算法讨论班第二十一期------2016 年 1 月 22 日

主讲人：王海璐

141. Linked List Cycle

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

Follow up: Can you solve it without using extra space?

142. Linked List Cycle II

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

Note: Do not modify the linked list.

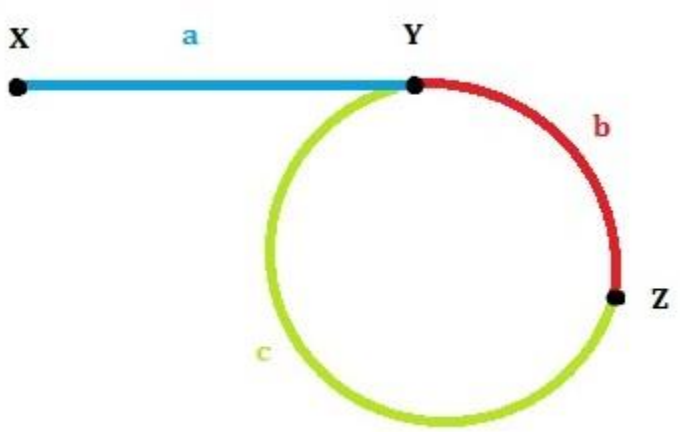
Follow up: Can you solve it without using extra space?

1．对于判断链表是否有环，用两个指针 slow 和 fast，一开始都指向头结点， fast 指针一次走两步，slow 指针一次只走一步，当两个指针重合时表示存在环了。

证明：假设链表有环，环的长度为 l，环内指针位置从 0 开始计数，fast 指针比 slow 指针先进入环，假设当 slow 慢指针在环的起始位置时，fast 快指针在位置 k，那么 fast 指针只要比 slow 指针多走经过 l-k 步，就可以追上慢指针了。又因为每一次 fast 指针都比 slow 指针多走一步，所以一定可以在 l-k 步（从进入环追上的步数）后追上慢指针。综上，那么从链表头结点开始，只要链表中有环，fast 指针一定会在有限步追上 slow 指针。

2．求链表环起始位置的算法思路：当快指针和慢指针重合的时候，把一个指针重新指向链表的头指针，两个指针一次走一步，那么当两个指针值相同时，所在的指针就是我们要找的起始位置。

证明：



上图中假设：链表头结点是 X，环的第一个结点是 Y，slow 和 fast 第一次的结点是 Z。各段的长度分别是 a,b,c，那么，环的长度是 l=c+b。

由上一证明可以知道 slow 第一次到达 Y 的时候，需要 l-k 步（不到一圈）就可以达到 Z 点，Z 也就是环内的第 l-k 位置（Y 从 0 开始计数）。所以相遇时 slow 走过的距离（步数）是 a+b；而 fast 在环内多走了数圈 (假设为 *x*)，那么 fast 走过的步数为 a+b+x*l。又因为 fast 的速度是 slow 的两倍，所以 fast 走的距离是 slow 的两倍，有 2(a+b) =a+b+x*l，可以得到 a+b=x*l；进而 a+b=x*(b+c);就可得到 a=(x-1)*l+c。可以从式子看出，从链表头结点 X 开始走 a 步的结点，与从环内位置 Z 开始走 c 步并加上绕 x-1 圈，两个指针可以在环的开始位置 Y 相遇。

证毕。

3 代码如下：

代码需注意：判断 **fast 指针为空 NULL 的时候就不能进入循环**，即最后两个指针都为空的时候返回 false。两个指针重合并且不为空，返回 true。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head){
        ListNode *slow = head;
        ListNode *fast = head;
        while (fast!=NULL && fast->next!=NULL)
        {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast)
            {
                return true;
            }
        }
        return false;
    }
};
```

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head){
        ListNode *slow = head;
        ListNode *fast = head;
        ListNode *begin = NULL;
        while (fast != NULL && fast->next != NULL)
        {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast)
            {
                slow = head;
                begin = fast;
                break;
            }
        }
        while (begin!= NULL && begin != slow)
        {
            slow = slow->next;
            begin = begin->next;
        }
        return begin;
    }
};
```

188. Best Time to Buy and Sell Stock IV

Say you have an array for which the i^{th} element is the price of a given stock on day *i*.

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

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

题意：用一个数组表示股票每天的价格，数组的第 i 个数表示股票在第 i 天的价格。最多交易 k 次，手上最多只能持有一支股票，求最大收益。

思路分析：动态规划法。

传统的动态规划我们会这样想，到第 i 天时进行 j 次交易的最大收益，要么等于到第 i-1 天时进行 j 次交易的最大收益（第 i 天价格低于第 i-1 天的价格），要么等于到第 i-1 天时进行 j-1 次交易，然后第 i 天进行一次交易（第 i 天价格高于第 i-1 天价格时）。于是得到动态规划方程如下（其中 diff = prices[i] – prices[i – 1]）：

profit[i][j] = max(profit[i – 1][j], profit[i – 1][j – 1] + diff)

看起来很有道理，但其实不对，为什么不对呢？因为 diff 是第 i 天和第 i-1 天的差额收益，如果第 i-1 天当天本身也有交易呢（也就是说第 i-1 天刚卖出了股票，然后又买入等到第 i 天再卖出），那么这两次交易就可以合为一次交易，这样 profit[i – 1][j – 1] + diff 实际上只进行了 j-1 次交易，而不是最多可以的 j 次，这样得到的最大收益就小了。

特殊动态规划法。

那么怎样计算第 i 天进行交易情况的最大收益，才会避免少计算一次交易呢？我们用一个局部最优解和全局最优解表示到第 i 天进行 j 次的收益，这就是该动态规划的特殊之处。

用 local[i][j]表示到达第 i 天时，最多进行 j 次交易的局部最优解；用 global[i][j]表示到达第 i 天时，最多进行 j 次的全局最优解。之所以在 global 之外还要维护一个 local 数组，是因为在计算 global[i][j]时，面临两种情况：最后一天不做交易，那么直接等于 global[i-1][j]；最后一天要做交易，那么又需要分别考虑最后一天是否有收益的问题，所以，它们二者的关系如下（其中 diff = prices[i] – prices[i – 1]）：

local[i][j] = max(global[i – 1][j – 1], local[i – 1][j] + diff)
global[i][j] = max(global[i – 1][j], local[i][j])

local[i][j]和 global[i][j]的区别是：local[i][j]意味着在第 i 天一定有交易（卖出）发生，当第 i 天的价格高于第 i-1 天（即 diff > 0）时，那么可以把这次交易（第 i-1 天买入第 i 天卖出）跟第 i-1 天的交易（卖出）合并为一次交易，即 local[i][j]=local[i-1][j]+diff；当第 i 天的价格不高于第 i-1 天（即 diff<=0）时，由于交易亏损，所以 local[i][j]=global[i-1][j-1]。global[i][j]就是我们所求的前 i 天最多进行 k 次交易的最大收益，可分为两种情况：如果第 i 天没有交易（卖出），那么 global[i][j]=global[i-1][j]；如果第 i 天有交易（卖出），那么 global[i][j]=local[i][j]。

代码时间复杂度 O(nk)，空间复杂度 O(nk)：

当 k(最多交易次数)>n(交易天数)时，相当于不限交易次数（Best Time to Buy and Sell Stock II）

```cpp
class Solution {
public:
    int maxProfit(int k, vector<int>& prices) {
        if (prices.size()<2)
        {
            return 0;
        }
        if (k >= prices.size())
            return maxProfit2(prices);
        int n = prices.size();
        vector<vector<int>> local(n,vector<int>(k+1,0));
        vector<vector<int>> global(n, vector<int>(k + 1, 0));
        for (int i = 1; i < n; i++)
        {
            int diff = prices[i] - prices[i - 1];
            for (int j = 1; j <=k; j++)
            {
                local[i][j] = max(global[i][j - 1], local[i - 1][j] + diff);
                global[i][j] = max(global[i - 1][j], local[i][j]);
            }
        }
        return global[n-1][k];
    }
    int maxProfit2(vector<int>& prices) {
        if (prices.empty())
        {
            return 0;
        }
        if (prices.size() < 2)
        {
            return 0;
        }
        int maxProfit=0;
        for (int i = 1; i < prices.size(); i++)
        {
            int diff = prices[i] - prices[i - 1];
            if (diff>0)
            {
                maxProfit += diff;
            }
        }
        return maxProfit;
    }
};
```

下期题目：80,82,83