LeetCode 热题 HOT 100 （3）

139. 单词拆分

(动态规划) $O(n^3)$

思路

状态表示： `f[i]`表示字符串`s`的前`i`个字符是否可以拆分成`wordDict`，其值有两个`true`和`false`。

状态计算： 依据最后一次拆分成的字符串`str`划分集合，最后一次拆分成的字符串`str`可以为`s[0 ~ i - 1]`，`s[1 ~ i - 1]`，，，`s[j ~ i - 1]`。

状态转移方程： `f[i] = ture` 的条件是：`f[j] = ture`并且`s[j, i - 1]`在`hash`表中存在。

初始化： `f[0] = true`，表示空串且合法。

实现细节：

为了快速判断字符串`s`拆分出来的子串在`wordDict`中出现，我们可以用一个哈希表存贮`wordDict`中的每个`word`。

时间复杂度分析： 状态枚举$O(n^2)$，状态计算$O(n)$，因此时间复杂度为$O(n^3)$。

c++代码

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> hash;   //存贮单词
        vector<bool> f(s.size() + 1, false);
        f[0] = true;  //初始化
        for(string word : wordDict){
            hash.insert(word);
        }
        for(int i = 1; i <= s.size(); i++){
            for(int j = 0; j < i; j++){
                if(f[j] && hash.find(s.substr(j, i - j)) != hash.end()){
                    f[i] = true;
                    break;  //只要有一个子集满足就ok了
                }
            }
        }
        return f[s.size()];
    }
};
```

141. 环形链表

思路

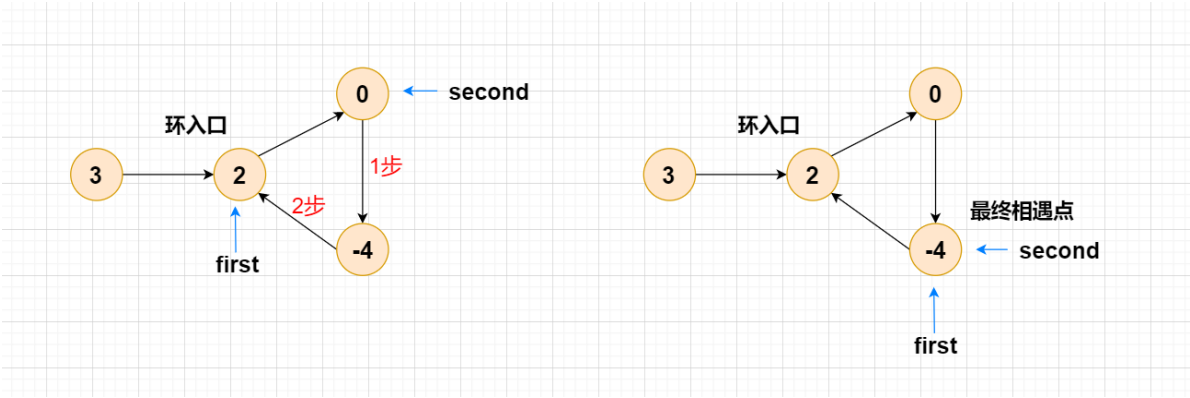
(链表，指针扫描) $O(n)$

用两个指针从头开始扫描，第一个指针每次走一步，第二个指针每次走两步。如果走到`null`，说明不存在环；否则如果两个指针相遇，则说明存在环。

为什么呢?

假设链表存在环，则当第一个指针走到环入口时，第二个指针已经走到环上的某个位置，距离环入口还差 `x` 步。由于第二个指针每次比第一个指针多走一步，所以第一个指针再走 `x` 步，两个指针就相遇了。

如下图所示:



第二个指针还差2步就可以到达环入口，但是第二个指针每次比第一个指针多走1步，因此第一个指针再走2步，两个指针就会相遇。

时间复杂度分析:

第一个指针在环上走不到一圈，所以第一个指针走的总步数小于链表总长度。而第二个指针走的路程是第一个指针的两倍，所以总时间复杂度是$O(n)$。

c++代码

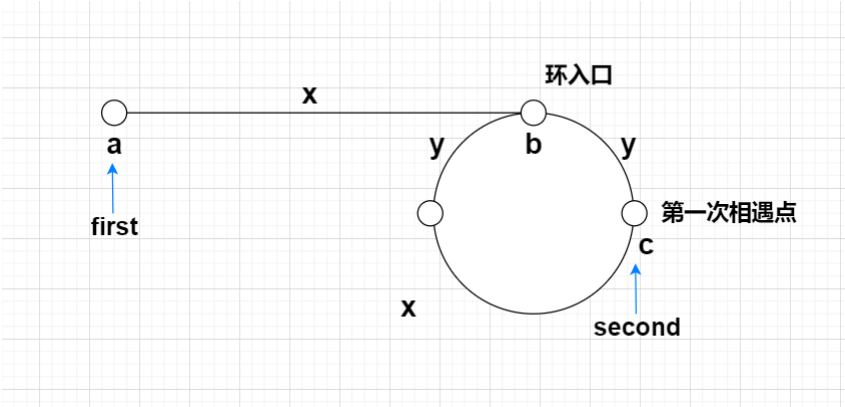
```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        // 只有零个或者1个节点，必然不会成环
        if(!head || !head->next) return false;
        ListNode *first = head;
        ListNode *second = head;
        while(second){
            first = first->next, second = second->next;
            if(!second) return false;
            second = second->next;
            if(second == first) return true;
        }
        return false;
    }
};
```

142. 环形链表 II

思路

(链表，快慢指针) $O(n)$

本题的做法比较巧妙。
用两个指针 $first, second$分别从起点开始走，$first$每次走一步，$second$每次走两步。
如果过程中 $second$ 走到`null`，则说明不存在环。否则当 $first$和 $second$相遇后，让 $first$ 返回起点，$second$ 待在原地不动，然后两个指针每次分别走一步，当相遇时，相遇点就是环的入口。



证明：如上图所示，a 是起点，b 是环的入口，c 是两个指针的第一次相遇点，ab之间的距离是x，bc之间的距离是 y。
则当 $first$ 走到 b 时，由于$second$比 $first$多走一倍的路，所以$second$已经从 b 开始在环上走了 x 步，可能多余1圈，距离 b 还差 y 步（这是因为第一次相遇点在 b 之后 y 步，我们让 first 退回 b 点，则 $second$ 会退 2y 步，也就是距离 b 点还差 y 步）；所以 $second$从 b 点走 x+y 步即可回到 b 点，所以 $second$ 从 c 点开始走，走 x 步即可恰好走到b 点，同时让 $first$ 从头开始走，走 x 步也恰好可以走到 b 点。所以第二次相遇点就是 b 点。

时间复杂度分析：
$first$ 总共走了$2x + y$ 步，$second$ 总共走了 $2x + 2y + x$步，所以两个指针总共走了 $5x + 3y$步。由于当第一次$first$ 走到 b 点时，$second$ 最多追一圈即可追上$first$，所以 y 小于环的长度，所以 $x + y$小于等于链表总长度。所以总时间复杂度是$O(n)$。

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if(!head || !head->next) return NULL;
        ListNode *first = head;
        ListNode *second = head;
        while(second){
            first = first->next, second = second->next;
            if(!second) return NULL;
            second = second->next;
            if(second == first){
```

```
                first = head;
                while(first != second){
                    first = first->next;
                    second = second->next;
                }
                return first;
            }
        }
        return NULL;
    }
};
```

146. LRU 缓存

题意解释

请为**LRU缓存**设计一个数据结构。支持两种操作：`get`和`set`。

- `get(key)`：如果`key`在缓存中，则返回`key`对应的值（保证是正的）；否则返回`-1`；
- `set(key, value)`：如果`key`在缓存中，则更新`key`对应的值；否则插入`(key, value)`，如果缓存已满，则先删除上次使用时间最老的`key`。

思路

(双链表+哈希) $O(1)$

使用一个双链表和一个哈希表：

- 双链表存储一个节点被使用（get或者put）的时间戳，且按最近使用时间从左到右排好序，最先被使用的节点放在双链表的第一位，因此双链表的最后一位就是最久未被使用的节点；
- 哈希表存储`key`对应的链表中的节点地址,用于`key-value`的增删改查；

初始化：

- `n`是缓存大小；
- 双链表和哈希表都为空；

`get(key)`： 首先用哈希表判断key是否存在：

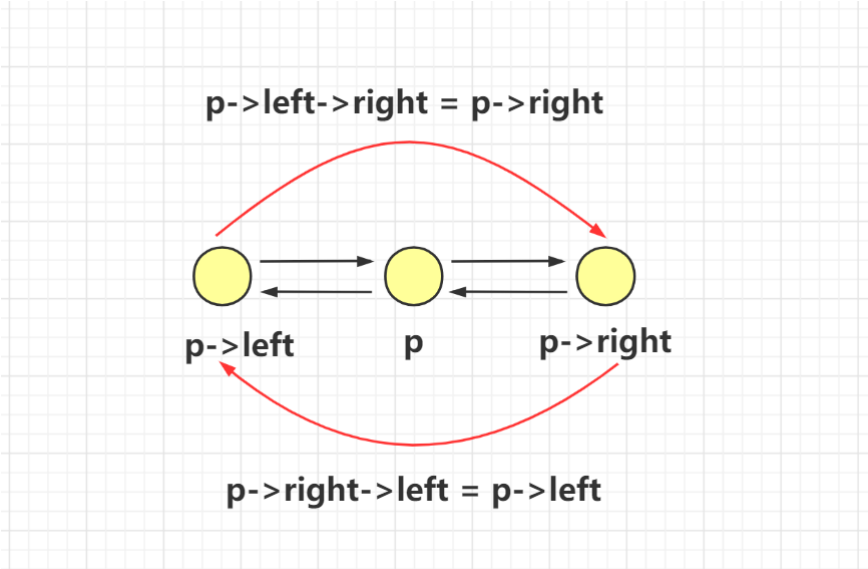
- 如果key不存在，则返回-1；
- 如果key存在，则返回对应的value，同时将key对应的节点放到双链表的最左侧；

`put(key, value)`： 首先用哈希表判断key是否存在：

- 如果key存在，则修改对应的value，同时将key对应的节点放到双链表的最左侧；
- 如果key不存在：
    - 如果缓存已满，则删除双链表最右侧的节点（上次使用时间最老的节点），更新哈希表；
    - 否则，插入(key, value)：同时将key对应的节点放到双链表的最左侧，更新哈希表；

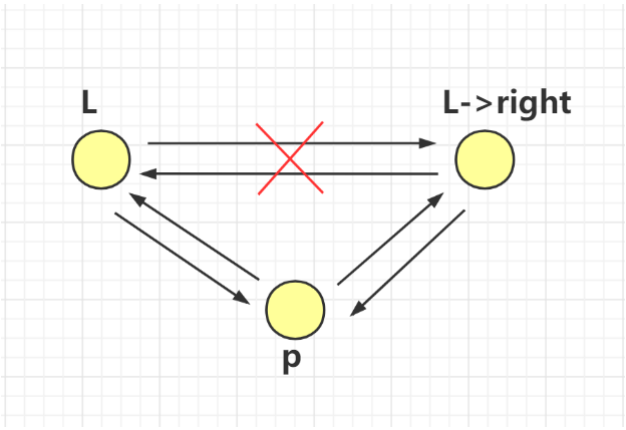
对应的双链表的几种操作

1、删除p节点




```
p->right->left = p->left;
p->left->right = p->right;
```

2、在L节点之后插入p节点




```
p->right = L->right;
p->left = L;
L->right->left = p;
L->right = p;
```

时间复杂度分析：双链表和哈希表的增删改查操作的时间复杂度都是 $O(1)$，所以get和set操作的时间复杂度也都是 $O(1)$。

c++代码

```
class LRUCache {
public:

    //定义双链表
    struct Node{
        int key,value;
        Node* left ,*right;
        Node(int _key,int _value):
key(_key),value(_value),left(NULL),right(NULL){}
    }*L,*R;//双链表的最左和最右节点，不存贮值。
    int n;
    unordered_map<int,Node*>hash;
```

```
12
13      void remove(Node* p)
14      {
15          p->right->left = p->left;
16          p->left->right = p->right;
17      }
18      void insert(Node *p)
19      {
20          p->right = L->right;
21          p->left = L;
22          L->right->left = p;
23          L->right = p;
24      }
25      LRUCache(int capacity) {
26          n = capacity;
27          L = new Node(-1,-1),R = new Node(-1,-1);
28          L->right = R;
29          R->left = L;
30      }
31
32      int get(int key) {
33          if(hash.count(key) == 0) return -1; //不存在关键字 key
34          auto p = hash[key];
35          remove(p);
36          insert(p);//将当前节点放在双链表的第一位
37          return p->value;
38      }
39
40      void put(int key, int value) {
41          if(hash.count(key)) //如果key存在，则修改对应的value
42          {
43              auto p = hash[key];
44              p->value = value;
45              remove(p);
46              insert(p);
47          }
48          else
49          {
50              if(hash.size() == n) //如果缓存已满，则删除双链表最右侧的节点
51              {
52                  auto  p = R->left;
53                  remove(p);
54                  hash.erase(p->key); //更新哈希表
55                  delete p; //释放内存
56              }
57              //否则，插入(key, value)
58              auto p = new Node(key,value);
59              hash[key] = p;
60              insert(p);
61          }
62      }
63  };
```

148. 排序链表

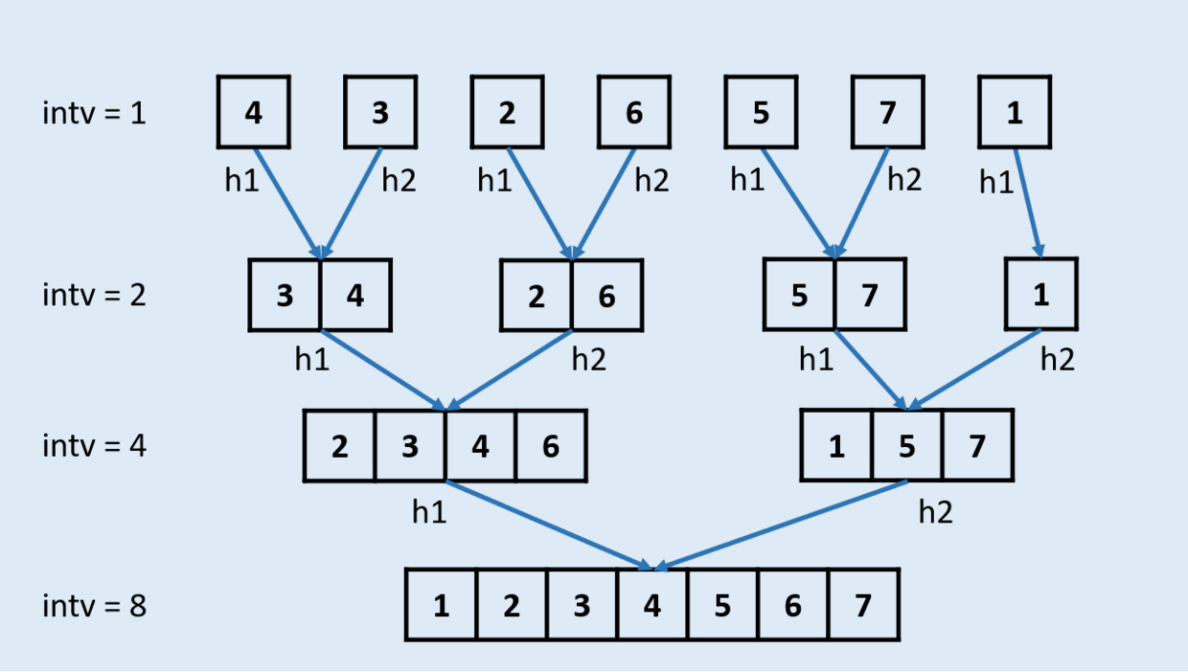
思路

(归并排序) $O(nlogn)$

自顶向下递归形式的归并排序，由于递归需要使用系统栈，递归的最大深度是 $logn$，所以需要额外 $O(logn)$ 的空间。
所以我们需要使用自底向上非递归形式的归并排序算法。
基本思路是这样的，总共迭代 $logn$次：

1. 第一次，将整个区间分成连续的若干段，每段长度是2：
   $[a0, a1], [a2, a3], \dots [an-1, an-1][a0, a1]$,然后将每一段内排好序，小数在前，大数在后；
2. 第二次，将整个区间分成连续的若干段，每段长度是4：
   $[a0, \dots, a3], [a4, \dots, a7], \dots [an-4, \dots, an-1][a0, \dots, a3]$,然后将每一段内排好序，这次排序可以利用之前的结果，相当于将左右两个有序的半区间合并，可以通过一次线性扫描来完成；
3. 依此类推，直到每段小区间的长度大于等于 n 为止；

另外，当 n 不是2的整次幂时，每次迭代只有最后一个区间会比较特殊，长度会小一些，遍历到指针为空时需要提前结束。



举个例子：

根据图片可知，从底部往上逐渐进行排序，先将长度是`1`的链表进行两两排序合并，再形成新的链表`head`，再在新的链表的基础上将长度是`2`的链表进行两两排序合并，再形成新的链表`head` ... 直到将长度是`n / 2`的链表进行两两排序合并

```
step=1: (3->4) -> (1->7) -> (8->9) -> (2->11) -> (5->6)
step=2: (1->3->4->7) -> (2->8->9->11) -> (5->6)
step=4: (1->2->3->4->7->8->9->11) ->5->6
step=8: (1->2->3->4->5->6->7->8->9->11)
```

具体操作，当将长度是`i`的链表两两排序合并时，新建一个虚拟头结点`dummy`，`[j,j + i - 1]`和`[j + i, j + 2 * i - 1]`两个链表进行合并，在当前组中，`p`指向的是当前合并的左边的链表，`q`指向的是当前合并的右边的链表，`o`指向的是下一组的开始位置，将左链表和右链表进行合并，加入到`dummy`的链表中，操作完所有组后，返回`dummy.next`链表给`i * 2`的长度处理

注意的是：需要通过`l`和`r`记录当前组左链表和右链表使用了多少个元素，用的个数不能超过`i`，即使长度不是`2n`也可以同样的操作

时间复杂度分析：

整个链表总共遍历 $logn$ 次，每次遍历的复杂度是 $O(n)$，所以总时间复杂度是$O(nlogn)$。

空间复杂度分析：

整个算法没有递归，迭代时只会使用常数个额外变量，所以额外空间复杂度是 $O(1)$.

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        int n = 0;
        for(auto p = head; p; p = p->next) n++;
        auto dummy = new ListNode(-1); //虚拟头节点
        dummy->next = head;
        //每次归并段的长度，每次长度依次为1,2,4,8...n/2, 小于n是因为等于n时说明所有元素均归并完毕，大于n时同理
        for(int i = 1; i < n; i *= 2)
        {
            auto cur = dummy ;
            for(int j = 1; j + i <= n; j += 2*i ){ //j代表每一段的开始，每次将两段有序段归并为一个大的有序段，故而每次+2i            //必须保证每段中间序号是小于等于链表长度的，显然，如果大于表长，就没有元素可以归并了
                auto p = cur->next, q = p;//p表示第一段的起始点，q表示第二段的起始点，之后开始归并即可
                for(int k = 0; k < i; k++) q = q->next;
                //l,r用于计数第一段和第二段归并的节点个数，由于当链表长度非2的整数倍时表长会小于i,故而需要加上p && q的边界判断
                int l = 0, r = 0;
                while(l < i && r < i && p && q) //二路归并
                {
                    if(p->val <= q->val)  cur = cur->next = p, p = p->next, l++;
                    else cur = cur->next = q, q = q->next, r++;
                }

                while(l < i && p) cur = cur->next = p, p = p->next ,l++;
                while(r < i && q) cur = cur->next = q, q = q->next ,r++;
                cur->next = q;//记得把排好序的链表尾链接到下一链表的表头，循环完毕后q为下一链表表头
            }
        }
        return dummy->next6+
    }
```

```
40 };
```

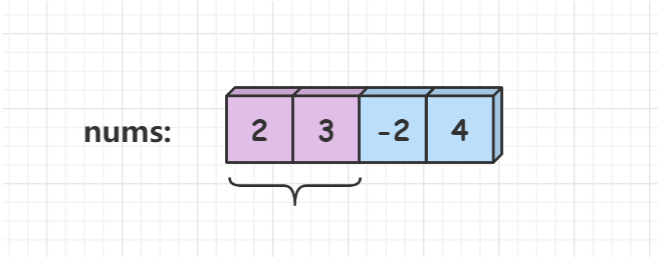
152. 乘积最大子数组

思路

(动态规划) $O(n)$

给你一个整数数组 `nums` ，让我们找出数组中乘积最大的连续子数组对应的乘积。

样例：



如样例所示，`nums = [2,3,-2,4]`，连续子数组 `[2,3]` 有最大乘积 `6`，下面来讲解动态规划的做法。

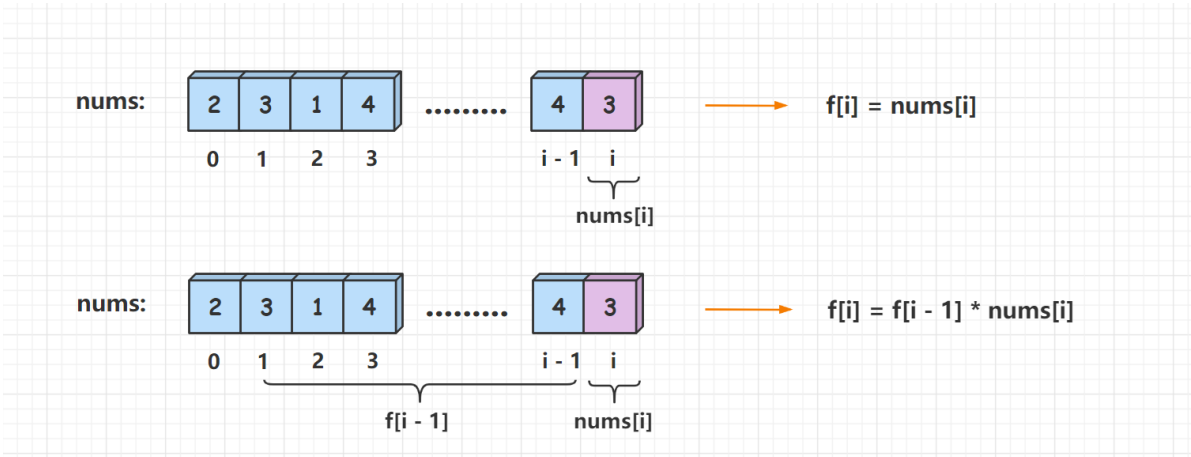
状态表示：

`f[i]` 表示以 `num[i]` 结尾的连续子数组乘积的最大值。

假设 `nums` 数组都是非负数，对于每个以 `nums[i]` 结尾的连续子数组，我们有两种选择方式：

- 1、只有 `nums[i]` 一个数，那么以 `num[i]` 结尾的连续子数组乘积的最大值则为 `nums[i]` ，即 `f[i] = nums[i]`。
- 2、以 `nums[i]` 为结尾的多个数连续组成的子数组，那么问题就转化成了以 `nums[i - 1]` 结尾的连续子数组的最大值再乘以 `nums[i]` 的值，即 `f[i] = f[i - 1] * nums[i]`。

图示：



最后的结果是两种选择中取最大的一个，因此**状态转移方程为：** `f[i] = max(nums[i], f[i - 1] * nums[i])`。

但是 `nums` 数组中包含有正数，负数和零，当前的最大值如果乘以一个负数就会变成最小值，当前的最小值如果乘以一个负数就会变成一个最大值，因此我们还需要维护一个最小值。

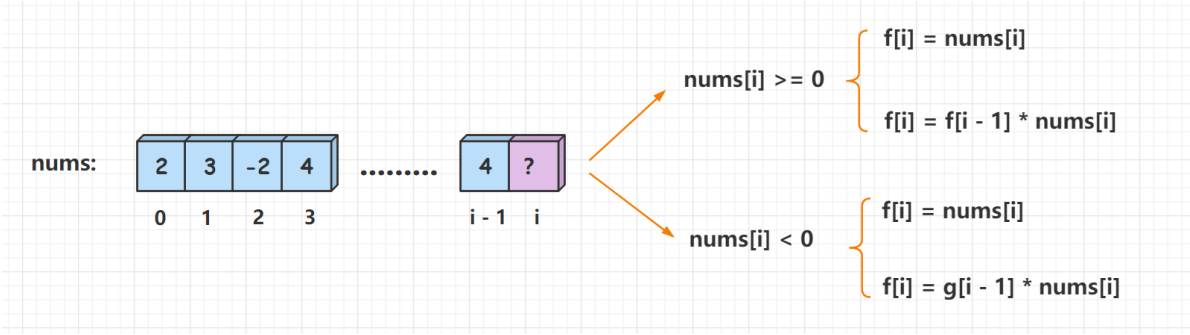
新的状态表示：

`f[i]` 表示以 `num[i]` 结尾的连续子数组乘积的最大值，`g[i]` 表示以 `num[i]` 结尾的连续子数组乘积的最小值。

我们先去讨论以 `nums[i]` 结尾的连续子数组的最大值的状态转移方程：

- 1、如果`nums[i] >= 0`，同刚开始讨论的一样，`f[i] = max(nums[i], f[i - 1] * nums[i])`。
- 2、如果`nums[i] < 0`，只有`nums[i]`一个数，最大值为`nums[i]`。有多个数的话，问题就转化成了以`nums[i - 1]`结尾的连续子数组的最小值再乘以`nums[i]`(最小值乘以一个负数变成最大值)，即`f[i] = max(nums[i], g[i - 1] * nums[i])`。

图示：

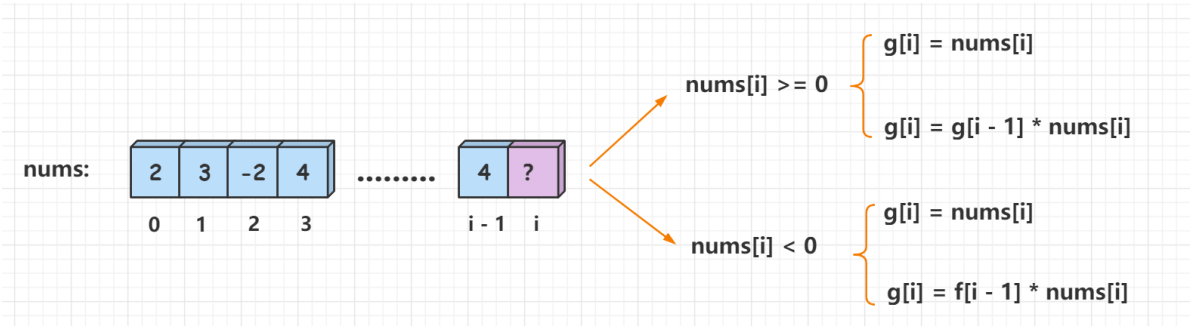


综上，最大值的状态转移方程为：`f[i] = max(nums[i], max(f[i - 1] * nums[i], g[i - 1] * nums[i]))`。

再去讨论以`nums[i]`结尾的连续子数组的最小值的状态转移方程：

- 1、如果`nums[i] >= 0`，同最大值的思考方式一样，只需把`max`换成`min`，即`g[i] = min(nums[i], g[i - 1] * nums[i])`。
- 2、如果`nums[i] < 0`，只有`nums[i]`一个数，最小值为`nums[i]`。有多个数的话，问题就转化成了以`nums[i - 1]`结尾的连续子数组的最大值再乘以`nums[i]`(最大值乘以一个负数变成最小值)，即`f[i] = min(nums[i], f[i - 1] * nums[i])`。

图示：



综上，最小值的状态转移方程为：`g[i] = min(nums[i], min(g[i - 1] * nums[i], f[i - 1] * nums[i]))`。

最后的结果就是分别以`nums[0]`或`nums[1]`，，，或`nums[i]`为结尾的连续子数组中取乘积结果最大的。

初始化：

只有一个数`nums[0]`时，以`nums[i]`结尾的连续子数组乘积的最大值和最小值都为`nums[0]`。

时间复杂度分析： 只遍历一次`nums`数组，因此时间复杂度为$O(n)$，n是`nums`数组的长度。

c++代码

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int n = nums.size();
```

```
        vector<int>f(n + 1), g(n + 1);
        f[0] = nums[0], g[0] = nums[0];
        int res = nums[0];
        for(int i = 1; i < n; i++){
            f[i] = max(nums[i], max(f[i - 1] * nums[i], g[i - 1] *
nums[i]));
            g[i] = min(nums[i], min(g[i - 1] * nums[i], f[i - 1] *
nums[i]));
            res = max(res, f[i]);
        }
        return res;
    }
};
```

155. 最小栈

思路

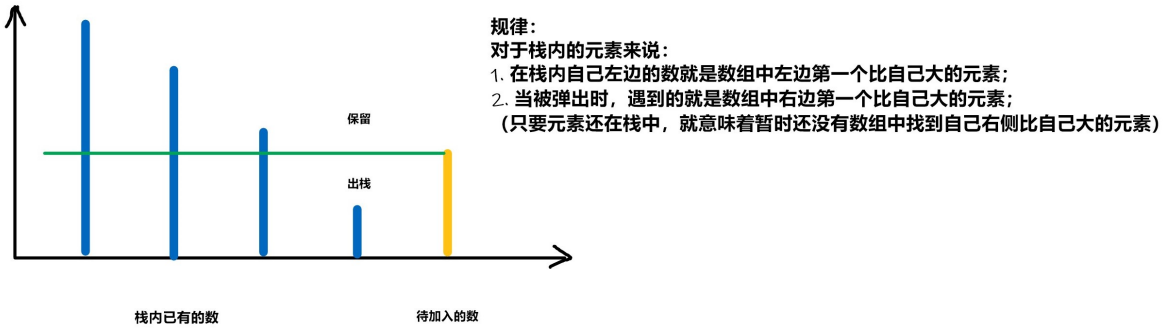
(单调栈) $O(1)$

我们除了维护基本的栈结构之外，还需要维护一个单调递减栈，来实现返回最小值的操作。

下面介绍如何维护单调递减栈：

- 当我们向栈中压入一个数时，如果该数 $\leq$ 单调栈的栈顶元素，则将该数同时压入单调栈中；否则，不压入，这是由于栈具有先进后出性质，所以在该数被弹出之前，栈中一直存在一个数比该数小，所以该数一定不会被当做最小数输出。
- 当我们从栈中弹出一个数时，如果该数等于单调栈的栈顶元素，则同时将单调栈的栈顶元素弹出。
- 单调栈的栈顶元素，就是当前栈中的最小数。

单调递减栈



时间复杂度分析：四种操作都只有常数次入栈出栈操作，所以时间复杂度都是 $O(1)$.

c++代码

```
class MinStack {
public:

    stack<int> stackValue;
    stack<int> stackMin;  //单调递减栈
    MinStack() {

    }

    void push(int val) {
        stackValue.push(val);
        if(stackMin.empty() || stackMin.top() >= val) stackMin.push(val);
```

```
    }

    void pop() {
        if(stackValue.top() == stackMin.top()) stackMin.pop();
        stackValue.pop();
    }

    int top() {
        return stackValue.top();
    }

    int getMin() {
        return stackMin.top();
    }
};

```

160. 相交链表

思路

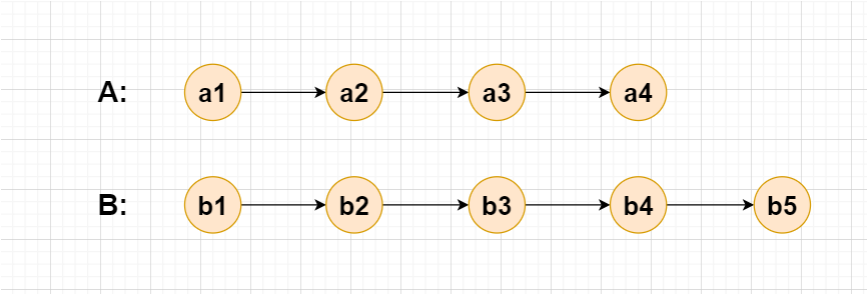
(链表，指针扫描) $O(n)$

算法步骤：

1. 用两个指针分别从两个链表头部开始扫描，每次分别走一步；
2. 如果一个指针走到`null`，则从另一个链表头部开始走；
3. 当两个指针相同时，
    - 如果指针不是`null`，则指针位置就是相遇点；
    - 如果指针是`null`，则两个链表不相交；

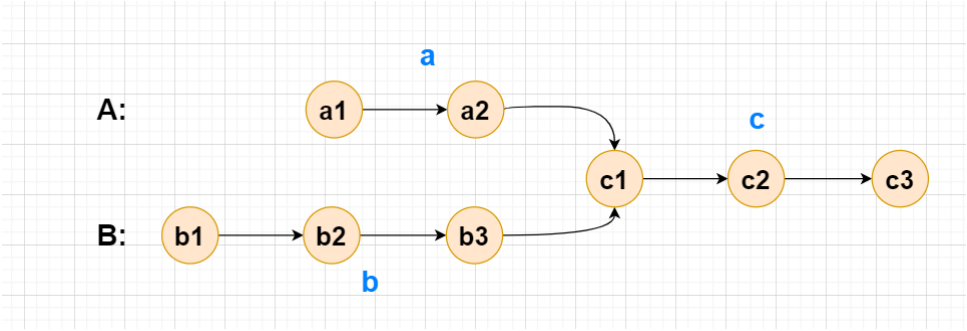
此题我们画图讲解，一目了然：

1、两个链表不相交：



`a`,`b` 分别代表两个链表的长度，则两个指针分别走 `a+b` 步后都变成 `null`。

2 . 两个链表相交：



则两个指针分别走 `a + b + c` 步后在两链表交汇处相遇。

时间复杂度分析：每个指针走的长度不大于两个链表的总长度，所以时间复杂度是$O(n)$。

c++代码

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        auto pA = headA, pB = headB; //定义两个指针
        while(pA != pB){
            if(pA) pA = pA->next;
            else pA = headB;
            if(pB) pB = pB->next;
            else pB = headA;
        }
        return pA;
    }
};
```

169. 多数元素

思路

(投票算法) $O(n)$

当一个国家的总统候选人 `r` 的支持率大于50%的话，即使每个反对他的人都给他投一个反对票，抵消掉他的支持票，他的支持票也不会被完全消耗掉。因此，我们可以假定和 `r` 相同的数都是支持票，和 `r` 不同的数都是反对票。

维护两个变量：候选人和他的票数

- 1、候选人初始化为 `r = 0`，票数 `c` 初始化为 `0`，遍历整个数组
- 2、当候选人的票数为 `0` 时，更换候选人，并将票数重置为 `1`（默认自己投自己一票）
- 3、当候选人的值和当前元素相同时，票数加 `1`，否则减 `1`
- 4、最后维护的候选人即是答案

时间复杂度分析： $O(n)$，n是数组的大小。

空间复杂度分析： 仅使用了两个变量，故需要 $O(1)$ 的额外空间。

c++代码

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int r = 0, c = 0;
        for(int x : nums){
            if(c == 0) r = x, c = 1;
            else if(x == r) c++;
            else c--;
        }
        return r;
    }
};
```

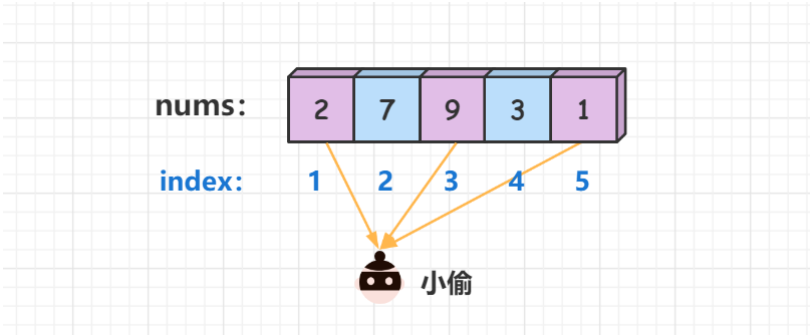
198. 打家劫舍

思路

(动态规划) $O(n)$

给定一个代表金额的非负整数数组`nums`，相邻房间不可偷，让我们输出可以偷窃到的最高金额。

样例:



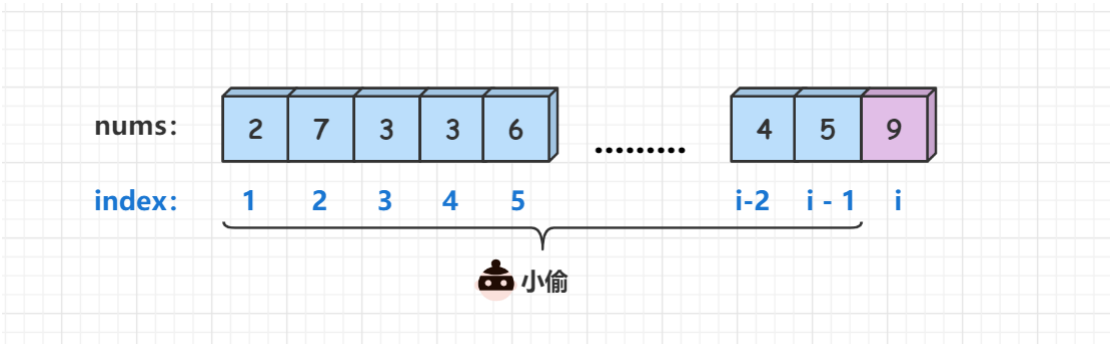
如样例所示，`nums = [2,7,9,3,1]`，偷窃`1`，`3`，`5`号房间可以获得最高金额`12`，下面来讲解动态规划的做法。

状态表示: `f[i]`表示偷窃`1`号到`i`号房间所能获得的最高金额。那么，`f[n]`就表示偷窃`1`号到`n`号房间所能获得的最高金额，即为答案。

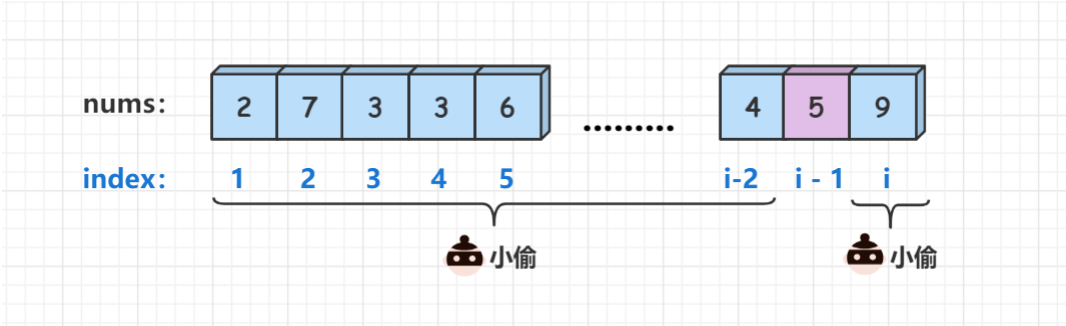
状态计算:

假设有`i`间房间，考虑最后一间偷还是不偷房间，有两种选择方案:

- 1、偷窃前`i-1`间房间，不偷窃最后一间房间，那么问题就转化为了偷窃`1`号到`i - 1`号房间所能获得的最高金额，即`f[i] = f[i-1]`。



- 2、偷窃前`i - 2`间房间和最后一间房间(相邻的房屋不可闯入)，那么问题就转化为了偷窃`1`号到`i- 2`号房间所能获得的最高金额再加上偷窃第`i`号房间的金额，即`f[i] = f[i - 2] + nums[i]`。(下标均从`1`开始)



两种方案，选择其中金额最大的一个。因此**状态转移方程为：** `f[i] = max(f[i - 1], f[i - 2] + nums[i])`。（下标均从`1`开始）

初始化： `f[1] = nums[0]`，偷窃`1`号房间所能获得的最高金额为`nums[0]`。

实现细节：

我们定义的状态表示`f[]`数组和`nums[]`数组下标均是从`1`开始的，而题目给出的`nums[]`数组下标是从`0`开始的。为了一一对应，状态转移方程中的`nums[i]`的值要往前错一位，取`nums[i - 1]`，这点细节希望大家可以注意一下。

时间复杂度分析： $O(n)$，其中 n是数组长度。只需要对数组遍历一次。

c++代码

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        vector<int>f(n + 1);
        f[1] = nums[0];
        for(int i = 2; i <= n; i++){
            f[i] = max(f[i - 1], f[i - 2] + nums[i - 1]);
        }
        return f[n];
    }
};
```

338. 比特位计数

思路

(动态规划) $O(n)$

状态表示： `f[i]`表示`i`的二进制表示中`1`的个数。

状态计算：

考虑`i`的奇偶性，有两种不同选择：

- `i`是偶数，则`f[i] = f[i/2]`，因为`i/2 * 2`本质上是`i/2`的二进制左移一位，低位补零，所以`1`的数量不变。
- `i`是奇数，则`f[i] = f[i - 1] + 1`，因为如果`i`为奇数，那么`i - 1`必定为偶数，而偶数的二进制最低位一定是`0`，那么该偶数`+1`后最低位变为`1`且不会进位，所以奇数比它上一个偶数二进制表示上多一个`1`。

初始化： `f[0] = 0`。

时间复杂度分析： $O(n)$。

c++代码

```
class Solution {
public:
    vector<int> countBits(int n) {
        vector<int> f(n + 1);
        f[0] = 0; //初始化
        for(int i = 1; i <= n; i++){
            if(i & 1) f[i] = f[i - 1] + 1;
            else f[i] = f[i >> 1];
        }
        return f;
    }
};
```

200. 岛屿数量

思路

(深度优先遍历)

1. 从任意一个陆地点开始，即可通过四连通的方式，深度优先搜索遍历到所有与之相连的陆地，即遍历完整个岛屿。每次将遍历过的点清 `0`。
2. 重复以上过程，可行起点的数量就是答案。

时间复杂度分析：由于每个点最多被遍历一次，故时间复杂度为 $O(n*m)$

空间复杂度分析：最坏情况下，需要额外$O(n*m)$ 的空间作为系统栈。

c++代码

```
class Solution {
public:
    vector<vector<char>> g;
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    int numIslands(vector<vector<char>>& grid) {
        g = grid;
        int cnt = 0;
        for(int i = 0; i < g.size(); i++)
            for(int j = 0; j < g[i].size(); j++){
                if(g[i][j] == '1'){
                    dfs(i, j);
                    cnt++;
                }
            }
        return cnt;
    }

    void dfs(int x, int y){
        g[x][y] = '0';
        for(int i = 0; i < 4; i++){
            int a = x + dx[i], b = y + dy[i];
            if(a < 0 || a >= g.size() || b < 0 || b >= g[a].size() || g[a]
[b] == '0') continue;
            dfs(a, b);
        }
    }
};
```

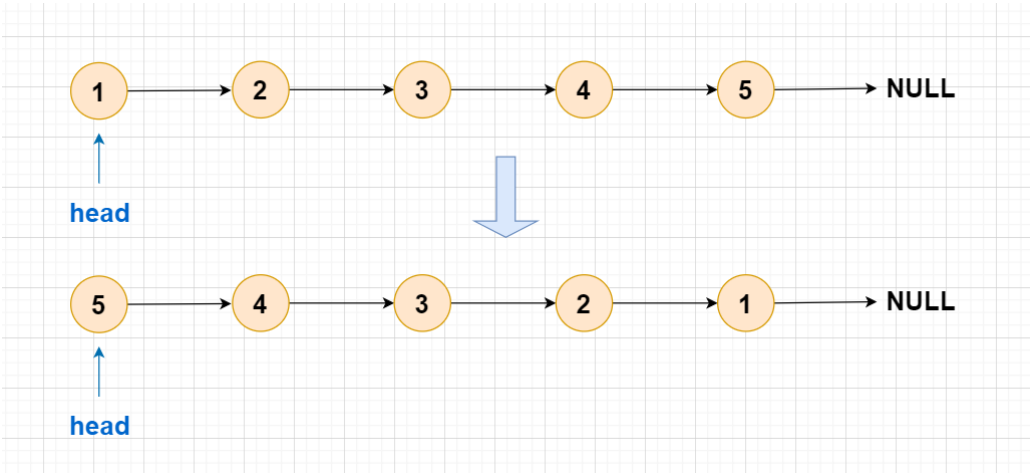
206. 反转链表

思路

(双指针，迭代) (n)

给定一个链表的头节点，让我们反转该链表并输出反转后链表的头节点。

样例：



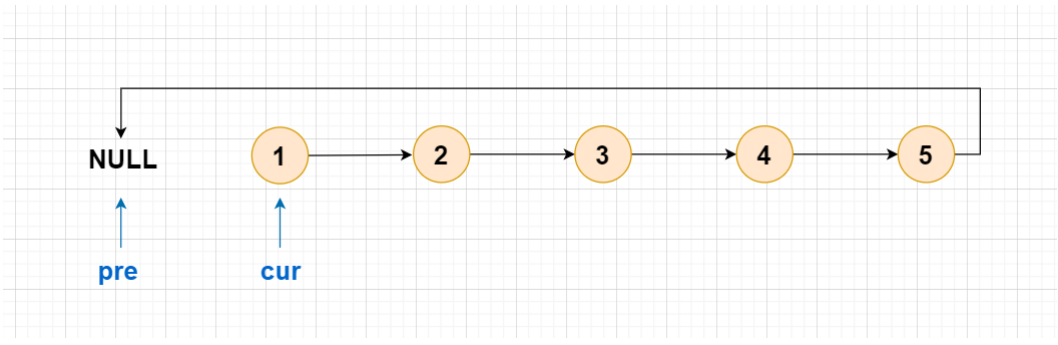
如样例所示，原始链表为`1->2->3->4->5->NULL`，我们将其翻转输出`5->4->3->2->1->NULL`。下面我们来讲解双指针的做法。

将一个链表翻转，即将该链表所有节点的`next`指针指向它的前驱节点。由于是单链表，我们在遍历时并不能直接找到其前驱节点，因此我们需要定义一个指针保存其前驱节点。

每次翻转时，我们都需要修改当前节点的`next`指针。如果不在改变当前节点的`next`指针前保存其后继节点，那么我们就失去了当前节点和后序节点的联系，因此还需要额外定义一个指针用于保存当前节点的后继节点。

具体过程如下：

1、定义一个前驱指针`pre`和`cur`指针，`pre`指针用来指向前驱节点，`cur`指针用来遍历整个链表，初始化`pre = null`，`cur = head`。

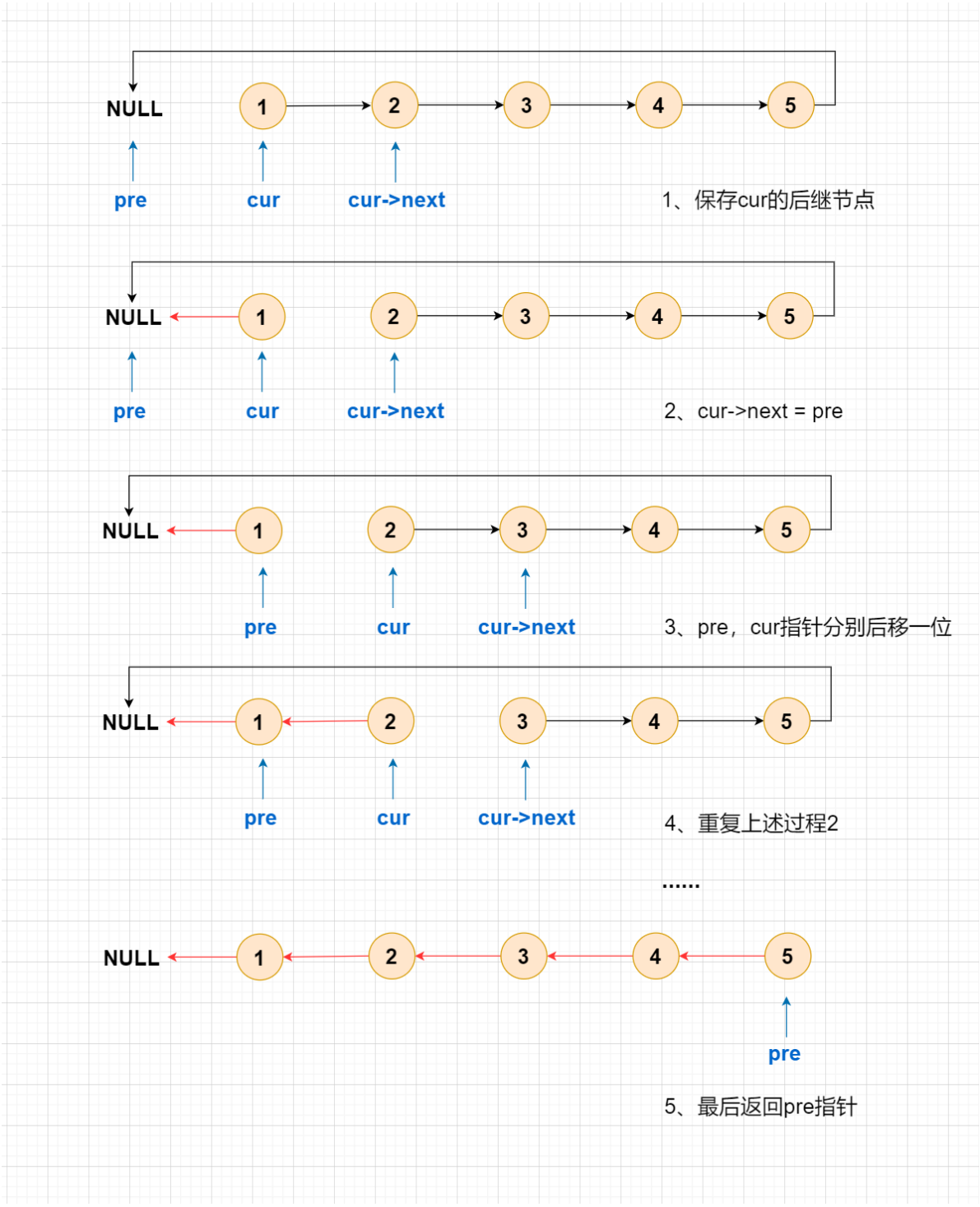


2、我们首先保存`cur`指针指向节点的后继节点，然后让`cur`指针指向节点的`next`指针指向其前驱节点，即`cur->next = pre`。

3、`pre`指针和`cur`指针分别后移一位，重复上述过程，直到`cur`指向空节点。

4、最后我们返回`pre`节点。

图示过程如下：



时间复杂度分析：只遍历一次链表，时间复杂度是$O(n)$。

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
```

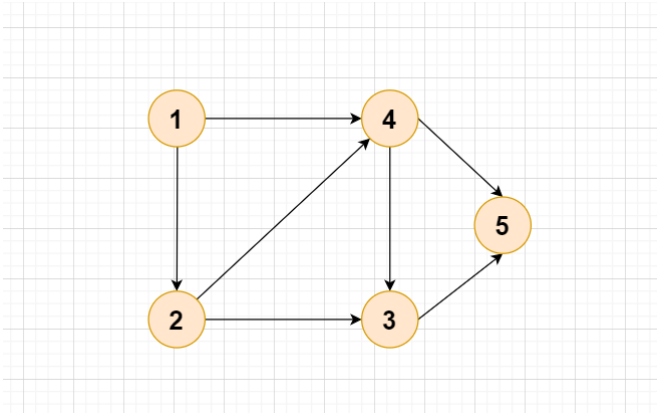
```
    ListNode* reverseList(ListNode* head) {
        ListNode* pre = nullptr; //前驱指针
        ListNode* cur = head;
        while(cur){
            ListNode* t = cur->next;  //先保存后继节点
            cur->next = pre;
            pre = cur, cur = t;
        }
        return pre;
    }
};
```

207. 课程表

思路

拓扑排序： $O(n+m)$

对一个有向无环图`G`进行拓扑排序，是将`G`中所有顶点排成一个线性序列，使得图中任意一对顶点`u`和`v`，若`<u，v> ∈E(G)`，则`u`在线性序列中出现在`v`之前。



一个合法的选课序列就是一个拓扑序，拓扑序是指一个满足有向图上，不存在一条边出节点在入节点前的线性序列，如果有向图中有环，就不存在拓扑序。可以通过拓扑排序算法来得到拓扑序，以及判断是否存在环。

拓扑排序步骤：

- 1、建图并记录所有节点的入度。
- 2、将所有入度为`0`的节点加入队列。
- 3、取出队首的元素`now`，将其加入拓扑序列。
- 4、访问所有`now`的邻接点`nxt`，将`nxt`的入度减`1`，当减到`0`后，将`nxt`加入队列。
- 5、重复步骤`3`、`4`，直到队列为空。
- 6、如果拓扑序列个数等于节点数，代表该有向图无环，且存在拓扑序。

时间复杂度分析：假设 n 为点数，m 为边数，拓扑排序仅遍历所有的点和边一次，故总时间复杂度为 $O(n+m)$。

c++代码

```
class Solution {
public:
    /**
     1、建图并记录所有节点的入度。
     2、将所有入度为`0`的节点加入队列。
     3、取出队首的元素`now`，将其加入拓扑序列。
```

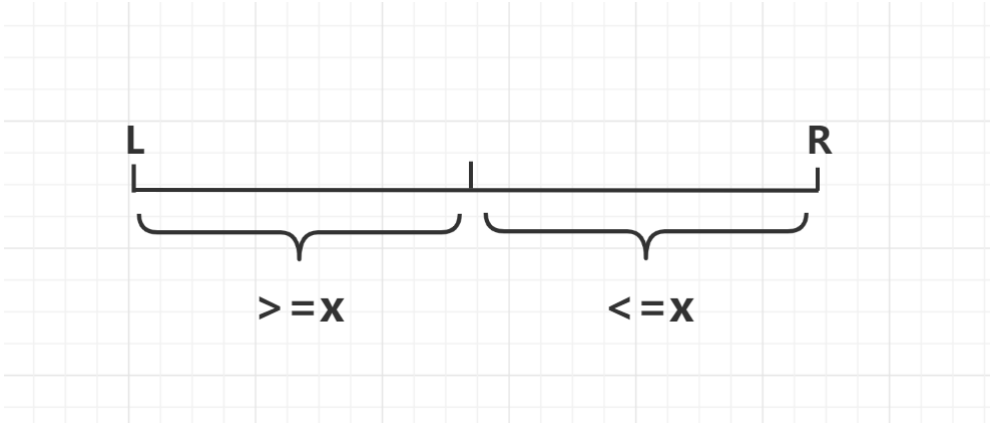
```
     4、访问所有`now`的邻接点`nxt`，将`nxt`的入度减`1`，当减到`0`后，将`nxt`加入队列。
     5、重复步骤`3`、`4`，直到队列为空。
     6、如果拓扑序列个数等于节点数，代表该有向图无环，且存在拓扑序。
    **/
    bool canFinish(int n, vector<vector<int>>& edges) {
        vector<vector<int>> g(n);
        vector<int> d(n);  // 存贮每个节点的入度
        for(auto edge : edges){
            g[edge[1]].push_back(edge[0]);  //建图
            d[edge[0]]++;  //入度加1
        }

        queue<int> q;
        for(int i = 0; i < n; i++){
            if(d[i] == 0) q.push(i);  //将所有入度为0的节点加入队列。
        }

        int cnt = 0;  //统计拓扑节点的个数
        while(q.size()){
            int t = q.front();
            q.pop();
            cnt++;
            for(int i : g[t]){  //访问t的邻接节点
                d[i]--;
                if(d[i] == 0) q.push(i);
            }
        }

        return cnt == n;
    }
};
```

215. 数组中的第K个最大元素

思路

快速选择 $O(n)$

快选是在快排的基础上只递归一半区间。



如果当前要找的数>= x递归左区间，否则递归右区间

具体过程：

- 1、在特定区间`[l, r]`中，选中某个数`x`，将大于等于`x`的放在左边，小于`x`的放在右边，其中`[l, j]`是大于等于`x`的区间，`[j + 1,r]`是小于`x`的区间。
- 2、判断出第`k`大与`j`的大小关系，若`k <= j`，则递归到`[l, j]`区间，否则递归到`[j + 1,r]`的区间

注意：此处求的是第`k`大，而里面的方法`k`是指第`k`个位置，需要变成`k - 1`。

c++代码

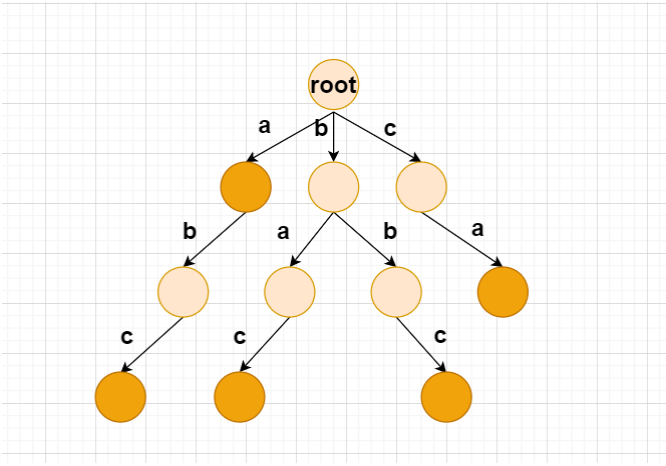
```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        return quick_sort(nums, 0, nums.size() - 1, k - 1); //注意下标
    }

    int quick_sort(vector<int>& nums, int l ,int r, int k){
        if(l == r) return nums[l];
        int x = nums[l], i = l - 1, j = r + 1;
        while(i < j){
            do i++; while(nums[i] > x);
            do j--; while(nums[j] < x);
            if(i < j) swap(nums[i], nums[j]);
        }

        if(k <= j) return quick_sort(nums, l, j, k);
        else return quick_sort(nums, j + 1, r, k);
    }
};
```

208. 实现 Trie (前缀树)

思路

字典树

字典树，顾名思义，是关于“字典”的一棵树。即：它是对于字典的一种存储方式（所以是一种数据结构而不是算法）。这个词典中的每个“单词”就是从根节点出发一直到某一个目标节点的路径，路径中每条边的字母连起来就是一个单词。



标橙色的节点是“目标节点”，即根节点到这个目标节点的路径上的所有字母构成了一个单词。

作用：

- 1、维护字符串集合（**字典**）
- 2、向字符串集合中插入字符串（**建树**）

- 3、查询字符串集合中是否有某个字符串（**查询**）
- 4、查询字符串集合中是否有某个字符串的前缀（**查询**）

具体操作：

定义字典树节点

```
struct Node {
    bool is_end;    // 表示是否存在以这个点为结尾的单词
    Node *son[26];  // 26个小写字母子结点
    Node() {        // 初始化
        is_end = false;
        for (int i = 0; i < 26; i ++ )
            son[i] = NULL;
    }
}*root;
```

向字典树中插入一个单词`word`

从根结点出发，沿着字符串的字符一直往下走，若某一字符不存在，则直接把它创建出来，继续走下去，走完了整个单词，标记最后的位置的`is_end = true`。

```
void insert(string word) {
    auto p = root;
    for (auto c: word) {
        int u = c - 'a';
        if (!p->son[u]) p->son[u] = new Node();
        p = p->son[u];
    }
    p->is_end = true;
}
```

查找字典树中是否存在单词`word`

从根结点出发，沿着字符串的字符一直往下走，若某一字符不存在，则直接`return false`，当很顺利走到最后的位置的时候，判断最后一个位置的`is_end`即可。

```
bool search(string word) {
    auto p = root;
    for (auto c: word) {
        int u = c - 'a';
        if (!p->son[u]) return false;
        p = p->son[u];
    }
    return p->is_end;
}
```

查找字典树中是否有以`prefix`为前缀的单词

从根结点出发，沿着字符串的字符一直往下走，若某一字符不存在，则直接`return false`，如果顺利走到最后一个位置，则返回`true`。

```
bool startsWith(string word) {
    auto p = root;
    for (auto c: word) {
        int u = c - 'a';
        if (!p->son[u]) return false;
        p = p->son[u];
    }
    return true;
}
```

时间复杂度分析： $O(n)$，`n`表示单词操作字符串长度。

c++代码

```
class Trie {
public:
    struct Node{
        bool is_end;
        Node *son[26];
        Node(){
            is_end = false;
            for(int i = 0; i < 26; i++)
                son[i] = NULL;
        }
    }*root;
    Trie() {
        root = new Node();
    }

    void insert(string word) {
        auto p = root;
        for(auto c : word){
            int u = c - 'a';
            if(!p->son[u]) p->son[u] = new Node();
            p = p->son[u];
        }
        p->is_end = true;
    }

    bool search(string word) {
        auto p = root;
        for(auto c : word){
            int u = c - 'a';
            if(!p->son[u]) return false;
            p = p->son[u];
        }
        return p->is_end;
    }

    bool startsWith(string prefix) {
        auto p = root;
        for(auto c : prefix){
            int u = c -'a';
            if(!p->son[u]) return false;
            p = p->son[u];
        }
        return true;
```

```
    }
};

/**
 * Your Trie object will be instantiated and called as such:
 * Trie* obj = new Trie();
 * obj->insert(word);
 * bool param_2 = obj->search(word);
 * bool param_3 = obj->startsWith(prefix);
 */
```

221. 最大正方形

思路

(动态规划) $O(nm)$

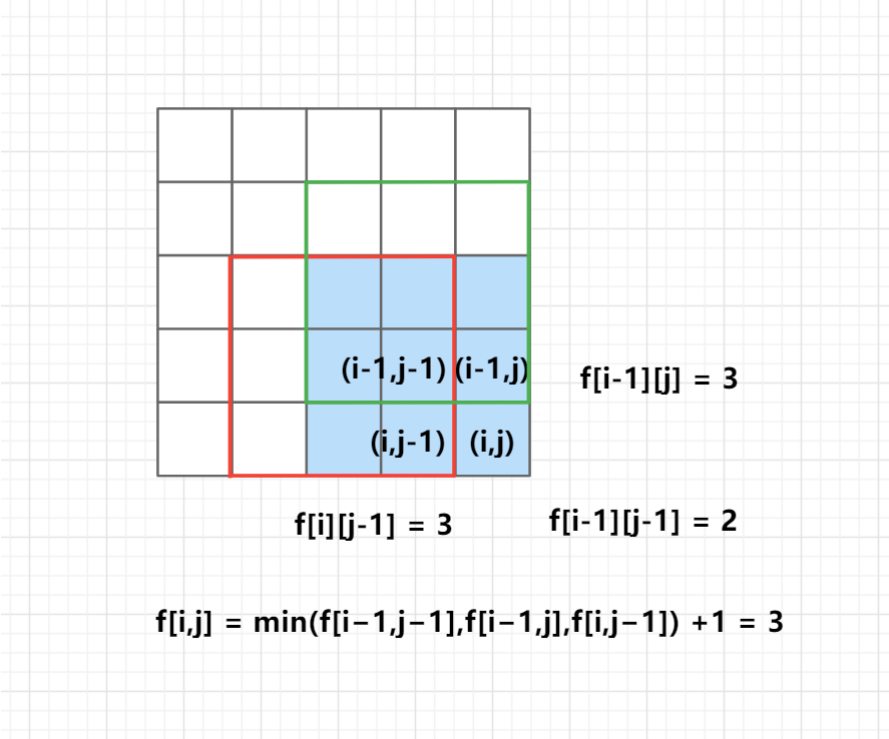
状态表示： `f[i][j]`表示所有以`(i,j)`为右下角的且只包含`1`的正方形的边长最大值。

状态计算：

对于每个位置`(i, j)`，检查在矩阵中该位置的值：

- 如果该位置的值是`0`，则`f[i][j] = 0`，因为当前位置不可能在由`1`组成的正方形中。
- 如果该位置的值是`1`，则`f[i][j]`的值由其上方、左方和左上方的三个相邻位置的状态值决定。具体而言，当前位置的元素值等于三个相邻位置的元素中的最小值加`1`。

状态转移方程： `f[i,j] = min(f[i-1,j-1],f[i-1,j],f[i,j-1]) + 1`



类似于 木桶的短板理论， 附近的最小边长，才与`(i,j)`的最长边长有关。

时间复杂度分析： $O(nm)$，其中 n和m是矩阵的行数和列数。

c++代码

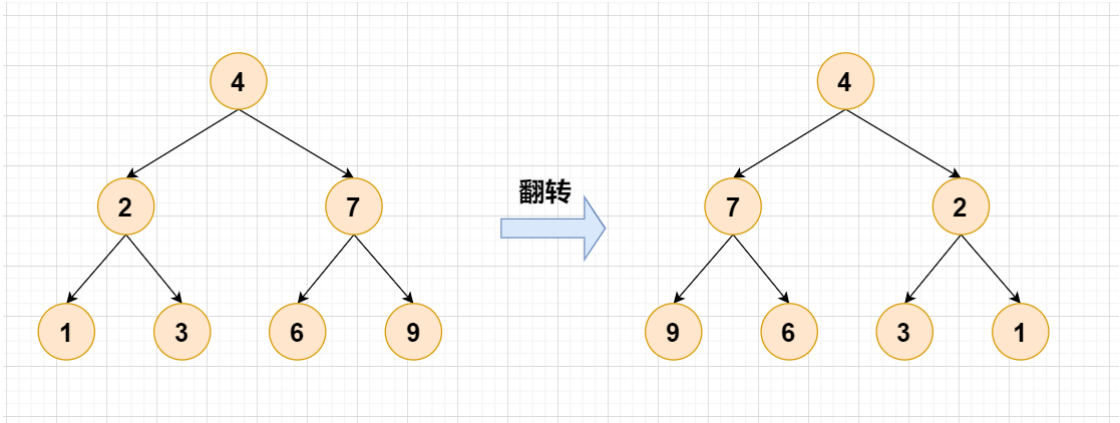
```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
```

```
        int n = matrix.size(), m = matrix[0].size();
        if(!n || !m) return 0;
        vector<vector<int>>f(n + 1, vector<int>(m + 1));
        int res = 0;
        for(int i = 1; i <= n; i++) //为了减少对边界的处理，这里我们下标从1开始
            for(int j = 1; j <= m; j++)
                if(matrix[i - 1][j - 1] == '1')
                {
                    f[i][j] = min(f[i][j - 1], min(f[i - 1][j], f[i - 1][j - 1])) + 1;
                    res = max(res, f[i][j]);
                }
        return res * res;
    }
};
```

226. 翻转二叉树

思路

(递归) $O(n)$



我们可以发现翻转后的树就是将原树的所有节点的左右儿子互换!

所以我们递归遍历原树的所有节点，将每个节点的左右儿子互换即可。

时间复杂度分析：每个节点仅被遍历一次，所以时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return nullptr;
```

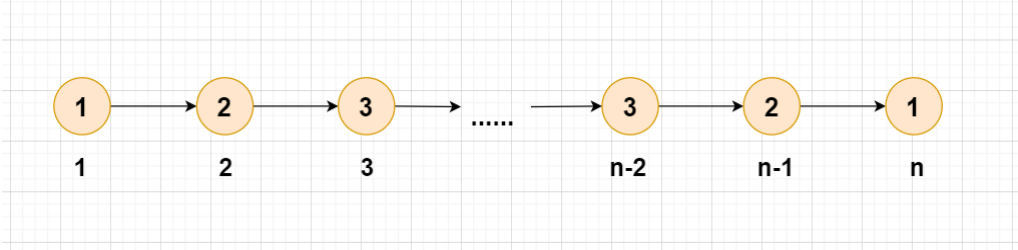
```
        swap(root->left, root->right);
        invertTree(root->left);
        invertTree(root->right);
        return root;
    }
};
```

234. 回文链表

思路

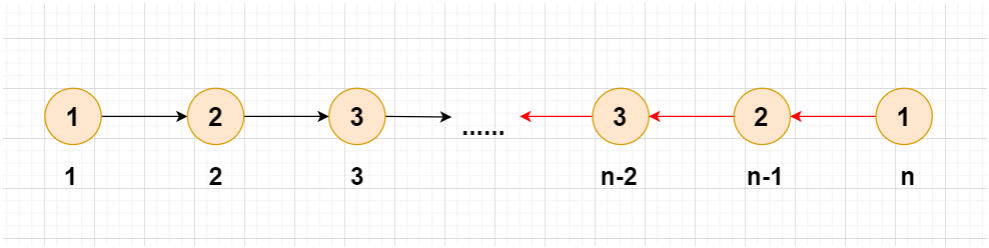
(链表操作)

假设初始的链表是 $L1 \to L2 \to L3 \to \ldots \to Ln$。

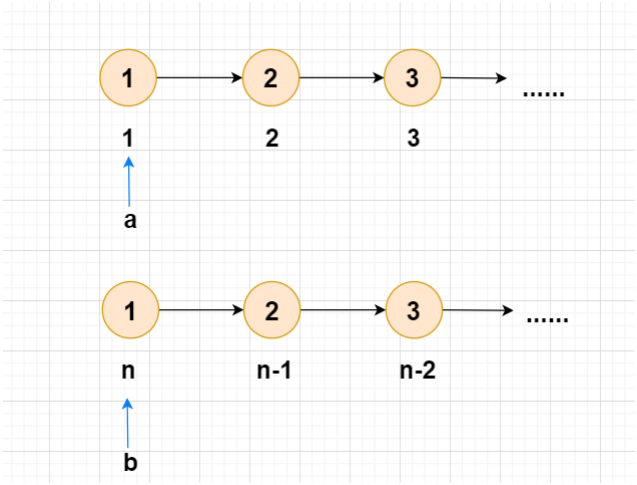


分两步处理：

- 找到链表的中点节点，将其后半段的指针都反向，变成：
  $L1 \to L2 \to L3 \to \ldots \to L\lceil n/2 \rceil \leftarrow L\lceil n/2 \rceil + 1 \leftarrow \ldots \leftarrow Ln;$



- 然后用两个指针分别从链表首尾开始往中间扫描，依次判断对应节点的值是否相等，如果都相等，说明是回文链表，否则不是。

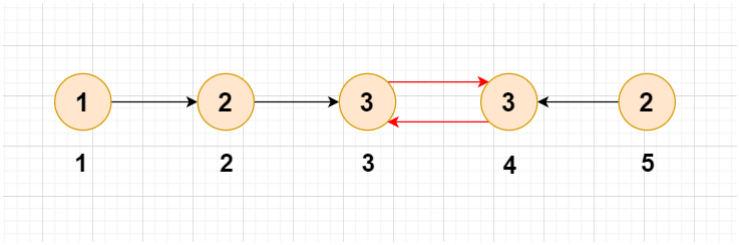


- 最后再将整个链表复原。

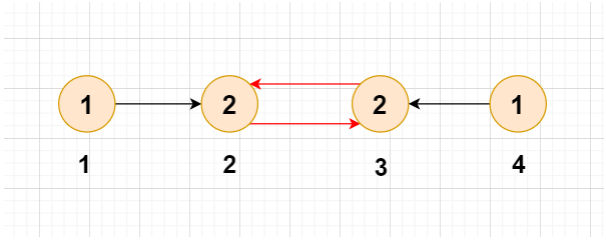
注意：

1、我们选取链表的中点节点为$(n+1)/2$ 下取整，n是链表的节点个数。

2、如果一个链表是奇数个节点(假设为5个节点)，将其后半段翻转完后的链表为：



3、如果一个链表是偶数个节点(假设为4个节点)，将其后半段翻转完后的链表为：



连接左右链表节点之间的指向是双向的

4、具体实现细节看代码

空间复杂度分析：链表的迭代翻转算法仅使用额外 $O(1)$ 的空间，所以本题也仅使用额外 $O(1)$ 的空间。

时间复杂度分析：整个链表总共被遍历`4`次，所以时间复杂度是 $O(n)$。

c++代码

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        int n = 0 ; //统计节点的个数
        for(ListNode *p = head ; p ; p = p->next) n++;
        if(n <= 1) return true; //节点数<=1的一定是回文链表
        //找到中点节点，由第一个节点跳(n+1)/2 -1步到达中点节点
        ListNode* a = head;
        for(int i = 0; i < (n+1)/2 - 1; i++) a = a->next; //a指针指向链表中点
        ListNode* b = a->next;  //b指针指向链表中点的下一个节点
        while(b) //将链表的后半段反向
        {
            ListNode* next = b->next; //保留b的next节点
            b->next = a;
            a = b, b = next;
        }6
        //此时a指向链表的尾节点,我们让b指向链表的头节点
        b = head;
        ListNode* tail = a; //保留一下尾节点
        bool res = true;
        for(int i = 0; i < n/2; i++) //判断是否是回文链表
```

```
        {
            if(b->val != a->val)
            {
                res = false;
                break;
            }
            b = b->next;
            a = a->next;
        }
        //将链表复原，后半段链表翻转
        //a指向尾节点，b指向a的下一个节点
        a = tail, b = a->next;
        for(int i = 0; i < n/2; i++)
        {
            ListNode* next = b->next;
            b->next = a;
            a = b , b = next;
        }
        tail->next = 0;
        return res;
    }
};
```