(0);
    int n = heights.size();
    int maxr = 0;
    for(int i = 0; i < n; i++){
        while(stki.size() && heights[stki.top()] > heights[i]){
            int tp = stki.top();
            stki.pop();
            if(stki.empty())    maxr = max(maxr, i * heights[tp]);
            else    maxr = max(maxr, (i - stki.top() - 1)*heights[tp]);
        }
        stki.push(i);
    }
    return maxr;
}
```

> 下一个更大元素I

```
vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
    int n1 = nums1.size();
    int n2 = nums2.size();
    unordered_map<int, int> um;
    for(int i = 0; i < n1; i++){
        um[nums1[i]] = -1;
    }
    stack<int> stk;
    for(int i = 0; i < n2; i++){
        // 单调递减
        while(stk.size() && stk.top() <= nums2[i]){
            um[stk.top()] = nums2[i];
            stk.pop();
        }
        stk.push(nums2[i]);
    }
    vector<int> res;
    for(int i = 0; i < n1; i++){
        res.push_back(um[nums1[i]]);
    }
    return res;
}
```

> 每日温度

```
vector<int> dailyTemperatures(vector<int>& temperatures) {
    stack<int> stk;
    vector<int> res(temperatures.size(), 0);
    int n = temperatures.size();
    for(int i = 0; i < n; i++){
        while(stk.size() && temperatures[stk.top()] < temperatures[i]){
            int idx = stk.top();
            stk.pop();
            res[idx] = i - idx;
        }
        stk.push(i);
    }
```

```
    return res;
}
```

> 移掉 K 位数字

```
string removeKdigits(string num, int k) {
    stack<char> stk;
    int n = num.size();
    for(int i = 0; i < n; i++){
        while(stk.size() && stk.top() > num[i] && k){
            stk.pop();
            k--;
        }
        stk.push(num[i]);
    }
    while(k && stk.size()){
        stk.pop();
        k--;
    }
    if(stk.empty()) return "0";
    string res;
    while(stk.size()){
        res += stk.top();
        stk.pop();
    }
    reverse(res.begin(), res.end());
    while(res.front() == '0' && res.size() > 1)   res.erase(0,1);
    return res;
}
```

S

1.3

二、字符串

2.1 Trie树

> 添加与搜索单词 - 数据结构设计

```
//Trie树
class WordDictionary {
public:
    const static int N = 3e5;
    int vi[N][26];
    int cnt[N];
    int idxx = 0;
    WordDictionary() {
        memset(vi, 0, sizeof(vi));
```

```
        memset(cnt, 0, sizeof(cnt));
    }
    void addWord(string word) {
        int p = 0;
        for(int i = 0; i < word.size(); i++){
            int u = word[i] - 'a';
            if(!vi[p][u]){
                vi[p][u] = ++idxx;
            }
            p = vi[p][u];
        }
        cnt[p] = 1;
    }
    bool search(string word) {
        return dfs(word, 0, 0);
    }
    bool dfs(string& word, int idx, int p){
        if(idx >= word.size()){
            if(cnt[p] == 1) return true;
            return false;
        }
        if(word[idx] == '.'){
            for(int i = 0; i < 26; i++){
                if(vi[p][i]){
                    bool rt = dfs(word, idx+1, vi[p][i]);
                    if(rt)  return true;
                }
            }
        }
        else{
            int u = word[idx] - 'a';
            if(vi[p][u]){
                bool rt = dfs(word, idx+1, vi[p][u]);
                if(rt)  return true;
            }
            else{
                return false;
            }
        }
        return false;
    }
};
```

2.2 回文串

> 分割回文串

```
class Solution {
public:
    int f[16][16];
    vector<vector<string>> vvs;
    vector<string> vs;
    vector<vector<string>> partition(string s) {
        isPalindrome(s);
        dfs(s, 0);
```

```
        return vvs;
    }
    void dfs(string& s, int begin){
        if(begin >= s.size()){
            vvs.push_back(vs);
            return;
        }
        int end = begin;
        while(end < s.size()){
            if(f[begin][end] == 1){
                vs.push_back(s.substr(begin, end - begin + 1));
                dfs(s, end+1);
                vs.pop_back();
            }
            end++;
        }
    }
    void isPalindrome(string& s) {
        memset(f, 0, sizeof(f));
        for (int i = 0; i < s.size(); i++) {
            f[i][i] = 1;
        }
        int n = s.size();
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < n - i; j++) {
                if (s[j] == s[j + i]) {
                    if (i == 1)     f[j][j + i] = 1;
                    else
                    {
                        f[j][j + i] = f[j + 1][j + i - 1];
                    }
                }
                else {
                    f[j][j + i] = -1;
                }
            }
        }
    }
};
```

> 动态规划判定回文串

```
int f[1000][1000];
void isPalindrome(string& s) {
    memset(f, 0, sizeof(f));
    for (int i = 0; i < s.size(); i++) {
        f[i][i] = 1;
    }
    int n = s.size();
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < n - i; j++) {
            if (s[j] == s[j + i]) {
                if (i == 1){
                    f[j][j + i] = 1;
                }
```

```
                else
                {
                    f[j][j + i] = f[j + 1][j + i - 1];
                }
            }
            else {
                f[j][j + i] = -1;
            }
        }
    }
}
```

2.3 常规题

> 字符串相乘

```
class Solution {
public:
    string multiply(string num1, string num2) {
        if(num1=="0" || num2=="0")  return "0";
        int l1 = num1.size(), l2 = num2.size();
        string mid(201, '0');
        for(int i = l1-1; i >= 0; i--){
            string tmp;
            int r = 0;
            for(int j = l2-1; j >= 0; j--){
                int kk = (num1[i] - '0') * (num2[j] - '0') + r;
                tmp.insert(0, to_string(kk % 10));
                r = kk / 10;
            }
            if(r)   tmp.insert(0, to_string(r));
            tmp.append(l1 - 1 - i, '0');
            add(mid, tmp);
        }
        int idx = 200;
        while(mid[idx] == '0')    idx--;
        string res = mid.substr(0, idx + 1);
        reverse(res.begin(), res.end());
        return res;
    }
    void add(string& all, string& ext){
        int r = 0;
        int ll = ext.size();
        for(int j = ll-1; j >= 0; j--){
            int kk = (all[ll - 1 - j] - '0') + (ext[j] - '0') + r;
            all[ll-1-j] = (char)(kk % 10 + '0');
            r = kk / 10;
        }
        all[ll] = (char)(r + '0');
    }
};
```

2.4 递归

> 括号生成

```
class Solution {
public:
    vector<string> vs;
    vector<string> generateParenthesis(int n) {
        string str = "";
        process(str, n, n);
        return vs;
    }
    void process(string str, int l, int r){
        if(l == 0 && r == 0){
            vs.push_back(str);
            return;
        }
        if(l)   process(str+'(', l-1, r);
        if(l < r)   process(str+')', l, r-1);
    }
};
```

三、树

3.1 最近公共祖先

> 朴素版lca

```
int n, m, s;
const int N = 5e5 + 10, M = 5e5 + 10;
int d[N], f[N];     // 节点深度、节点的父节点
int h[N], e[N * 2], ne[N * 2], idx;

void add(int x, int y) {
    e[idx] = y;
    ne[idx] = h[x];
    h[x] = idx++;
}

void dfs(int u, int fa) {
    f[u] = fa;
    d[u] = d[fa] + 1;
    for (int ii = h[u]; ii != -1; ii = ne[ii]) {
        int j = e[ii];
        if(j != fa) dfs(ii, u);
    }
}

int lca(int u, int v) {
    if (d[u] > d[v]) {
        swap(u, v);
```

```
    }
    while (d[u] < d[v]) {
        v = f[v];
    }
    while (u != v) {
        u = f[u], v = f[v];
    }
    if (u == -1)    u = s;
    return u;
}

int main() {
    cin >> n >> m >> s;
    memset(h, -1, sizeof(h));
    for (int i = 1; i < n; i++) {
        int x, y;
        cin >> x >> y;
        add(x, y);
        add(y, x);
    }
    dfs(s, -1);
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        cout << lca(a, b) << endl;
    }
}
```

> 倍增lca/祖孙查询

```
// 倍增lca
const int N = 4e4 + 10, M = N * 2;
int n, root;
int f[N][16], d[N];
int h[N], e[M], ne[M], idx;
int q[N];

void add(int x, int y) {
    e[idx] = y;
    ne[idx] = h[x];
    h[x] = idx++;
}
void bfs(int root) {
    memset(d, 0x3f, sizeof(d));
    d[0] = 0, d[root] = 1;
    int hh = 0, tt = -1;
    q[++tt] = root;
    while (hh <= tt) {
        int t = q[hh++];
        for (int ii = h[t]; ii != -1; ii = ne[ii]) {
            int j = e[ii];
            if (d[j] > d[t] + 1) {
                d[j] = d[t] + 1;
                q[++tt] = j;
                f[j][0] = t;
```

```
                for (int k = 1; k <= 15; k++) {
                    f[j][k] = f[f[j][k - 1]][k - 1];
                }
            }
        }
    }
}

int lca(int a, int b) {
    if (d[a] < d[b]) swap(a, b);
    for (int k = 15; k >= 0; k--) {
        if (d[f[a][k]] >= d[b]) {
            a = f[a][k];
        }
    }
    if (a == b) return a;
    for (int k = 15; k >= 0; k--) {
        if (f[a][k] != f[b][k]) {
            a = f[a][k];
            b = f[b][k];
        }
    }
    return f[a][0];
}

int main() {
    cin >> n;
    memset(h, -1, sizeof(h));
    for (int i = 0; i < n; i++) {
        int x, y;
        cin >> x >> y;
        if (y == -1)    root = x;
        else {
            add(x, y);
            add(y, x);
        }
    }
    bfs(root);
    int m;
    cin >> m;
    while (m--) {
        int a, b;
        cin >> a >> b;
        int p = lca(a, b);
        if (p == a)         cout << 1 << endl;
        else if (p == b)        cout << 2 << endl;
        else
        {
            cout << 0 << endl;
        }
    }
}
```

> 二叉树的最近公共祖先

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root || root==p || root==q) return root;
        //
        auto left = lowestCommonAncestor(root->left, p, q);
        auto right = lowestCommonAncestor(root->right, p, q);
        if(!left)   return right;
        if(!right)  return left;
        else    return root;
    }
};
```

3.2 回溯

> 路径总和II

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> vi;
    int ts;
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        if(!root)   return res;
        ts = targetSum;
        dfs(root, 0);
        return res;
    }
    void dfs(TreeNode* root, int nwSum){
        vi.push_back(root->val);
        int sum = root->val + nwSum;
        if(!root->left && !root->right && sum == ts){
            res.push_back(vi);
        }
        if(root->left)  dfs(root->left, sum);
        if(root->right) dfs(root->right, sum);
        vi.pop_back();
    }
};
```

> 路径总和III

```
class Solution {
public:
    int ts;
    unordered_map<long long, int> um;
    int pathSum(TreeNode* root, int targetSum) {
        ts = targetSum;
        um[0] = 1;
        int res = dfs(root, 0);
        return res;
    }
    int dfs(TreeNode* root, long long nsum){
        if(!root)   return 0;
```

```
        int res = 0;
        nsum +=  root->val;
        res += um[nsum - ts];
        um[nsum]++;
        int ln = dfs(root->left, nsum);
        int rn = dfs(root->right, nsum);
        um[nsum]--;
        return res + ln + rn;
    }
};
```

3.3 各种遍历

> 二叉树展开为链表

```
class Solution {
public:
    TreeNode* pre = NULL;
    void flatten(TreeNode* root) {
        rightPostOrder(root);
    }
    void rightPostOrder(TreeNode* root){
        if(!root)   return;
        rightPostOrder(root->right);
        rightPostOrder(root->left);
        root->right = pre;
        root->left = NULL;
        pre = root;
    }
};
```

```
// 非递归 待补充
```

> 有序链表转换二叉搜索树

```
class Solution {
public:
    int lnl = 0;
    ListNode* nw = NULL;
    TreeNode* sortedListToBST(ListNode* head) {
        ListNode* lp = head;
        while(lp){
            lnl++;
            lp = lp->next;
        }
        if(lnl == 0)    return NULL;
        TreeNode* rt;
        nw = head;
        inorder(rt, 1);
        return rt;
    }
    void inorder(TreeNode*& root, int idx){
        if(idx > lnl)   return;
        root = new TreeNode();
```

```
        inorder(root->left, 2*idx);
        root->val = nw->val;
        nw = nw->next;
        inorder(root->right, 2*idx+1);
    }
};
```

3.4 经典递归思路

> 二叉树展开为链表

```
class Solution {
public:
    void flatten(TreeNode* root) {
        dosome(root);
    }
    void dosome(TreeNode* root){
        while(root){
            if(!root->left){
                root = root->right;
                continue;
            }
            TreeNode* p = root->left;
            while(p->right){
                p = p->right;
            }
            p->right = root->right;
            root->right = root->left;
            root->left = NULL;          // 这里别忘了
            root = root->right;
        }
    }
};
```

> 完全二叉树的节点个数

```
class Solution {
public:
    int res = 0;
    int countNodes(TreeNode* root) {
        // 树的深度
        int depth = 0;
        TreeNode* p = root;
        while(p){
            depth++;
            p = p->left;
        }
        // 特判
        if(depth == 0)  return 0;
        process(root, depth, 1);
        return res;
    }
```

```
    // 当前节点root 当前节点高度height 当前节点编号nw
    void process(TreeNode* root, int height, int nw){
        TreeNode* p = root -> right;
        if(!p){
            // 在没有右子树的情况下：
            // 如果当前节点高度>1，说明它有左子树，那么答案就是左子树编号
            // 否则当前节点就是最大编号的节点
            if(height > 1)
                res = nw * 2;
            else
                res = nw;
            return;
        }
        // 右子树最左节点深度（相对于当前节点）
        int rmld = 1;
        while(p->left){
            rmld++;
            p = p->left;
        }
        // 如果右子树最左节点深度能达到当前树的最底层则答案在当前节点的右子树中寻找，反之在左子树中寻找。
        if(rmld == height - 1){
            process(root->right, height - 1, nw * 2 + 1);
        }
        else{
            process(root->left, height - 1, nw * 2);
        }
    }
};
```

> 后继者

```
// 二叉搜索树某节点的后继结点：
// 如果该节点有右子树,则后继节点是该节点右子树的最左节点
// 否则,后继节点就是第一个右祖先。
class Solution {
public:
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        if(!root)   return root;
        if(p->val >= root->val){
            return inorderSuccessor(root->right, p);
        }
        TreeNode* res = inorderSuccessor(root->left, p);
        return res == NULL ? root : res;
    }
};
```

> 最深叶节点的最近公共祖先

```
class Solution {
public:
    //求当前结点的深度
    int depth(TreeNode* root){
        if(!root) return 0;
```

```
        int left = depth(root->left);
        int right = depth(root->right);

        return 1+max(left, right);

    }
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        if(!root) return NULL;

        int left = depth(root->left); //左树深度
        int right = depth(root->right); //右树深度

        if(left == right){ //如果左右两边深度相等，那么当前结点就是最近公共祖先
            return root;
        }
        else if(left>right){ //左边更深，那么就去左边找
            return lcaDeepestLeaves(root->left);
        }
        else{
            return lcaDeepestLeaves(root->right);
        }
    }
};
// mine
class Solution {
public:
    unordered_map<TreeNode*, int> um;
    TreeNode* lcaDeepestLeaves(TreeNode* root) {
        int maxh = dfs1(root);
        return dfs2(root, maxh-1);
    }
    int dfs1(TreeNode* rt){
        if(!rt)     return 0;
        int ld = dfs1(rt->left);
        int rd = dfs1(rt->right);
        int dep = max(ld, rd)+1;
        um[rt] = dep;
        return dep;
    }
    TreeNode* dfs2(TreeNode* rt, int height){
        if(!rt->left && !rt->right) return rt;
        if(rt->left && um[rt->left] == height && rt->right && um[rt->right] ==
height)    return rt;
        if(rt->left && um[rt->left] == height)    return dfs2(rt->left, height-
1);
        if(rt->right && um[rt->right] == height)    return dfs2(rt->right,
height-1);
        return NULL;
    }
};
```

四、队列

4.1 设计循环队列

```
class MyCircularQueue {
public:
    int hh = 0;
    int tt = -1;
    int n;  //队列长度为n;
    int size = 0;
    vector<int> vi;
    MyCircularQueue(int k) {
        n = k;
        vi.resize(n + 1);
    }

    bool enQueue(int value) {
        if(isFull())    return false;
        if(tt == n) tt = -1;
        vi[++tt] = value;
        return true;
    }

    bool deQueue() {
        if(isEmpty())   return false;
        if(hh == n) hh = -1;
        hh++;
        return true;
    }

    int Front() {
        if(isEmpty())   return -1;
        return vi[hh];
    }

    int Rear() {
        if(isEmpty())   return -1;
        return vi[tt];
    }
    bool isEmpty() {
        int sz = n + 1;
        // 这样能防止队列长度为1或0
        return (tt + 1 + sz) % sz == hh;
    }

    bool isFull() {
        int sz = n + 1;
        return (tt + 2 + sz) % sz == hh;
    }
};
//
class MyCircularQueue {
public:
    int hh = 0;
    int tt = -1;
```

```
    int n;  //队列长度为n;
    int size = 0;
    vector<int> vi;
    MyCircularQueue(int k) {
        n = k;
        vi.resize(n);
    }
    bool enQueue(int value){
        if(isFull())    return false;
        tt = (tt+1) % n;
        vi[tt] = value;
        size++;
        return true;
    }
    bool deQueue() {
        if(isEmpty())   return false;
        hh = (hh+1) % n;
        size--;
        return true;
    }
    int Front() {
        if(isEmpty())   return -1;
        return vi[hh];
    }
    int Rear() {
        if(isEmpty())   return -1;
        return vi[tt];
    }
    bool isEmpty1(){
        return size == 0;
    }
    bool isFull1(){
        return  size == n;
    }
};
```

4.2 设计循环双端队列

```
class MyCircularDeque {
public:
    int hh = 0;
    int tt = -1;
    int n;
    int size = 0;
    vector<int> vi;
    MyCircularDeque(int k) {
        vi.resize(k + 1);
        n = k;
    }

    bool insertFront(int value) {
        if (isFull())    return false;
        if (tt == hh - 1 || tt == (hh - 1 + n) % n) {
            tt = hh;
            vi[hh] = value;
```

```
        }
        else {
            if (hh == 0)     hh = n + 1;
            vi[--hh] = value;
        }
        size++;
        return true;
    }

    bool insertLast(int value) {
        if (isFull())   return false;
        if (tt == n)     tt = -1;
        vi[++tt] = value;
        size++;
        return true;
    }

    bool deleteFront() {
        if (isEmpty())   return false;
        if (hh == n)     hh = -1;
        ++hh;
        size--;
        return true;
    }

    bool deleteLast() {
        if (isEmpty())   return false;
        if (tt == 0)     tt = n + 1;
        --tt;
        size--;
        return true;
    }

    int getFront() {
        if(isEmpty())   return -1;
        return vi[hh];
    }

    int getRear() {
        if(isEmpty())   return -1;
        return vi[tt];
    }

    bool isEmpty() {
        return size == 0;
    }

    bool isFull() {
        return size == n;
    }
};
```

4.3 单调队列

```
int n = nums.size();
int hh = 0;
int tt = -1;
vector<int> que(n);
for(int i = 0; i < n; i++){
    if(i - que[hh] == k){
        hh++;
    }
    while(hh <= tt && nums[que[tt]] <= nums[i]){
        tt--;
    }
    que[++tt] = i;
}
```

4.4 滑动窗口最大值

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    int n = nums.size();
    int hh = 0;
    int tt = -1;
    vector<int> res;
    vector<int> que(n);
    for(int i = 0; i < n; i++){
        if(i - que[hh] == k){
            hh++;
        }
        while(hh <= tt && nums[que[tt]] <= nums[i]){
            tt--;
        }
        que[++tt] = i;
        if(i >= k - 1)  res.push_back(nums[que[hh]]);
    }
    return res;
}
```

4.5 按递增顺序显示卡牌

```
vector<int> deckRevealedIncreasing(vector<int>& deck) {
    // 2 3 5 7 11 13 17
    // 7
    int n = deck.size();
    deque<int> idxq;
    for(int i = 0; i < n; i++){
        idxq.push_back(i);
    }
    sort(deck.begin(), deck.end());
    vector<int> res(n);
    for(int i = 0; i < n; i++){
        res[idxq.front()] = deck[i];
        idxq.pop_front();
        int ele = idxq.front();
```

```
            idxq.pop_front();
            idxq.push_back(ele);
        }
        return res;
    }
```

4.6绝对差不超过限制的最长连续子数组

```
int longestSubarray(vector<int>& nums, int limit) {
    int res = 0;
    int n = nums.size();
    multiset<int> ms;
    int l = 0, r = 0;
    while(r < n){
        ms.insert(nums[r]);
        while(*ms.rbegin() - *ms.begin() > limit){
            ms.erase(ms.find(nums[l++]));
        }
        res = max(res, r - l + 1);
        r++;
    }
    return res;
}
//
int longestSubarray(vector<int>& nums, int limit) {
    int res = 0;
    int n = nums.size();
    deque<int> queMax, queMin;
    int l = 0, r = 0;
    while(r < n){
        while(queMax.size() && queMax.back() < nums[r]) queMax.pop_back();
        while(queMin.size() && queMin.back() > nums[r]) queMin.pop_back();
        queMax.push_back(nums[r]);
        queMin.push_back(nums[r]);
        while(queMax.size() && queMin.size() && queMax.front() - queMin.front() >
limit){
            if(nums[l] == queMin.front())   queMin.pop_front();
            if(nums[l] == queMax.front())   queMax.pop_front();
            l++;
        }
        res = max(res, r - l + 1);
        r++;
    }
    return res;
}
```

4.7 队列的最大值

```
class MaxQueue {
public:
    vector<int> vi; //原队列的数据
    int begin = 0;  //原队列的头指针
    int end = -1;       //原队列的尾指针
    vector<int> dq; //单调队列
```

```
    int hh = 0;
    int tt = -1;
    MaxQueue() {
        vi.resize(10010);
        dq.resize(10010);
    }

    int max_value() {
        if(begin > end)   return -1;
        return vi[dq[hh]];
    }

    void push_back(int value) {
        vi[++end] = value;
        while(hh <= tt && vi[dq[tt]] < value){
            tt--;
        }
        dq[++tt] = end;
    }

    int pop_front() {
        if(begin > end)   return -1;
        int res = vi[begin];
        if(begin == dq[hh]) hh++;
        begin++;
        return res;
    }
};
```

4.8 知道秘密的人数

```
// 这道题 暂时是DP
class Solution {
public:
    const static int mod = 1e9 + 7;
    int peopleAwareOfSecret(int n, int delay, int forget) {
        long long res = 0;
        vector<long long> f(n+1);
        f[1] = 1;
        for(int i = 2; i <= n; i++){
            // 从 i - delay 天及其之前，有人告诉第i天的人这个秘密
            // 但 i - forget 天及其之前的人都忘记了这个秘密
            for (int j = i - forget + 1; j < i - delay + 1; ++j) {
                if (j > 0) {
                    f[i] += f[j] % mod;
                }
            }
        }
        // 最后只有这些人记得这个秘密
        for (int i = n - forget + 1; i <= n; ++i) {
            res = (res + f[i]) % mod;
        }
        return res;
    }
```

```
};
```

4.9 Dota2 参议院

```
string predictPartyVictory(string senate) {
    deque<int> ri;
    deque<int> di;
    int n = senate.size();
    for (int i = 0; i < n; i++) {
        if (senate[i] == 'R') {
            ri.push_back(i);
        }
        else {
            di.push_back(i);
        }
    }
    while (true) {
        for (int i = 0; i < n; i++) {
            if (ri.empty())  return "Dire";
            if (di.empty())  return "Radiant";
            if (senate[i] == 'R' && ri.front() == i) {
                ri.pop_front();
                ri.push_back(i);
                di.pop_front();
            }
            else if (senate[i] == 'D' && di.front() == i) {
                di.pop_front();
                di.push_back(i);
                ri.pop_front();
            }
            else;
        }
    }
}
```

五、前缀和

5.1最后 K 个数的乘积

```
class ProductOfNumbers {
public:
    vector<int> product;
    int n = 0;
    int zero_idx = -1;
    ProductOfNumbers() {
        product.push_back(1);
    }

    void add(int num) {
        if(num == 0){
            product.push_back(1);
            n++;
```

```
            zero_idx = n;
        }
        else{
            product.push_back(num);
            n++;
            product[n] *= product[n-1];
        }

    }

    int getProduct(int k) {
        if(n - k < zero_idx) return 0;
        return product[n] / product[n-k];
    }
};
//
class ProductOfNumbers {
public:
    #define N 40010
    int len,pre[N];
    ProductOfNumbers() {
        pre[0]=1;
        len=0;
    }

    void add(int num) {
        if (!num) len=0;    //这里直接从头开始的思路真的很赞
        else{
            pre[++len]=num;
            pre[len]*=pre[len-1];
        }
    }

    int getProduct(int k) {
        if (len<k) return 0;
        return pre[len]/pre[len-k];
    }
};
```

六、DP

6.1DFS记忆化搜索

> 单词拆分

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        for(auto& ele : wordDict){
            kset.insert(ele);
        }
        vector<bool> vc(s.size()+1);
        vc[0] = true;
        for(int i = 1; i <= s.size(); i++){
            for(int j = 0; j < i; j++){
```

```
                if(vc[j] && kset.find(s.substr(j, i - j)) != kset.end()){
                    vc[i] = true;
                    break;
                }
            }
        }
        return vc[s.size()];
    }

private:
    unordered_set<string> kset;
};
```

6.2 树形DP

> 不同的二叉搜索树

```
// 左右子树都是连续递增的值，所以可以忽略数字特征，只管数量。
class Solution {
public:
    int numTrees(int n) {
        vector<int> f(n+1, 0);
        f[0] = 1;
        f[1] = 1;
        // i代表节点数 j代表左子树的节点数
        for(int i = 2; i <= n; i++){
            for(int j = 0; j < i; j++){
                f[i] += f[j] * f[i - j - 1];
            }
        }
        return f[n];
    }
};
```

七、二进制

> 重复的DNA序列

```
unordered_map<char, int> bin = {{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};
unordered_map<int, int> cnt;

vector<string> findRepeatedDnaSequences(string s) {
    vector<string> vs;
    int n = s.size();
    if (n <= 10)
        return vs;
    int x = 0;
    for (int i = 0; i < 10 - 1; i++) {
        x = (x << 2) | bin[s[i]];
```

```
    }
    for (int i = 0; i <= n - 10; i++) {
        x = ((x << 2) | bin[s[i + 10 - 1]]) & ((1 << 20) - 1);
        if (++cnt[x] == 2) {
            vs.push_back(s.substr(i, 10));
        }
    }
    return vs;
}
```

八、排序

8.1跟排序没啥关系的题

> 三数之和

```
class Solution {
public:
    unordered_set<int> kset;
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> res;
        int n = nums.size();
        if(n < 3)   return res;
        sort(nums.begin(), nums.end());
        for(auto ele : nums){
            kset.insert(ele);
        }
        int idx1 = n; // 第一个正数的索引
        while(idx1 > 0 && nums[idx1 - 1] > 0){
            idx1--;
        }
        int idx2 = -1;   // 最后一个负数的索引
        while(idx2 < n - 1 && nums[idx2+1] < 0){
            idx2++;
        }
        if(idx1 == n || idx2 == -1){
            if(idx1 - idx2 > 3){
                vector<int> vc{0, 0, 0};
                res.push_back(vc);
            }
            return res;
        }
        if(idx1 - idx2 > 3){
            vector<int> vc{0, 0, 0};
            res.push_back(vc);
        }
        if(idx1 - idx2 > 1){
            for(int i = 0; i <= idx2; i++){
                if(i > 0 && nums[i] == nums[i-1])   continue;
                if(kset.find(-nums[i]) != kset.end()){
                    vector<int> vc;
                    vc.push_back(nums[i]);
                    vc.push_back(0);
```

```
                    vc.push_back(-nums[i]);
                    res.push_back(vc);
                }
            }
        }
        for(int i = 0; i < idx2; i++){
            if(i > 0 && nums[i] == nums[i-1])   continue;
            for(int j = i+1; j <= idx2; j++){
                if(j > i + 1 && nums[j] == nums[j-1])   continue;
                if(kset.find(-(nums[i] + nums[j])) != kset.end()){
                    vector<int> vc;
                    vc.push_back(nums[i]);
                    vc.push_back(nums[j]);
                    vc.push_back(-nums[i]-nums[j]);
                    res.push_back(vc);
                }
            }
        }
        for(int i = idx1; i < n-1; i++){
            if(i > 0 && nums[i] == nums[i-1])   continue;
            for(int j = i+1; j < n; j++){
                if(j > i + 1 && nums[j] == nums[j-1])   continue;
                if(kset.find(-(nums[i] + nums[j])) != kset.end()){
                    vector<int> vc;
                    vc.push_back(nums[i]);
                    vc.push_back(nums[j]);
                    vc.push_back(-nums[i]-nums[j]);
                    res.push_back(vc);
                }
            }
        }
        return res;
    }
};
```

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums)
    {
        int size = nums.size();
        if (size < 3)   return {};          // 特判
        vector<vector<int> >res;            // 保存结果（所有不重复的三元组）
        std::sort(nums.begin(), nums.end());// 排序（默认递增）
        for (int i = 0; i < size; i++)      // 固定第一个数，转化为求两数之和
        {
            if (nums[i] > 0)    return res; // 第一个数大于 0，后面都是递增正数，不可能
相加为零了
            // 去重：如果此数已经选取过，跳过
            if (i > 0 && nums[i] == nums[i-1])  continue;
            // 双指针在nums[i]后面的区间中寻找和为0-nums[i]的另外两个数
            int left = i + 1;
            int right = size - 1;
            while (left < right)
            {
                if (nums[left] + nums[right] > -nums[i])
```

```
                    right--;    // 两数之和太大，右指针左移
                else if (nums[left] + nums[right] < -nums[i])
                    left++;     // 两数之和太小，左指针右移
                else
                {
                    // 找到一个和为零的三元组，添加到结果中，左右指针内缩，继续寻找
                    res.push_back(vector<int>{nums[i], nums[left], nums[right]});
                    left++;
                    right--;
                    // 去重：第二个数和第三个数也不重复选取
                    // 例如：[-4,1,1,1,2,3,3,3], i=0, left=1, right=5
                    while (left < right && nums[left] == nums[left-1])  left++;
                    while (left < right && nums[right] == nums[right+1])
 right--;
                }
            }
        }
        return res;
    }
};
```

> 多数元素II

```
// 摩尔投票法 不同元素对冲抵消，最后剩下的就是多数元素。
class Solution {
public:
    vector<int> majorityElement(vector<int>& nums) {
        vector<int> res;
        int ele1 = 0, ele2 = 0;
        int vote1 = 0, vote2 = 0;
        for(int num: nums){
            if(vote1 > 0 && ele1 == num){
                vote1++;
            }
            else if(vote2 > 0 && ele2 == num){
                vote2++;
            }
            else if(vote1 == 0){
                vote1++;
                ele1 = num;
            }
            else if(vote2 == 0){
                vote2++;
                ele2 = num;
            }
            else{
                vote1--;
                vote2--;
            }
        }
        int cnt1 = 0, cnt2 = 0;
        for(int num : nums){
            if(vote1 > 0 && ele1 == num){
                cnt1++;
            }
```

```
            else if(vote2 > 0 && ele2 == num){
                cnt2++;
            }
        }
        if(vote1 > 0 && cnt1 > nums.size() / 3){
            res.push_back(ele1);
        }
        if(vote2 > 0 && cnt2 > nums.size() / 3){
            res.push_back(ele2);
        }
        return res;
    }
};
```

九、链表

9.1 双指针

> 环形链表II

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        // 性质：快慢指针2:1，第一次相遇时，慢指针走了nb步。（b是环长）
        // 因为快指针比慢指针多走了n圈，且步数是二倍。
        ListNode* f = head, *s = head;
        while(true){
            if(f == NULL || f->next ==NULL)   return NULL;
            f = f->next->next;
            s = s->next;
            if(f == s)  break;
        }
        f = head;
        while(f != s){
            f = f->next;
            s = s->next;
        }
        return f;
    }
};
```

一百、散题

> 后缀表达式转中缀表达式

```
/*括号的添加方法：
```

```
（1）对于当前节点，如果其左节点是操作符，且运算优先级小于当前节点操作符，则给左子树的中缀表达式加括号；
（2）对于当前节点，如果其右节点是操作符，且运算优先级小于等于当前节点操作符，则给右子树的中缀表达式加括号；
*/
using namespace std;
// 100 25 + 27 25 - / 248 + 201 -
// (100+25)/(27-25)+248-201
struct TreeNode {
        string str;
        TreeNode *left;
        TreeNode *right;
    public:
        TreeNode() {
            str = "";
            left = right = NULL;
        }
        TreeNode(string _str) {
            str = _str;
            left = right = NULL;
        }
        TreeNode(string _str, TreeNode *_left, TreeNode *_right) {
            str = _str;
            left = _left;
            right = _right;
        }
} *rt;
stack<double> num;
stack<TreeNode *> stk;
bool isSymbol(string &str) {
    if (str == "+" || str == "-" || str == "*" || str == "/")
        return true;
    return false;
}
int getPriority(string &str) {
    char c = str[0];
    switch (c) {
        case '*':
        case '/':
            return 1;
        case '+':
        case '-':
            return 0;
        default:
            return -1;
    }
}
bool greaterr(string &a, string &b) {
    return getPriority(a) > getPriority(b);
}
TreeNode *createTree(string &line) {
    int n = line.size();
    TreeNode *p;
    for (int i = 0; i < n; i++) {
        int j = i;
        double number = 0;
```

```
        bool isd = false;
        while (j < n && line[j] != ' ') {
            if (isdigit(line[j])) {
                isd = true;
                number = number * 10.0 + line[j] - '0';
            } else {
                char c = line[j];
                string s(1, c);
                TreeNode *right = stk.top();
                stk.pop();
                TreeNode *left = stk.top();
                stk.pop();
                p = new TreeNode(s, left, right);
                stk.push(p);
                if (c == '+') {
                    double a = num.top();
                    num.pop();
                    double b = num.top();
                    num.pop();
                    double tmp = a + b;
                    num.push(tmp);
                } else if (c == '-') {
                    double a = num.top();
                    num.pop();
                    double b = num.top();
                    num.pop();
                    double tmp = b - a;
                    num.push(tmp);
                } else if (c == '*') {
                    double a = num.top();
                    num.pop();
                    double b = num.top();
                    num.pop();
                    double tmp = b * a;
                    num.push(tmp);
                } else {
                    double a = num.top();
                    num.pop();
                    double b = num.top();
                    num.pop();
                    double tmp = b / a;
                    num.push(tmp);
                }
            }
            j++;
        }
        if (isd) {
            num.push(number);
            stk.push(new TreeNode(to_string((int)number)));
        }
        i = j;
    }
    return p;
}
void inorder(TreeNode *root) {
    if (root) {
```

```
        if (root->left && isSymbol(root->left->str) && greaterr(root->str, root->left->str)) {
            cout << '(';
            inorder(root->left);
            cout << ')';
        } else {
            inorder(root->left);
        }
        cout << root->str;
        if (root->right && isSymbol(root->right->str) && !greaterr(root->right->str, root->str)) {
            cout << '(';
            inorder(root->right);
            cout << ')';
        } else {
            inorder(root->right);
        }
    }
}
int main() {
    string line;
    getline(cin, line);
    rt = createTree(line);
    inorder(rt);
    printf("\n%.2lf", num.top());
}
```