链表、栈与递归

七月算法 邹博

2015年4月21日

链表相加

- 给定两个链表，分别表示两个非负整数。它们的数字逆序存储在链表中，且每个结点只存储一个数字，计算两个数的和，并且返回和的链表头指针。
  - 如：输入：2->4->3、5->6->4，输出：7->0->8

问题分析

- 输入：2->4->3、5->6->4，输出：7->0->8
- 因为两个数都是逆序存储，正好可以从头向后依次相加，完成"两个数的竖式计算"。

Code

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x): val(x),
                     next(nullptr)
    { }
};
```

```
class Solution {
public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        ListNode dummy(-1); // 头节点
        int carry = 0;
        ListNode *prev = &dummy;
        for (ListNode *pa = l1, *pb = l2;
             pa != nullptr || pb != nullptr;
             pa = pa == nullptr ? nullptr : pa->next,
             pb = pb == nullptr ? nullptr : pb->next,
             prev = prev->next) {
            const int ai = pa == nullptr ? 0 : pa->val;
            const int bi = pb == nullptr ? 0 : pb->val;
            const int value = (ai + bi + carry) % 10;
            carry = (ai + bi + carry) / 10;
            prev->next = new ListNode(value); // 尾插法
        }
        if (carry > 0)
            prev->next = new ListNode(carry);
        return dummy.next;
    }
};
```

说明

- 因为两个数字求和的范围是[0,18]，进位最大是1，从而，第i位相加不会影响到第i+2位的计算。事实上，上述代码可以在发现一个链表为空后，直接结束for循环。最后只需要进位和较长链表的当前结点相加，较长链表的其他结点直接拷贝到最终结果即可。
  - 没有提高时间复杂度，trick而已。

链表的部分翻转

- 给定一个链表，翻转该链表从m到n的位置。要求直接翻转而非申请新空间。
  - 如：给定1->2->3->4->5，m=2，n=4，返回1->4->3->2->5。假定给出的参数满足：1≤m≤n≤链表长度。

分析

- 空转m次，找到第m个结点，即开始翻转的链表头部，记做head；
- 以head为起始结点遍历n-m次，将第i次时，将找到的结点插入到head的next中即可。
  - 即头插法

Code

```
class Solution {
public:
    ListNode *reverseBetween(ListNode *head, int m, int n) {
        ListNode dummy(-1);
        dummy.next = head;

        ListNode *prev = &dummy;
        for (int i = 0; i < m-1; ++i)
            prev = prev->next;
        ListNode* const head2 = prev;

        prev = head2->next;
        ListNode *cur = prev->next;
        for (int i = m; i < n; ++i) {
            prev->next = cur->next;
            cur->next = head2->next;
            head2->next = cur;  // 头插法
            cur = prev->next;
        }

        return dummy.next;
    }
};
```

链表划分

- 给定一个链表和一个值x，将链表划分成两部分，使得划分后小于x的结点在前，大于等于x的结点在后。在这两部分中要保持原链表中的出现顺序。
  - 如：给定链表1->4->3->2->5->2和x = 3，返回1->2->2->4->3->5。

问题分析

- 分别申请两个指针p1和p2，小于x的添加到p1中，大于等于x的添加到p2中；最后，将p2链接到p1的末端即可。
- 时间复杂度是O(N)，空间复杂度为O(1)；该问题其实说明：快速排序对于单链表存储结构仍然适用。
  - 注：不是所有排序都方便使用链表存储，如堆排序，将不断的查找数组的n/2和n的位置，用链表做存储结构会不太方便。

Code

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode left_dummy(-1); // 头结点
        ListNode right_dummy(-1); // 头结点

        auto left_cur = &left_dummy;
        auto right_cur = &right_dummy;

        for (ListNode *cur = head; cur; cur = cur->next) {
            if (cur->val < x) {
                left_cur->next = cur;
                left_cur = cur;
            } else {
                right_cur->next = cur;
                right_cur = cur;
            }
        }

        left_cur->next = right_dummy.next;
        right_cur->next = nullptr;

        return left_dummy.next;
    }
};
```

排序链表中去重

- 给定排序的链表，删除重复元素，只保留重复元素第一次出现的结点。

问题分析

- 若p->next的值和p的值相等，则将p->next->next赋值给p，删除p->next；重复上述过程，直至链表尾端。

Code

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return nullptr;
        for (ListNode *prev = head, *cur = head->next; cur;) {
            if (prev->val == cur->val) {
                prev->next = cur->next;
                delete cur;
                cur = prev->next;//链表在前一步已经完成，该句是为了衔接for循环中的判断语句
            } else {
                prev = cur;
                cur = cur->next;
            }
        }
        return head;
    }
};
```

思考

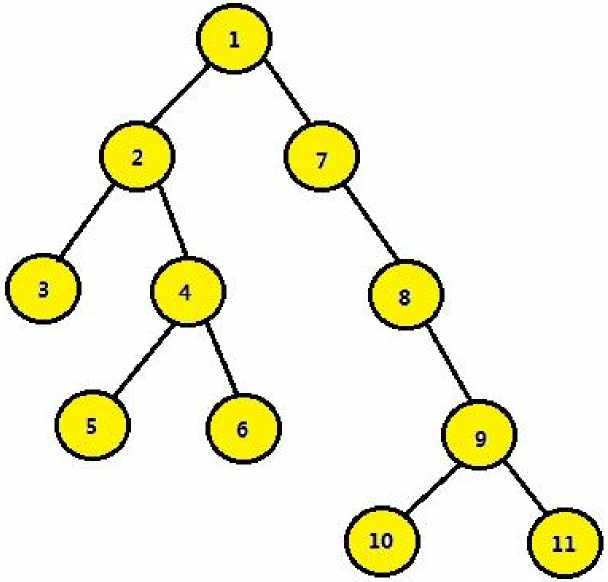
- 若题目变成：若发现重复元素，则重复元素全部删除，代码应该怎么实现呢？
  - 如：给定1->2->3->3->4->4->5, 返回1->2->5。

小结

- 可以发现，纯链表的题目，往往不难，但需要需要扎实的Coding基本功，在实现过程中，要特别小心next的指向，此外，删除结点时，一定要确保该结点不再需要。
- 小心分析引用类型的指针。

由LCA引出指针和递归问题

- 最近公共祖先(Lowest Common Ancestor, LCA)：给定一棵树 T 和两个结点 u 和 v，找出 u 和 v 离根结点最远的公共祖先。
- LCA(3,4)=2
- LCA(3,2)=2
- LCA(3,6)=4
- LCA(6, 10)=1



问题转化

- 在有父指针的前提下，该问题即为寻找两个**单向链表**的第一个公共结点。
- 两个单链表的第一个公共结点问题，下文不妨简称**单链公共结点**问题。

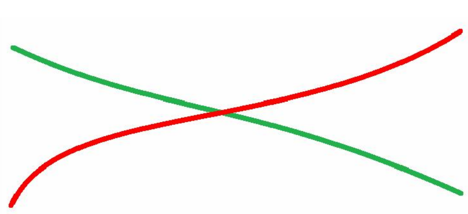
单链公共结点问题

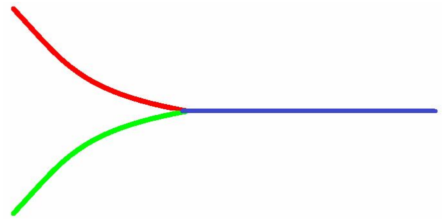
- 暴力求解：在第一个链表上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们的公共结点。如果第一个链表的长度为m，第二个链表的长度为n，显然，该方法的时间复杂度为O(mn)——即平方级时间复杂度。

单链公共结点问题

- 如果两个单向链表有公共的结点，也就是说两个链表从某一结点开始，它们的next指针都指向同一个结点。但由于是单向链表的结点，每个结点只有一个next指针，因此从第一个公共结点开始，之后它们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个Y，而不可能像X。

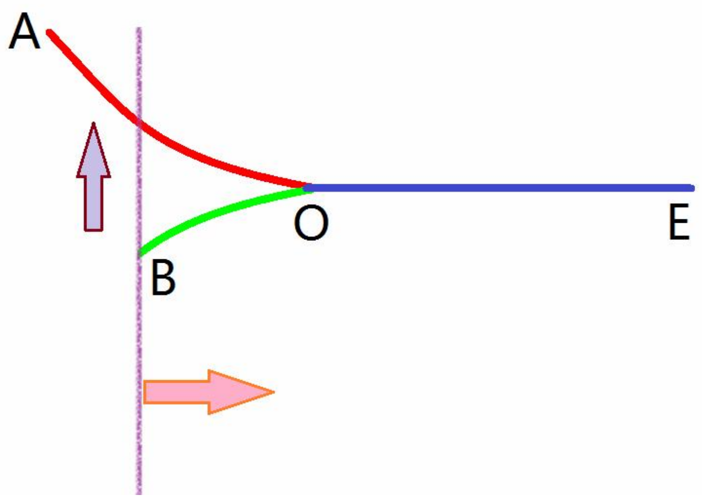
```
typedef struct tagListNode
{
  int m_nKey;
  tagListNode* m_pNext;
}ListNode;
```





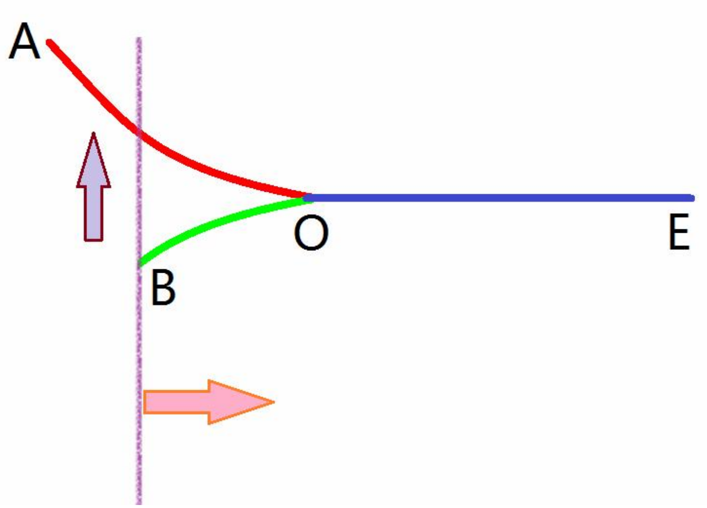
单链公共结点问题

- 分析：如果第一个链表的长度为m，第二个链表的长度为n，不妨认为m≥n，由于两个链表从第一个公共结点到链表的尾结点是完全重合的。所以**前面的(m-n)个结点一定没有公共结点**。



单链公共结点问题

- 算法：先分别遍历两个链表得到它们的长度m，n。在长的链表上先遍历|m-n|次后，再同步遍历两个链表，直到找到相同的结点，或者一直到链表结束。时间复杂度为O(m+n)。



思考

- 进一步的问题：如果两个链表可能有环，如何判断两个链表是否相交？以及找到两个链表的第一个公共点？
  - 快慢指针

一般LCA

- 在没有父指针的情况下，可以通过从根到v，u的递归查找，找到最近公共祖先。

Code

```
node* getLCA(node* root, node* node1, node* node2)
{
    if(root == null)
        return null;
    if(root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if(left != null && right != null)
        return root;
    else if(left != null)
        return left;
    else if (right != null)
        return right;
    else
        return null;
}
```

```
node* getLCA(node* root, node* node1, node* node2)
{
    if(root == null)
        return null;
    if(root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if(left != null && right != null)
        return root;
    else if(left != null)
        return left;
    else if (right != null)
        return right;
    else
        return null;
}
```

递归代码详解

```
if(root== node1 || root==node2)
    return root;
```

- 因为函数要返回node1和node2的最近祖先，但事实上，此处返回的，并不一定是“最正”的结论。
- 比如，node1==root，同时，node1不是node2的祖先，那么，“正”结论应该是null，但该代码返回的是node1

```
node* getLCA(node* root, node* node1, node* node2)
{
    if(root == null)
        return null;
    if(root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if(left != null && right != null)
        return root;
    else if(left != null)
        return left;
    else if (right != null)
        return right;
    else
        return null;
}
```

递归代码详解

node* left = getLCA(root->left, node1, node2);

node* right = getLCA(root->right, node1, node2);

- 第一句，会返回(node1,node2)的“潜在祖先”
  - ①如果node1，node2分立在root的左、右子树中，不妨认为node1在左、node2在右，返回的是node1；
  - ②如果node1，node2都在root的左子树中，将返回node1，node2的LCA
  - ③如果node1，node2都在root的右子树中，将返回null
- 第二句，返回的情况和第一句对称，不再赘述
- ②、③的情况，可以认为返回的是(node1,node2)的LCA，但①的情况，不是LCA。但仅仅暂时不是，没关系。

递归代码详解

```
node* getLCA(node* root, node* node1, node* node2)
{
    if(root == null)
        return null;
    if(root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if(left != null && right != null)
        return root;
    else if(left != null)
        return left;
    else if (right != null)
        return right;
    else
        return null;
}
```

```
if(left != null && right != null)
        return root;
```

- 如果发现left和right都不空，说明前面的第一、二句都返回了非空结果，那必然是node1在root的一侧，node2在另外一侧。root就是node1与node2的LCA。

递归代码详解

```
node* getLCA(node* root, node* node1, node* node2)
{
    if(root == null)
        return null;
    if(root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if(left != null && right != null)
        return root;
    else if(left != null)
        return left;
    else if (right != null)
        return right;
    else
        return null;
}
```

```
else if(left != null)
        return left;
else if (right != null)
        return right;
else
        return null;
```

- "最正"的情况：第一句的②情况：node1，node2都在root的左子树中，因此，只有left为非空(第二个else if情况对称，不再赘述)
- return null：就是node1和node2，两个都没有在树root中，显然应该返回null

括号匹配

- 给定字符串，仅由"()[]{}"六个字符组成。设计算法，判断该字符串是否有效。
  - 括号必须以正确的顺序配对，如："()"、"()[]"是有效的，但"([)]"无效。

算法分析

- 在考察第i位字符c与前面的括号是否匹配时：
- 如果c为左括号，开辟缓冲区记录下来，希望c能够与后面出现的同类型最近右括号匹配。
- 如果c为右括号，考察它能否与缓冲区中的左括号匹配。
  - 这个匹配过程，是检查缓冲区最后出现的同类型左括号
  - 即：后进先出——栈

算法流程

- 从前向后扫描字符串：
- 遇到左括号x，就压栈x；
- 遇到右括号y：
  - 如果发现栈顶元素x和该括号y匹配，则栈顶元素出栈，继续判断下一个字符。
  - 如果栈顶元素x和该括号y不匹配，字符串不匹配；
  - 如果栈为空，字符串不匹配；
- 扫描完成后，如果栈恰好为空，则字符串匹配，否则，字符串不匹配。

Code

```
class Solution {
public:
    bool isValid (string const& s) {
        string left = "([{";
        string right = ")]}";
        stack<char> stk;

        for (auto c : s) {
            if (left.find(c) != string::npos) {
                stk.push (c);
            } else {
                if (stk.empty () || stk.top () != left[right.find (c)])
                    return false;
                else
                    stk.pop ();
            }
        }
        return stk.empty();
    }
};
```

最长括号匹配

- 给定字符串，仅包含左括号'('和右括号')'，它可能不是括号匹配的，设计算法，找出最长匹配的括号子串，返回该子串的长度。
  如："(()"：2；"()(())"：6。

算法分析

- 考察第i位字符c：
- 如果c为左括号，在缓冲区中记录下来；
- 如果c为右括号，考察它能否与缓冲区中的左括号匹配。
  - 如果匹配，则计算两者之间的距离
  - 因为入栈的一定是左括号，显然没有必要将它们本身入栈，应该入栈的是该字符在字符串中的索引

Code

```
int GetLongestParenthese(const char* p)
{
    int size = (int)strlen(p);
    stack<int> s;
    int answer = 0;
    for(int i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            s.push(i);
        }
        else    //p[i] == ')'
        {
            if(!s.empty())
            {
                answer = max(answer, i - s.top() + 1);
                s.pop();
            }
        }
    }
    return answer;
}
```

Code2

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int max_len = 0, last = -1; // the position of the last ')'
        stack<int> lefts;  // keep track of the positions of non-matching '('s

        for (int i = 0; i < s.size(); ++i) {
            if (s[i] =='(') {
                lefts.push(i);
            } else {
                if (lefts.empty()) {
                    // no matching left
                    last = i;
                } else {
                    // find a matching pair
                    lefts.pop();
                    if (lefts.empty()) {
                        max_len = max(max_len, i-last);
                    } else {
                        max_len = max(max_len, i-lefts.top());
                    }
                }
            }
        }
        return max_len;
    }
};
```

观察与思考

- 经过分析算法得知，只有在右括号和左括号的发生匹配时，才有可能更新最终解；
- 做记录前缀串p[0…i-1]中左括号数目与右括号数目的差x，若x为0时，考察是否最终解得以更新即可。这个差x，其实是入栈的数目，代码中用“深度”deep表达；
- 由于可能出现左右括号不相等——尤其是左括号数目大于右括号数目，所以，再从右向前扫描一次。
- 这样完成的代码，用deep值替换了stack栈，空间复杂度由O(N)降到O(1)。

Code

```
int GetLongestParenthese2(const char* p)
{
    int size = (int)strlen(p);
    int answer = 0; //最终解
    int deep = 0;   //遇到了多少左括号
    int start = -1; //最深的(deep==0时)左括号的位置
                    //其实，为了方便计算长度，该变量是最深左括号的前一个位置
    int i;
    for(i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            deep++;
        }
        else    //p[i] == ')'
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, i - start);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    deep = 0;       //遇到了多少右括号
    start = size;   //最深的(deep==0时)右括号的位置
                    //其实，为了方便计算长度，该变量是最深右括号的后一个位置
    for(i = size-1; i >= 0; i--)
    {
        if(p[i] == ')')
        {
            deep++;
        }
        else    //p[i] == '('
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, start - i);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    return answer;
}
```

p="()(()))"

循环次数	深度	起始位置	当前解
<- 第一次循环 ->			
0	1	-1	0
1	0	-1	2
2	1	-1	2
3	2	-1	2
4	1	-1	2
5	0	-1	6
6	0	6	6
<- 第二次循环 ->			
6	1	7	6
5	2	7	6
4	3	7	6
3	2	7	6
2	1	7	6
1	2	7	6
0	1	7	6

p="((()()))"

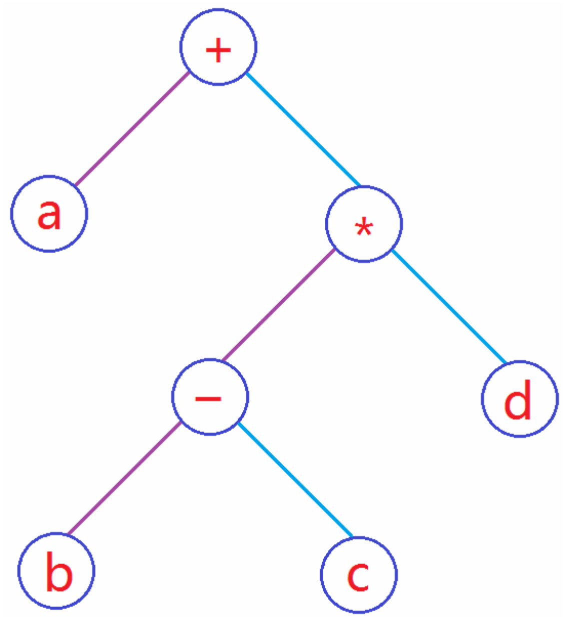
循环次数	深度	起始位置	当前解
<- 第一次循环 ->			
0	1	-1	0
1	2	-1	0
2	3	-1	0
3	4	-1	0
4	3	-1	0
5	4	-1	0
6	3	-1	0
7	2	-1	0
8	1	-1	0
<- 第二次循环 ->			
8	1	9	0
7	2	9	0
6	3	9	0
5	2	9	0
4	3	9	0
3	2	9	0
2	1	9	0
1	0	9	8
0	0	0	8

逆波兰表达式RPN

- 逆波兰表达式Reverse Polish Notation，又叫后缀表达式。
- 习惯上，二元运算符总是置于与之相关的两个运算对象之间，即中缀表达方法。波兰逻辑学家J.Lukasiewicz于1929年提出了运算符都置于其运算对象之后，故称为后缀表示。
- 如：
  - 中缀表达式：a+(b-c)*d
  - 后缀表达式：abc-d*+

运算与二叉树

- 事实上，二元运算的前提下，中缀表达式可以对应一颗二叉树；逆波兰表达式即该二叉树后序遍历的结果。
- 中缀表达式：a+(b-c)*d
- 后缀表达式：abc-d*+
  - 该结论对多元运算也成立，如“非运算”等



计算逆波兰表达式

- 计算给定的逆波兰表达式的值。有效操作只有+-*/，每个操作数都是整数。
- 如：
  - “2”, “1”, “+”, “3”, “*”：9——(2+1) * 3
  - "4", "13", "5", "/", "+"：6——4+(13/5)

逆波兰表达式的计算方法

- abc-d*+
- 若当前字符是操作数，则压栈
- 若当前字符是操作符，则弹出栈中的两个操作数，计算后仍然压入栈中
  - 若某次操作，栈内无法弹出两个操作数，则表达式有误。

Code

```
class Solution {
public:
    int evalRPN(vector<string> &tokens) {
        stack<string> s;
        for (auto token : tokens) {
            if (!is_operator(token)) {
                s.push(token);
            } else {
                int y = stoi(s.top());
                s.pop();
                int x = stoi(s.top());
                s.pop();
                if (token[0] == '+')        x += y;
                else if (token[0] == '-')   x -= y;
                else if (token[0] == '*')   x *= y;
                else                        x /= y;
                    s.push(to_string(x));
            }
        }
        return stoi(s.top());
    }
private:
    bool is_operator(const string &op) {
        return op.size() == 1
        && string("+-*/").find(op) != string::npos;
    }
};
```

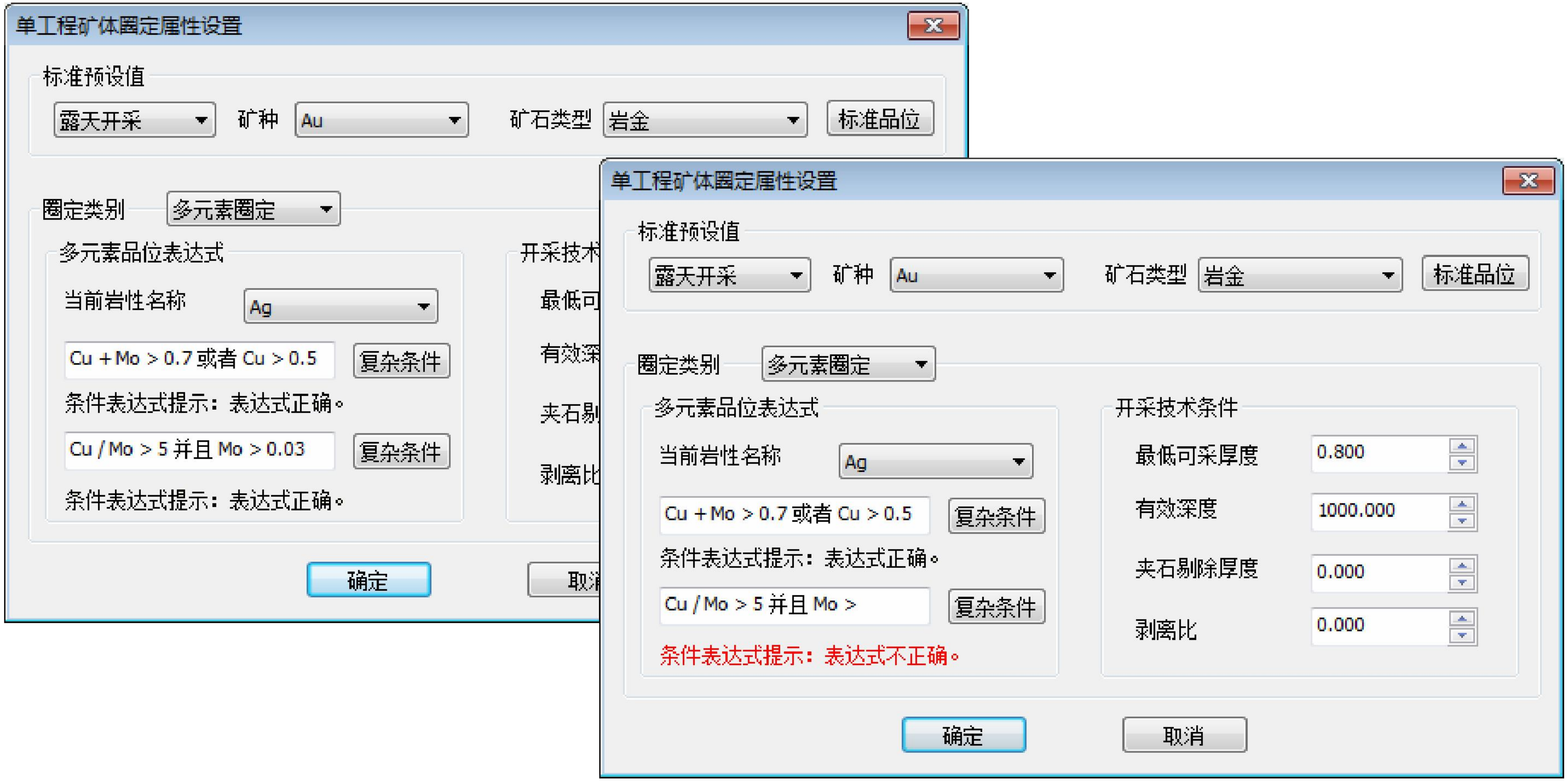
逆波兰表达式

- 计算数学表达式的最常用方法；
- 在实践中，往往给出的不是立即数，而是变量名称；若经常计算且表达式本身不变，可以事先将中缀表达式转换成逆波兰表达式存储。

计算器

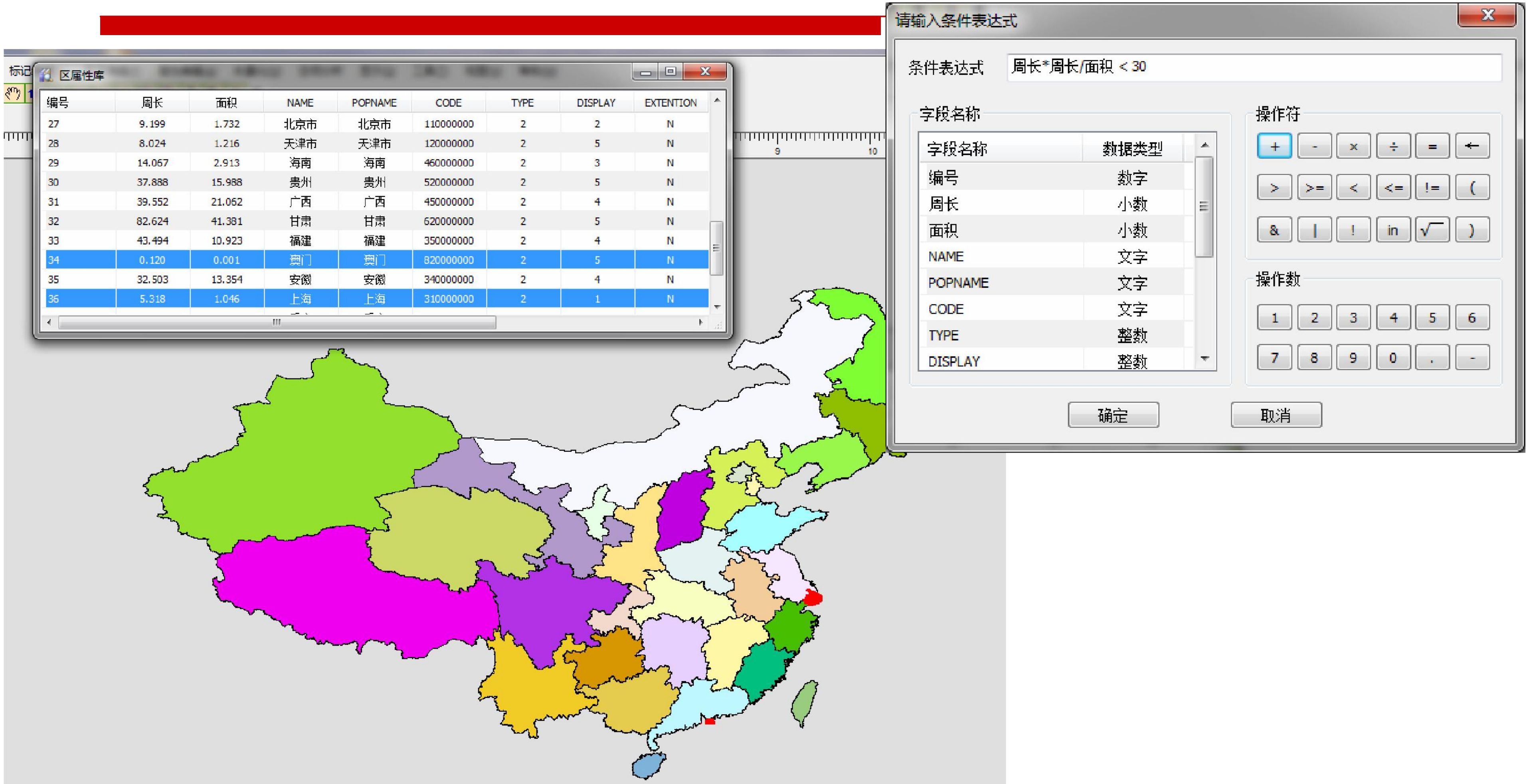
- 将中缀表达式转换成逆波兰表达式，然后正常计算。

逆波兰表达式的用途



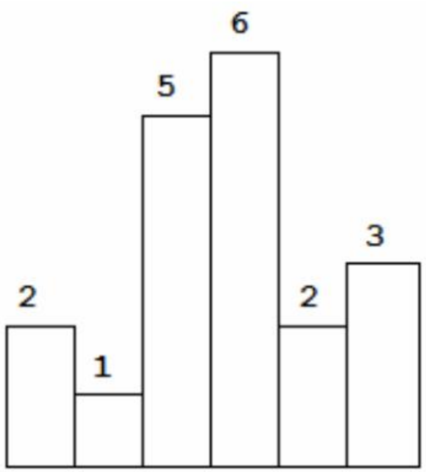
单工程矿体圈定属性设置
标准预设值
露天开采
矿种
Au
矿石类型
岩金
标准品位
圈定类别
多元素圈定
多元素品位表达式
当前岩性名称
Ag
Cu + Mo > 0.7 或者 Cu > 0.5
复杂条件
条件表达式提示：表达式正确。
Cu / Mo > 5 并且 Mo > 0.03
复杂条件
条件表达式提示：表达式正确。
确定
单工程矿体圈定属性设置
标准预设值
露天开采
矿种
Au
矿石类型
岩金
标准品位
圈定类别
多元素圈定
多元素品位表达式
当前岩性名称
Ag
Cu + Mo > 0.7 或者 Cu > 0.5
复杂条件
条件表达式提示：表达式正确。
Cu / Mo > 5 并且 Mo >
复杂条件
条件表达式提示：表达式不正确。
开采技术条件
最低可采厚度
0.800
有效深度
1000.000
夹石剔除厚度
0.000
剥离比
0.000
确定
取消

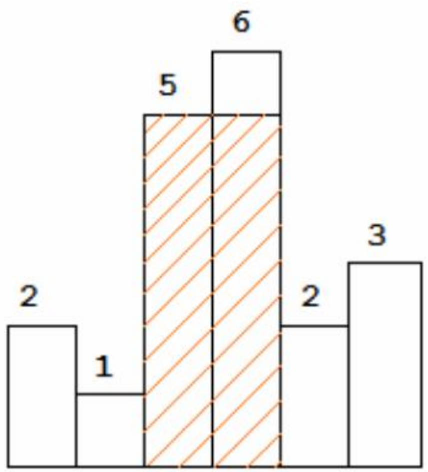
省级行政区中哪几个最接近圆形？



直方图矩形面积

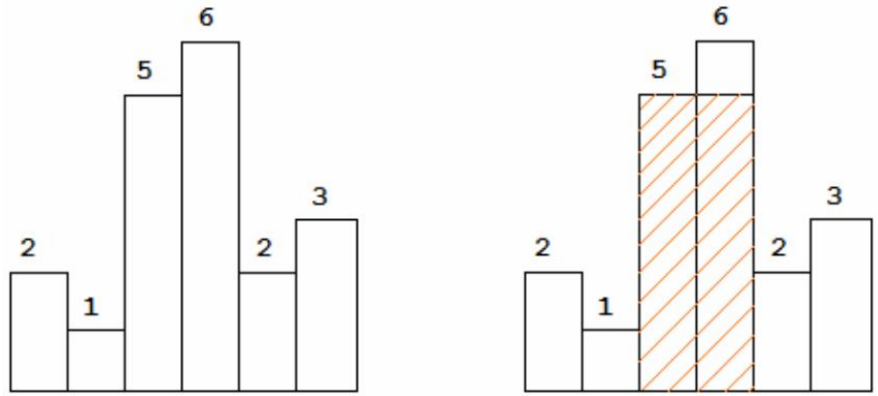
- 给定n个非负整数，表示直方图的方柱的高度，同时，每个方柱的宽度假定都为1；试找出直方图中最大的矩形面积。如：给定高度为：2,1,5,6,2,3，最大面积为10。





暴力求解

- 将直方图的数组记做a[0…size-1]；
- 计算以方柱a[i]为右边界的直方图中，遍历a[0…i]，依次计算可能的高度和面积，取最大者；
- i从0遍历到size-1；
- 时间复杂度为$O(N^2)$。



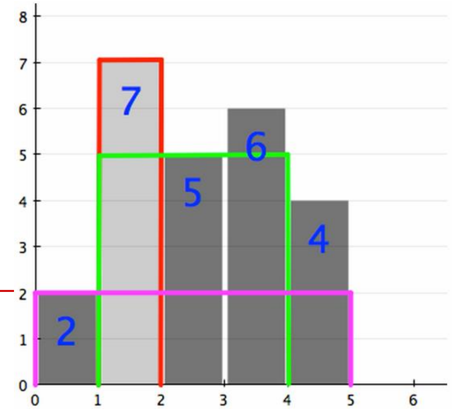
分析

- 显然，若a[i+1]≥a[i]，则以a[i]为右边界的矩形Rect(width,height)，总可以添加a[i+1]带来的矩形Rect(1,height)，使得面积增大
- 只有当a[i+1]＜a[i]时，才计算a[i]为右边界的矩形面积。
  - trick：为了算法一致性，在a[0…size-1]的最后，添加a[size]=0，保证a[size-1]为右边界的矩形得到计算。

算法思想

- 从前向后遍历a[0…size](末尾添加了0)，若a[i]>a[i-1]，则将a[i]放入缓冲区；
- 若a[i]≤a[i-1]，则计算缓冲区中能够得到的最大矩形面积。
- 从a[i]>a[i-1]可以得出：
  - 缓冲区中放入的值是递增的
  - 每次只从缓冲区取出最后元素和a[i]比较——栈。

算法分析



- 以2、7、5、6、4为例：
- 假设当前待分析的元素是4，由刚才的分析得知，栈内元素是2,5,6，其中，6是栈顶。
  - 此时，栈顶元素6＞4，则6出栈
    - 出栈后，新的栈顶元素为5，5和4的横向距离差为1：以6为高度，1为宽度的矩形面积是6*1=6
  - 此时，栈顶元素5＞4，则5出栈
    - 出栈后，新的栈顶元素为2，2和4的横向距离差为3：以5为高度，3为宽度的矩形面积是5*3=15
  - 此时，栈顶元素2≤4，则i++，继续遍历直方图后面的值

说明

- 显然，为了能够方便的计算“横向距离”，压入栈的是方柱的索引，而非方柱的高度本身。
  - 这种trick在实践中经常使用。
- 因为每个方柱最多只计算一次——只有压栈的方柱才会计算面积，并且每次计算一次即可完成。所以，它的时间复杂度是O(N) 。

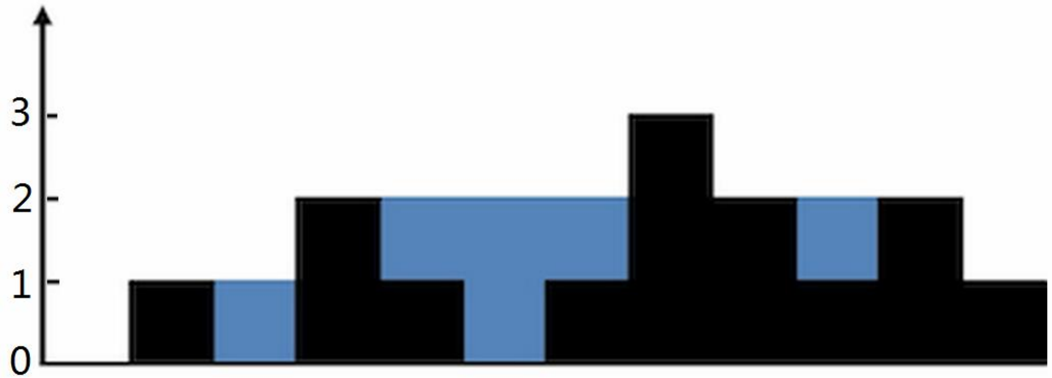
Code

```
int LargestRectangleArea(vector<int>& height)
{
    height.push_back(0);    //确保原数组height的最后一位能够得到计算

    stack<int> s;
    int answer = 0;
    int temp;   //临时变量
    for (int i = 0; i < (int)height.size(); )
    {
        if (s.empty() || height[i] > height[s.top()])
        {
            s.push(i);
            i++;
        }
        else
        {
            temp = s.top();
            s.pop();
            answer = max(answer,height[temp]*(s.empty() ? i : i-s.top()-1));
        }
    }
    return answer;
}
```

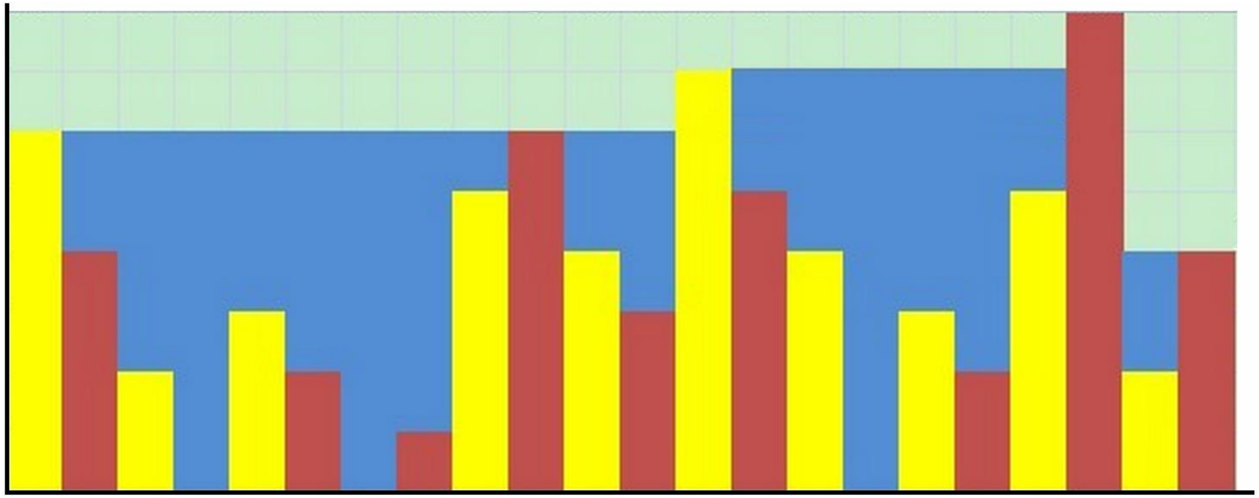
另一个直方图例题：收集雨水问题

- 给定n个非负整数，表示直方图的方柱的高度，同时，每个方柱的宽度假定都为1。若使用这样形状的容器收集雨水，可以盛多少水量？
  - 如输入：0,1,0,2,1,0,1,3,2,1,2,1；返回6。

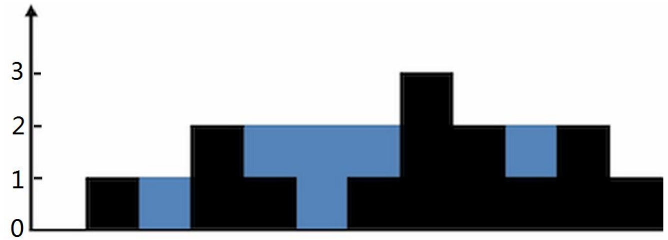


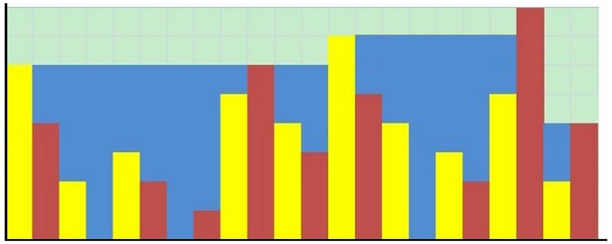
算法分析

- 计算所有局部最低点，然后每个“洼坑”分别计算，这种贪心的策略是不对的，因为，有可能局部最高点被覆盖：
  - 如：6,3,2,0,3,2,0,1,5,6,4,3,7,5,4,0,3,2,5,8,2,4



思路分析





- 记最终盛水量为trap，初值为0；
- 考察直方图最左边L和最右边R的两个方柱：
  - 它们两个本身，一定不可能存储雨水：因为在最边界；
  - 记它们比较低的那个为X，与X相邻的方柱记做Y。
    - 若Y≥X，可将X丢弃，且trap值不变；
    - 若Y＜X，则X-Y即为Y方柱最多盛水量；仍然丢弃X，且trap+=(X-Y)。
    - 无论如何，L或者R都将向中间靠近一步，重复上述过程，直至L==R。

Code

```
int TrappingRainWater(int A[], int n)
{
    int secHight = 0;    //当前找到的第二大的数
    int left = 0;
    int right = n-1;
    int trap = 0;    //依次遍历每个方柱能装水的容量
    while (left < right)
    {
        if (A[left] < A[right])
        {
            secHight = max(A[left], secHight);
            trap += (secHight-A[left]);
            left++;
        }
        else
        {
            secHight = max(A[right], secHight);
            trap += (secHight-A[right]);
            right--;
        }
    }
    return trap;
}
```

小结

- 栈的用途非常广泛，除了表达式求值，在深度优先遍历、保存现场等问题中常常出现。
- 关于栈的话题不限于此。
- 思考：一个栈(无穷大)的进栈序列为1,2,3,..n，共多少种不同的出栈序列?
  - 该问题将在动态规划中继续讨论。

参考文献

- 戴方勤,LeetCode题解,2014
- http://blog.163.com/clevertanglei900@126/blog/static/111352259201191414846 7/(单链公共结点问题)
- http://m.blog.csdn.net/blog/zh_qd1014/6879083(LCA&RMQ)
- http://www.cnblogs.com/avril/archive/2013/08/24/3278873.html(直方图矩形面积)

我们在这里

- 更多算法面试题在 7 七月算法
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @研究者July
  - @七月问答
  - @邹博_机器学习

感谢大家

恳请大家批评指正！