# 1 认识复杂度和简单排序算法

## 1.1 常数操作

与数据量的大小无关

int a = arr[i];  计算一个偏移量即可，属于常数操作

list.get(i);  链表需要一个一个遍历，一个一个跳，不是常数操作

加减乘除是常数操作

## 1.2 选择排序

### 1.2.1 原理

0~N-1，找到最小值的索引，值放到 0 位置上；

1~N-1，找到最小值的索引，值放到 1 位置上；

……

### 1.2.2 时间复杂度（常数操作数量的指标）

看：N + (N-1) + (N-2)+……

比：N + (N-1) + (N-2)+……

换：N

常数操作次数 $= aN^2 + bN + c$，不要低阶项，不要系数

$O(N^2)$

### 1.2.3 评价算法好坏

先看时间复杂度，当时间复杂度相同时，再分析实际运行时间（如加减乘除与位运算或与异或），以此比较出常数项时间

### 1.2.4 代码

```
void selectSort(int* arr, int len)
{
    if (arr == NULL || len < 2)
    {
        return;
    }
    for (int i = 0; i < len-1; i++)    //i~N-1
    {
        int minIndex = i; //记录最小值的索引
        for (int j = i + 1; j < len; j++)  //i~N-1
        {
            minIndex = arr[j] < arr[minIndex] ? j : minIndex;
        }
        swap(arr, i, minIndex);
    }
}
```

### 1.2.5 额外空间复杂度

额外空间复杂度 O(1)——只需要几个额外变量

若是需要额外数组，且大小与原数组相同，则为 O(N)

## 1.3 冒泡排序

### 1.3.1 原理

每次都从 0 位置开始，相邻谁大谁右移，第一轮搞定 N-1 位置上的数，第二轮搞定 N-2 位置上的数

### 1.3.2 时间复杂度

$O(N^2)$

### 1.3.3 额外空间复杂度

O(1)

### 1.3.4 代码

```
void bubbleSort(int* arr, int len)
{
    if (arr == NULL || len < 2)
    {
        return;
    }
    for (int end = len - 1; end > 0; end--)    //每次搞定最后一个数, N-1, N-2,
    {
        for (int j = 0; j < end; j++)//0~end
        {
            if (arr[j] > arr[j + 1])  //相邻谁大谁右移
            {
                swap(arr, j, j + 1);
            }
        }
    }
}
```

## 1.4 异或^

### 1.4.1 基本原理

两个数异或，相同为 0 不同为 1。或者理解为无进位相加。

(1) 0 ^ N = N

(2) N ^ N = 0

(3) 满足交换律和结合律

(4) 一堆数异或，只跟数量有关，与顺序无关

### 1.4.2 swap（数组中前提：i、j 是内存的两块不同位置）

```
int a =  甲;
```

int b = 乙;

a = a ^ b;　　a = 甲^乙；b = 乙；

b = a ^ b;　　a = 甲^乙；b = 甲^乙^乙 = 甲；

a = a ^ b;　　a = 甲^乙^甲 = 乙；b = 甲；

### 1.5 异或面试题

#### 1.5.1 问题描述

一个数组内全是 int，只有一种数出现了奇数次，其他出现了偶数次

(1) 如何找到出现奇数次的数？

(2) 如果两种数出现了奇数次，其他偶数次，如何找出这两个数？

#### 1.5.2 第一问解

准备一个 eor = 0，把数组内的每一个数都异或一遍，得到的结果即为出现奇数次的数。

```
void printOddTimesNum1(int* arr, int len)
{
    int eor = 0;
    for (int i = 0; i < len; i++)
    {
        eor ^= arr[i];
    }
    cout << eor << endl;
}
```

注：无进位相加。每一位的值由所有数该位上的 1 的个数决定。

#### 1.5.3 第二问解

先准备一个 eor，把数组所有数都异或一遍，最终 eor = a ^ b

由于 a≠b，那么 eor≠0，说明 eor 的某一位不等于 0，即 eor 的某一位是 1

假设第八位是 1，说明 a、b 的第八位不同，一个是 1 一个是 0

准备一个新的 eor’，把所有第八位是 1 的数异或一遍，就可以得到 a 或者 b

再用 eor’异或 eor，得到另一个数

| 出现偶数次的 other1<br><br>a 和 b 不在同一个集合内，假设 a 在这<br><br>a<br><br>eor’异或第八位是 1 的所有数<br>eor’ = a； eor’ ^ eor = b；<br>第八位是 1 | 出现偶数次的 other2<br><br><br><br>b<br><br><br><br>第八位是 0 |
|---|---|

找到最右侧的 1，eor & (~eor + 1)

eor 1010111100

~eor 0101000011

~eor + 1 0101000100

eor & (~eor + 1) 0000000100

找这一位上全是 1 的数，arr[i] & onlyOne == 1

一个数“与”上 0000000100，想得到 1，那么这个数的右边第三位上一定是 1，这样通过遍历就找出了所有这一位上是 1 的数。

```
void printOddTimesNum2(int* arr, int len)
{
    int eor = 0;
    for (int i = 0; i < len; i++)
    {
        eor ^= arr[i];
    }
    //此时eor = a^b
    //eor != 0
    //eor必有一个位置上是1
    //我们选最右侧的1
    int rightOne = eor & (~eor + 1);

    int onlyOne = 0; //eor'
    for (int i = 0; i < len; i++)
    {
        if ((arr[i] & rightOne) == 1)    //这一位是1的数
        {
```

```
            onlyOne ^= arr[i];      //才^
        }
    }
    cout << onlyOne << " " << (eor ^ onlyOne) << endl;
}
```

### 1.6 插入排序

#### 1.6.1 原理（斗地主抓牌）

0~0 有序

0~1，拿索引为 1 的数往前看，比左小换，一直比下去，直到不换为止

0~2，拿索引为 2 的数往前看……

第 i 轮拿第 i 个数往左比

#### 1.6.2 时间复杂度

数据状况不同，**时间复杂度**不同，**最差交换次数 $O(N^2)$**，最好 O(N)

#### 1.6.3 额外空间复杂度

O(1)

#### 1.6.4 代码

```
void insertionSort(int* arr, int len)
{
    if (arr == NULL || len < 2)
    {
        return;
    }
    //0~0有序，0~i想有序
    for (int i = 1; i < len; i++)
    {
        for (int j = i - 1; j >= 0 && arr[j] > arr[j + 1]; j--)     //j是当前数的前一个位置
        {
```

```
                swap(arr, j, j + 1);
            }
        }
    }
```

## 1.7 二分法

### 1.7.1 在一个有序数组中，找某个数是否存在

遍历 O(N)

找 mid 值，若大于 num，右边的数不要了；若小于 num，左边的数不要了

$O(\log_2 N)$，写作 O(logN)

### 1.7.2 在一个有序数组中，找>=某个数最左侧的位置

假设找>=3 的最左侧的位置。二分，mid 满足，用 t 记录索引，往左找；二分，mid 不满足，往右找；二分，mid 满足，是否比 t 索引更小，更新 t；……；直到二分到结束为止。

### 1.7.3 局部最小值问题

arr 无序，任意相邻数一定不相等，求一个局部最小的位置，好于 O(N)。

局部最小：arr[0] < arr[1]，0 位置是局部最小；arr[N-1] < arr[N-2]，N-1 位置是局部最小；arr[i-1] < arr[i] <arr[i+1]，i 位置是局部最小。

先判断 base case，0 位置和 N-1 位置，若未返回：左侧单调递减，右侧单调递增，则内部必存在一个拐点，也就是局部最小点。二分，mid 若满足要求则返回，不满足，假设 mid-1<mid，那么左侧必存在一个拐点，即局部最小点。

### 1.8 对数器

方法 a：想测的方法

方法 b：很好写，一定对，不考虑时间复杂度的方法

随机样本产生器，产生样本分别给两个方法得到 ret1 和 ret2，ret1==ret2 则对；若不相等，先产生少量样本，把每个方法调整好，再增大样本数量。

### 1.9 递归求最大值

mid = (L+R)/2，可能溢出，mid = L + (R-L)/2 = L+((R-L)>>1)

#### 1.9.1 代码

```
int process(int* arr, int L, int R)
{
    if (L == R)
    {
        return arr[L];
    }
    int mid = L + ((R - L) >> 1);
    int leftMax = process(arr, L, mid);
    int rightMax = process(arr, mid + 1, R);
    return max(leftMax, rightMax);
}
```

利用栈玩了一个遍历（树）

a = 2, b = 2, d = 0　　O(N)

#### 1.9.2 master 公式

$$T(N) = a * T(\frac{N}{b}) + O(N^d)$$

T(N)：母问题数据量

a：子问题调用次数

N/b：子过程规模等量

$O(N^d)$：除了子问题的调用外，剩下的 bigO

时间复杂度

$$log_b a < d, O(N^d)$$

$$log_b a > d, O(N^{log_b a})$$

$$log_b a == d, O(N^d * logN)$$

# 2 认识 O(NlogN)的排序

## 2.1 归并排序

### 2.1.1 原理

从 L 到 R 分两半，M，**分到数组的最小单位**。先将左侧有序，再将右侧有序，再 merge 整合。整合时，准备两个指针一个辅助数组，哪个指针的数小就放到辅助数组里，然后移动指针和辅助数组的下标。

先使用递归，找到最底下的两个数，merge，再依次传上去 merge。

外排序。

### 2.1.2 时间复杂度

a = 2， b = 2， d = 1； logab = d； O(NlogN)

### 2.1.3 额外空间复杂的

O(N)

### 2.1.4 代码

```
class MergeSort
{
public:
    void mergeSort(int* arr, int len)
    {
        if (arr == NULL || len < 2)
        {
            return;
        }
        process(arr, 0, len - 1);
    }

    void process(int* arr, int L, int R)
    {
        if (L == R)
        {
```

```
            return;
        }
        int mid = L + ((R - L) >> 1);
        process(arr, L, mid);
        process(arr, mid + 1, R);
        merge(arr, L, mid, R);
    }

    void merge(int* arr, int L, int M, int R)
    {
        int* temp = new int[R - L + 1];
        for (int j = 0; j < R - L + 1; j++)
        {
            temp[j] = 0;
        }
        int i = 0;
        int p1 = L;
        int p2 = M + 1;
        while (p1 <= M && p2 <= R)
        {
            temp[i++] = arr[p1] <= arr[p2] ? arr[p1++] : arr[p2++];
        }
        while (p1 <= M)
        {
            temp[i++] = arr[p1++];
        }
        while (p2 <= R)
        {
            temp[i++] = arr[p2++];
        }
        for (i = 0; i < R-L+1; i++)
        {
            arr[L + i] = temp[i];
        }
    }
};
```

### 2.1.5 优点

比较行为没有被浪费，变成了整体有序的部分，往下传递进行 merge。

## 2.2 小和问题

### 2.2.1 问题描述

有一个数组，第 i 个位置左边比它小的数的求和，把所有的小和加起来。

### 2.2.2 分析

左边比它小的等价于，第 i 个位置右边有多少个比它大的数，它对小和的贡献为个数×arr[i]。左侧小时才产生小和，并且发生左侧 copy。

利用 merge，在左严格小于右时，res+=左值*(r-p2+1)。

注意，当左右相等时，先 copy 右，边排序边计算小和。

### 2.2.3 代码

```
class SmallSum
{
public:
    int smallSum(int* arr, int len)
    {
        if (arr == NULL || len < 2)
        {
            return 0;
        }
        return process(arr, 0, len - 1);
    }

    int process(int* arr, int l, int r)
    {
        if (l == r)
        {
            return 0;
        }
        int mid = l + ((r - l) >> 1);
        return process(arr, l, mid) + process(arr, mid + 1, r) + merge(arr, l, mid, r);
    }

    int merge(int* arr, int L, int m, int r)
    {
        int* temp = new int[r - L + 1];
```

```
        for (int j = 0; j < r - L + 1; j++)
        {
            temp[j] = 0;
        }
        int i = 0;
        int p1 = L;
        int p2 = m + 1;
        int res = 0;
        while (p1 <= m && p2 <= r)
        {
            res += arr[p1] < arr[p2] ? (r - p2 + 1) * arr[p1] : 0;
            temp[i++] = arr[p1] < arr[p2] ? arr[p1++] : arr[p2++];
        }
        while (p1 <= m)
        {
            temp[i++] = arr[p1++];
        }
        while (p2 <= r)
        {
            temp[i++] = arr[p2++];
        }
        for (i = 0; i < r - L + 1; i++)
        {
            arr[L + i] = temp[i];
        }
        return res;
    }
};
```

### 2.3 逆序对问题

#### 2.3.1 问题描述

在一个数组中，左边的数如果比右边的数大，则这两个数构成一个逆序对，请打印所有逆序对。

#### 2.3.2 分析

右边比左边小，跟 merge 一样。

### 2.3.4 代码（部分）

```
while (p1 <= m && p2 <= r)
{
    res += arr[p1] > arr[p2] ? (r - p2 + 1) : 0;
    temp[i++] = arr[p1] > arr[p2] ? arr[p1++] : arr[p2++];
}
```

## 2.4 快速排序（荷兰国旗问题）

### 2.4.1 问题描述

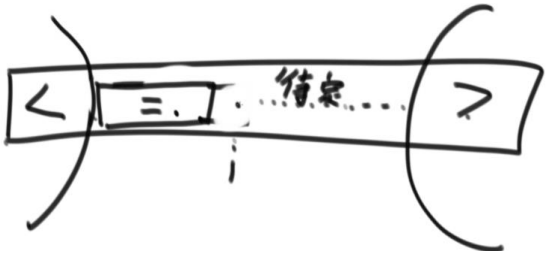
(1) 给定一个数组 arr，和一个数 num，请把小于等于 num 的数放在数组的左边，大于 num 的数放在数组的右边。要求额外空间复杂度 O(1)，时间复杂度 O(N)

(2) 小于放左边，等于放中间，大于放右边

### 2.4.2 分析

(1) 准备一个小于等于区域的右边界，和 i。[i]≤num，[i]和≤区域的下一个数交换，≤区右扩，i++；[i]>num，i++；直到 i 越界。

(2) 准备一个小于区的右边界和大于区的左边界，分别位于两端，和 i。[i]<num，[i]和＜区域的下一个数交换，＜区右扩，i++；[i]=num，i++；[i]>num，[i]和＞区域的前一个数交换，i 原地不动（因为换过来的数是新来的，没见过）；直到＞区与 i 撞上了停。（压缩待定区域）



### 2.4.3 快排 1.0

一个数组，拿最后一个数当作 num，小于等于放左边，大于放右边。分好后，把 num 和大于区的第一个数交换（这样这个数就处在正确位置上了，固定）。接下来对左右两个区域递归，做同样的操作。$O(N^2)$

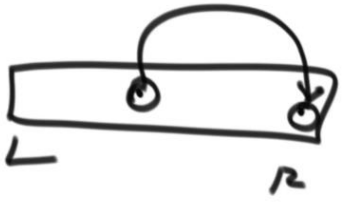
### 2.4.4 快排 2.0

拿最后一个数，把数组分为小于区，等于区，大于区。把 num 和大于区第一个数交换，**一次搞定一批数**，再在左右两区域递归。$O(N^2)$

### 2.4.5 快排 3.0（空 O(logN)）

当前两个版本，划分值选的很偏时，会出现差情况。好情况是划分值打到中间，master 公式 a=2，b=2，d=1，O(NlogN)。越偏越会趋近于 $O(N^2)$。

随机选一个数，把它放到最后一个位置上，再进行划分操作。



### 2.4.6 代码

```
class QuickSort
{
public:
    void quickSort(int* arr, int len)
    {
        if (arr == NULL || len < 2)
        {
            return;
        }
        process(arr, 0, len - 1);
    }

    void process(int* arr, int L, int R)
    {
        if (L < R)
        {
            //swap
            int* p = partition(arr, L, R);
            process(arr, L, p[0] - 1);
            process(arr, p[1] + 1, R);
        }
    }
    //处理arr[l..r]的函数
    //默认以arr[r]做划分，arr[r]->p <p ==p >p
    //返回等于区域（左边界，右边界），所以返回一个长度为2的数组res，
res[0] res[1]
    int* partition(int* arr, int L, int R)
    {
        int less = L - 1;   //<区右边界（-1）
        int more = R;     //>区左边界（R，因为划分值是arr[R]，所以R位置是大
于区左边界）
        while (L < more)//L表示当前数的位置  arr[R]->划分值（当前数未撞到大
于区边界）
        {
            /*
            less [L                              |R(==more)]
            后面less和more都是该区域最边界的索引，符合该区域要求，即less
是小于区最后一个数，more是大于区第一个数
            */
            if (arr[L] < arr[R])    //当前数 < 划分值
            {
                swap(arr, ++less, L++);   //交换小于区的下一个数++less和当前
```

```
数L++
                }
                else if (arr[L] > arr[R])    //当前数 > 划分值
                {
                    swap(arr, --more, L); //交换大于区的前一个数mroe--和当前数
L，L原地不动，因为换过来的数是新的
                }
                else
                {
                    L++;    //等于，L++
                }
            }
            swap(arr, more, R);   //交换大于区的第一个数和最后一个数R
            int p[] = { less + 1,more };     //返回等于区域，less+1和more（==之前的
R）
            return p;
        }

        void swap(int* arr, int i, int j)
        {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
};
```

# 3 详解桶排序以及排序内容大总结

## 3.1 完全二叉树

完全二叉树：满的或者子树按顺序。

### 3.1.1 数组与完全二叉树

可以把数组**从 0 出发的连续一段**想象成完全二叉树。数组与树位置的对应关系为：i 的左孩子为 2*i+1、右孩子为 2*i+2、父为(i-1)/2。

### 3.1.2 堆

堆在逻辑上是完全二叉树。分大根堆和小根堆。

以它为头的整棵树，每个子树的最大值是头节点的值——大根堆

### 3.1.3 数组与堆

heapsize 从 0 开始。每传来一个数，就放在 heapsize++的位置上，然后与父 PK（通过下标计算公式），一直 PK 上去。此过程叫做 heapinsert。

```
void heapInsert(int* arr, int index)
{
    while (arr[index] > arr[(index - 1) / 2]) //当前数比父大，且还未到头节点
    {
        swap(arr, index, (index - 1) / 2);
        index = (index - 1) / 2;
    }
}
```

现在让你找出最大的数——头节点，然后把它拿出去，剩下的如何保持大根堆形状？

先把最后一个数放在 0 位置上，heapsize 减小（最后一个数不释放也可以，已经在无效区里了）。从头节点开始，左孩子右孩子中，找个最大值与它 PK，如果孩子赢了，把大孩子拿上来；一直向下换下去，直到两个孩子都不比它大，或没有孩子了，停。此过程叫做 heapify 堆化。

```
//从index往下走，进行堆化，heapsize用于管理堆的大小——边界
```

```
void heapify(int* arr, int index, int heapSize)
{
    int left = index * 2 + 1;  //左孩子的下标
    while (left < heapSize)    //左孩子没越界==下方还有孩子的时候
    {
        //右孩子存在且右孩子值大，则右孩子胜出；其余情况左孩子胜出
        int largest = left + 1 < heapSize && arr[left + 1] > arr[left] ?
left + 1 : left;
        //父和较大孩子之间，谁的值大，把下标给largest
        largest = arr[largest] > arr[index] ? largest : index;
        if (largest == index)
        {
            break;  //父节点是最大的，不用往下走了
        }
        //如果父节点不是最大的，执行下面三行
        swap(arr, largest, index); //把子孩子中较大的与父交换
        index = largest;   //把index往下走
        left = index * 2 + 1;  //更新left
    }
}
```

如果现在用户抽风，随便挑一个 i 位置把它的数换成 a，如何保持大根堆？

如果 a 变小了，往下进行一个 heapify；如果 a 变大了，往上进行一个 heapinsert。两个只会中一个，调整完肯定对。——先 heapinsert，如果不动，就 heapify。

调整代价 O(logN)。

### 3.2 堆排序 时 O(NlogN) 空 O(1)

#### 3.2.1 代码

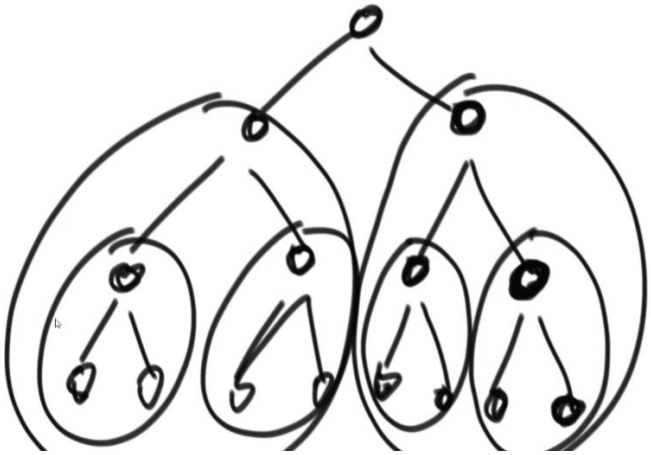
```
void heapSort(int* arr, int len)
{
    if (arr == NULL || len < 2)
    {
        return;
    }
    for (int i = 0; i < len; i++)  //整体变成大根堆
    {
        heapInsert(arr, i);
    }
    int heapSize = len;
    swap(arr, 0, --heapSize);  //交换[0] [--heapSize]
```

```
    while (heapSize > 0)
    {
        heapify(arr, 0, heapSize); //[0]往下heapify
        swap(arr, 0, --heapSize);  //交换[0] [--heapSize]
    }
}
```

### 3.2.2 优化 时 O(N)

把一个数组变成大根堆。

从最右下开始，从右到左，从下到上的顺序，heapify。



```
for (int i = len - 1; i >= 0; i--)
{
    heapify(arr, i, len);
}
```

注：堆结构比堆排序更重要；优先级队列结构就是堆结构。

### 3.2.3 拓展题

已知一个几乎有序的数组，几乎有序是指，如果把数组排好顺序的话，每个元素移动的距离可以不超过 k，并且 k 相对于数组来说比较小。请选择一个合适的排序算法针对这个数据进行排序。


k=6，小根堆，遍历前 7 个数，放到小根堆里，[0]一定是 min，弹出[0]；把[7]放到小根堆里，弹出[1]，……，最后临近结束时，依次弹出。

N 个数需要扩容 logN 次，均摊之后 O(N*logN)/N = O(logN)。

注：系统提供的堆结构，只支持你给它一个数，它给你一个数。不支持已经形成的东西重新调整成堆结构。

```
void SortArrayDistanceLessK(int* arr, int k,int len)
{
    priority_queue<int, vector<int>, greater<int>> heap;
    int index = 0;
    for (; index <= min(len, k); index++) //先创建一个小根堆
    {
        heap.push(arr[index]);
    }
    int i = 0;
    for (; index < len; i++, index++)
    {
        heap.push(arr[index]); //加入一个数
        arr[i] = heap.top();   //弹出顶
        heap.pop();
    }
    while (!heap.empty())  //还剩几个数的时候
    {
        arr[i++] = heap.top(); //依次弹出
        heap.pop();
    }
}
```

### 3.3 比较器

比较器==谓词：返回 bool 类型的仿函数。

```
class Person
{
```

```
public:
    Person(int name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }

    string m_Name;
    int m_Age;
};

class MyCompare
{
public:
    bool operator()(Person& p1, Person& p2)
    {
        return p1.m_Age > p2.m_Age;
    }
 };


void test()
{
    vector<Person>v;
    Person p1("Tom", 18);
    Person p2("Jerry", 20);
    v.push_back(p1);
    v.push_back(p2);
    sort(v.begin(), v.end(), MyCompare());
}
```

### 3.4 不基于比较的排序——计数排序

待排序数组 int []年龄，准备一个长度 201 的辅助数组，遍历一次原数组，遇到 0，就在辅助数组中 0 位置上++，遍历结束后得到一个词频数组，再将词频数组还原成正常数组。

时间复杂度 O(N)。用途较窄，只能根据数据状况定制。

### 3.5 基数排序

[17, 13, 25, 100, 72]。先看最大数字有几位，3 位，把不到 3 位的左边补 0。[017, 013, 025, 100, 072]。准备 10 个桶（10 进制，这里是队列）。

遍历，按个位数字入桶，再从左往右依次倒出来，[100, 072, 013, 025, 017]。

遍历，按十位数字入桶，再从左往右依次倒出来，[100, 013, 017, 025, 072]。

遍历，按百位数字入桶，再从左往右依次倒出来，[013, 017, 025, 072, 100]。

每一位的优先级都保留下来了，每一位的相对位置就排好了。

#### 3.5.1 代码思路

没有桶，用一个辅助数组 Count 充当词频表（[2]代表该位是 2 的数字有几个）。获得词频表后，将其累加，变成前缀和（[2]代表该位≤2 的数字有几个）。Count 数组已经把原数组划分区域了，接下来从右往左遍历原数组，将原数组中的数填入 help 数组对应区域的最右位置，更新 Count[i]。

#### 3.5.2 代码

```
class RadixSort //基数排序
{
public:
    void radixSort(int* arr, int len)
    {
        if (arr == NULL || len < 2)
        {
            return;
        }
        process(arr, 0, len - 1, maxbits(arr, len));
    }

    //最大值有多少位
    int maxbits(int* arr, int len)
    {
        int max_value = INT_MIN;
        for (int i = 0; i < len; i++)
        {
            max_value = max(max_value, arr[i]);
        }
```

```
        int res = 0;
        while (max_value != 0)
        {
            res++;
            max_value /= 10;
        }
        return res;
    }

    void process(int* arr, int L, int R, int digit)
    {
        const int radix = 10;
        int i = 0, j = 0;
        int* help = new int[R - L + 1];
        for (int d = 1; d <= digit; d++)  //有多少位就发生多少次入桶出桶
        {
            int* count = new int[radix];
            for (int k = 0; k < radix; k++)
            {
                count[k] = 0;
            }
            for (i = L; i <= R; i++)
            {
                j = getDigit(arr[i], d);   //取出对应位的数字
                count[j]++;    //词频数组更新
            }
            for (i = 1; i < radix; i++)        //将count处理成前缀和（累加和）
样子
            {
                count[i] += count[i - 1];
            }
            for (i = R; i >= L; i--)   //数组从右往左遍历
            {
                j = getDigit(arr[i], d);
                help[count[j] - 1] = arr[i];   //把这个数放到对应区域的最后
                count[j]--;    //更新对应区域的词频
            }
            for (i = L, j = 0; i <= R; i++, j++)
            {
                arr[i] = help[j];  //把出桶之后的数给原数组，再循环对下一位进行
操作
            }
        }
    }
```

```
    int getDigit(int x, int d)
    {

        return((x / ((int)pow(10, d - 1)) % 10));
    }

};
```

### 3.6 排序算法的稳定性及汇总

#### 3.6.1 稳定性概念

相同值的相对次序保持与原来一样

#### 3.6.2 排序算法汇总

| | 时间复杂度 | 额外空间复杂度 | 稳定性 |
|---|---|---|---|
| 选择排序 | $O(N^2)$ | O(1) | × |
| 冒泡排序 | $O(N^2)$ | O(1) | √ |
| 插入排序 | $O(N^2)$ | O(1) | √ |
| **归并排序** | **O(NlogN)** | **O(N)** | **√** |
| **快排（random）** | **O(NlogN)** | **O(logN)** | × |
| **堆排序** | **O(NlogN)** | **O(1)** | × |

首选快排，常数时间低；其次堆排；最后归并有稳定性。

Q：基于比较的排序，时间复杂度能否低于 O(NlogN)？

A：不行。

Q：时间复杂度 O(NlogN)，额外空间复杂度能否低于 O(N)且稳定？

A：不行。

#### 3.6.3 大坑题

归并排序的额外空间复杂度可以变为 O(1)，但会损失稳定性。

原地归并排序都是垃圾，时间复杂度变为 $O(N^2)$。

快速排序可以做到稳定，但额外空间复杂度变为 O(N)。

奇数放在数组左边，偶数放在数组右边，还要求原始的相对次序不变，时 O(N)，空 O(1)？经典快排的 partition 做不到稳定性，但是经典快排的 partition 又是 0、1 标准，和就奇偶问题是同一种调整策略，快排做不到，我也做不到。01stable sort

### 3.6.4 工程上对排序的改进

充分利用 O(NlogN)和 $O(N^2)$各自的优势，样本量小于 60 直接插入排序，否则快排。

稳定性的考虑：基础数据类型——快排；未知数据类型——归并排序。

# 4 链表

## 4.1 哈希表、有序表（详见 STL）

UnOrderedMap 或 UnSortedMap、OrderedMap 或 OrderedSet - > C++

Map 是 key-value；Set 是 key。

哈希表在使用时认为增删改查的时间是常数级别的。key 是基础数据直接 copykey；key 是自定义数据类型，key 占用的空间 8 字节（内存地址）。

有序表按 key 有序组织，性能略差，时间复杂度 O(logN)。放入有序表的东西，如果不是基础类型，必须提供比较器。

## 4.2 链表

### 4.2.1 水题

如何反转单向链表和反转双向链表。

换头操作需要让函数返回值类型为 Node。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev=NULL;
        ListNode* cur=head;
        while(cur)
        {
            ListNode* temp=cur->next;
            cur->next=prev;
            prev=cur;
            cur=temp;
```

```
        }
        return prev;
    }
};
```

递归的三个条件：大问题拆成两个子问题；子问题求解方式和大问题一样；存在最小子问题。

递：一直拆到一个元素->NULL；归：head.next.next=head; head.next=NULL;

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(head==NULL || head->next==NULL)
        {
            return head;
        }
        ListNode* newHead = reverseList(head->next);  //对除去头节点外的部分进行同样操作
        head->next->next = head;
        head->next = NULL;
        return newHead;
    }
};
```

### 4.2.2 打印两个有序链表的公共部分

给定两个有序链表的头指针 head1 和 head2，打印两个链表的公共部分。两个链表长度之和为 N，时间复杂度要求 O(N)，额外空间复杂度 O(1)。

谁小谁移动，相等打印且同时移动，有一个越界就停（while）。

### 4.2.3 面试时链表解题的方法论

笔试，不用太在乎空间复杂度，一切为了时间复杂度。

面试，时间复杂度依然放在第一位，但一定要找到空间最省的方法。

**重要技巧：**

额外数据结构记录（哈希表等）

快慢指针

### 4.2.4 判断一个单链表是否为回文结构

正着念和反着念一样，也可以理解为有一个对称轴，两边对称。

笔试：放到栈里，弹出的顺序就是逆序的顺序。从头遍历链表，同时出栈。

省一半空间：只把右边（快慢指针）的放入栈中，从头遍历链表，同时出栈，栈空了停，每一步都一样就是回文。

快慢指针：快指针每次走 2 步，慢指针每次走 1 步，快指针走完的时候，慢指针在中点位置。**要根据长度奇偶来定制快慢指针（coding 问题，边界问题）!**

时间复杂度 O(N)，额外空间复杂度 O(1)。

```
/**
 * 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) {
        if (head == nullptr) {
            return true;
        }

        // 找到前半部分链表的尾节点并反转后半部分链表
        ListNode* firstHalfEnd = endOfFirstHalf(head);
        ListNode* secondHalfStart = reverseList(firstHalfEnd->next);

        // 判断是否回文
        ListNode* p1 = head;
        ListNode* p2 = secondHalfStart;
        bool result = true;
        while (result && p2 != nullptr) {
            if (p1->val != p2->val) {
                result = false;
            }
            p1 = p1->next;
            p2 = p2->next;
```

```
        }

        // 还原链表并返回结果
        firstHalfEnd->next = reverseList(secondHalfStart);
        return result;
    }

    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr != nullptr) {
            ListNode* nextTemp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = nextTemp;
        }
        return prev;
    }

    ListNode* endOfFirstHalf(ListNode* head) {
        ListNode* fast = head;
        ListNode* slow = head;
        while (fast->next != nullptr && fast->next->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }
        return slow;
    }
};
```

### 4.2.5 单链表按某值划分为左小、中等、右大

笔试：Node 类型的数组，遍历把链表的每个节点放入数组中，partition，再变回链表。

面试：6 个指针，小于头、小于尾、等于头、等于尾、大于头、大于尾，初始化全指向空。遍历链表，把每一个节点加到对应区域的子链表中，更改头/尾指针的指向。最后将三个区域连接。注意：可能存在没有某区域的情况，因此重连时要讨论清楚边界。

```
#include<iostream>
using namespace std;
```

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode* next, ListNode* rand) : val(x), next(next) {}
};

class SmallerEqualBigger
{
public:
    ListNode* listPartition(ListNode* head, int pivot)
    {
        ListNode* sH = NULL;
        ListNode* sT = NULL;
        ListNode* eH = NULL;
        ListNode* eT = NULL;
        ListNode* bH = NULL;
        ListNode* bT = NULL;
        ListNode* next = NULL;

        while (head != NULL)
        {
            //把头（cur）断开
            next = head->next;
            head->next = NULL;
            if (head->val < pivot)
            {
                if (sH == NULL)
                {
                    sH = head;
                    sT = head;
                }
                else
                {
                    sT->next = head;
                    sT = head;
                }
            }
            else if (head->val == pivot)
            {
                if (eH == NULL)
                {
                    eH = head;
```

```
                eT = head;
            }
            else
            {
                eT->next = head;
                eT = head;
            }
        }
        else
        {
            if (bH == NULL)
            {
                bH = head;
                bT = head;
            }
            else
            {
                bT->next = head;
                bT = head;
            }
        }
        head = next;    //更新头节点，往下走
    }

    if (sT != NULL)//如果有小于区域
    {
        sT->next = eH; //小于区尾连等于区头
        //如果等于区有东西，那么eT就是eT，否则用小于区的尾连大于区的头，eT=sT
        //谁去连大于区的头，谁就是eT
        eT = eT == NULL ? sT : eT;
    }

    if (eT != NULL)//如果小于区和等于区不是都没有
    {
        eT->next = bH;
    }
    return sH != NULL ? sH : (eH != NULL ? eH : bH);
}

};
```

### 4.2.6 复制含有随即指针节点的链表

```
Class Node{
    int value;
    Node next;
    Node rand;
    Node(int val){
        value = val;
    }
}
```

rand 指针是单链表节点结构中新增的指针，rand 可能指向链表中的任意一个节点，也可能指向 null。给定一个由 Node 节点类型组成的无环单链表的头节点 head，请实现一个函数完成这个链表的赋值，并返回复制的新链表的头节点。要求时间复杂度 O(N)，额外空间复杂度 O(1)。

利用额外空间：

Hashmap。Key（老的节点），Value（老节点对应的克隆出来的新节点）。

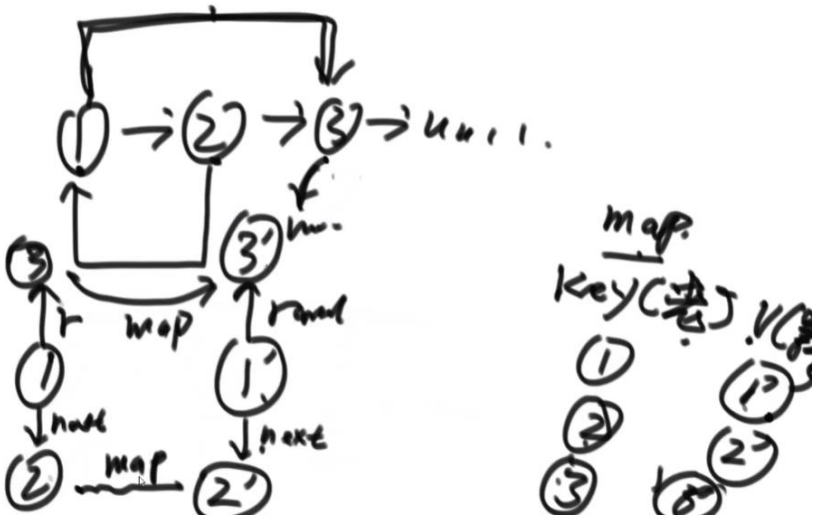
遍历链表，对每个节点进行克隆（new），放入 Hashmap 中。

遍历链表，对每一个老节点：

new_cur = map.get(cur); //取出这个老节点对应的 val（新节点）

new_cur->next=map.get(cur->next); //新节点的 next 等于老节点 next 的 val 值

new_cur->rand=map.get(cur->rand); //新节点的 rand 等于老节点 rand 的 val 值

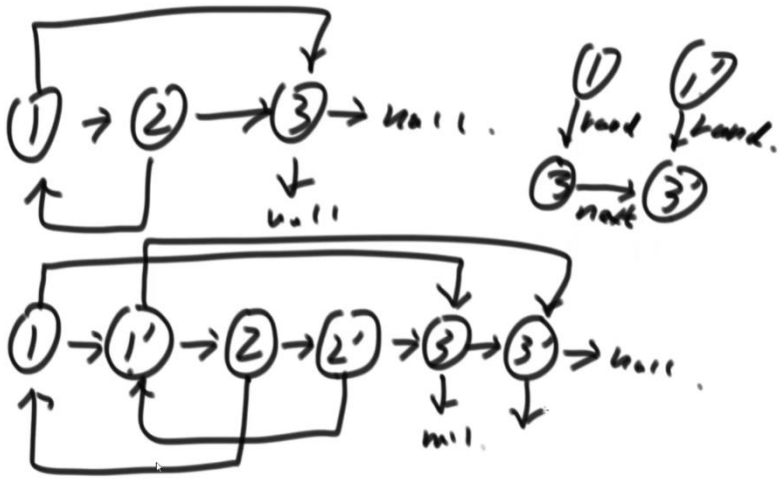


不使用额外空间：利用位置关系省去 map

遍历链表，每一个节点 copy 一个新节点，放在该节点的后面。

遍历链表，每个节点的 next 的 rand 等于该节点的 rand 的 next。

在 next 把新老链表分开。



```
#include<iostream>
#include<map>
using namespace std;

struct ListNode {
    int val;
    ListNode* next;
    ListNode* rand;
    ListNode() : val(0), next(nullptr), rand(nullptr) {}
    ListNode(int x) : val(x), next(nullptr), rand(nullptr) {}
```

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

class CopyListWithRandom
{
public:
    ListNode* copyListWithRand1(ListNode* head)
    {
        map<ListNode*, ListNode*>m;
        ListNode* cur = head;
        while (cur != NULL)
        {
            m.insert(make_pair(cur, new ListNode(cur->val)));
            cur = cur->next;
        }
        cur = head;
        while (cur != NULL)
        {
            //假设it->first = 1，it->second = 1'
            map<ListNode*, ListNode*>::iterator it = m.find(cur);
            //1'->next = m.find(1->next)->second
            it->second->next = m.find(it->first->next)->second;
            //1'->rand = m.find(1->rand)->second
            it->second->rand = m.find(it->first->rand)->second;
            cur = cur->next;
        }
        return m.find(head)->second;
    }

    ListNode* copyListWithRand2(ListNode* head)
    {
        if (head == NULL)
        {
            return NULL;
        }
        ListNode* cur = head;
        ListNode* temp = NULL;
        // 1->2 变成 1->1'->2
        while (cur)
        {
            temp = cur->next;
            cur->next = new ListNode(cur->val);
            cur->next->next = temp;
```

```
            cur = temp;
        }
        cur = head;
        ListNode* curCopy = NULL;
        while (cur)
        {
            temp = cur->next->next;
            curCopy = cur->next;
            curCopy->rand = cur->rand != NULL ? cur->rand->next : NULL;
            cur = temp;
        }
        ListNode* res = head->next;
        cur = head;
        while (cur)
        {
            temp = cur->next->next;
            curCopy = cur->next;
            curCopy->next = temp != NULL ? temp->next : NULL;
            cur = temp;
        }
        return res;
    }
};
```

### 4.3 两个单链表相交的一系列问题

给定两个可能有环也可能无环的单链表，头节点 head1 和 head2。请实现一个函数，如果两个链表相交，请返回相交的第一个节点。如果不相交，返回 null。

要求：如果两个链表长度之和为 N，时间复杂度 O(N)，额外空间复杂度 O(1)。

#### 4.3.1 判断一个链表是否有环

有环返回第一个入环节点，无环返回 false。

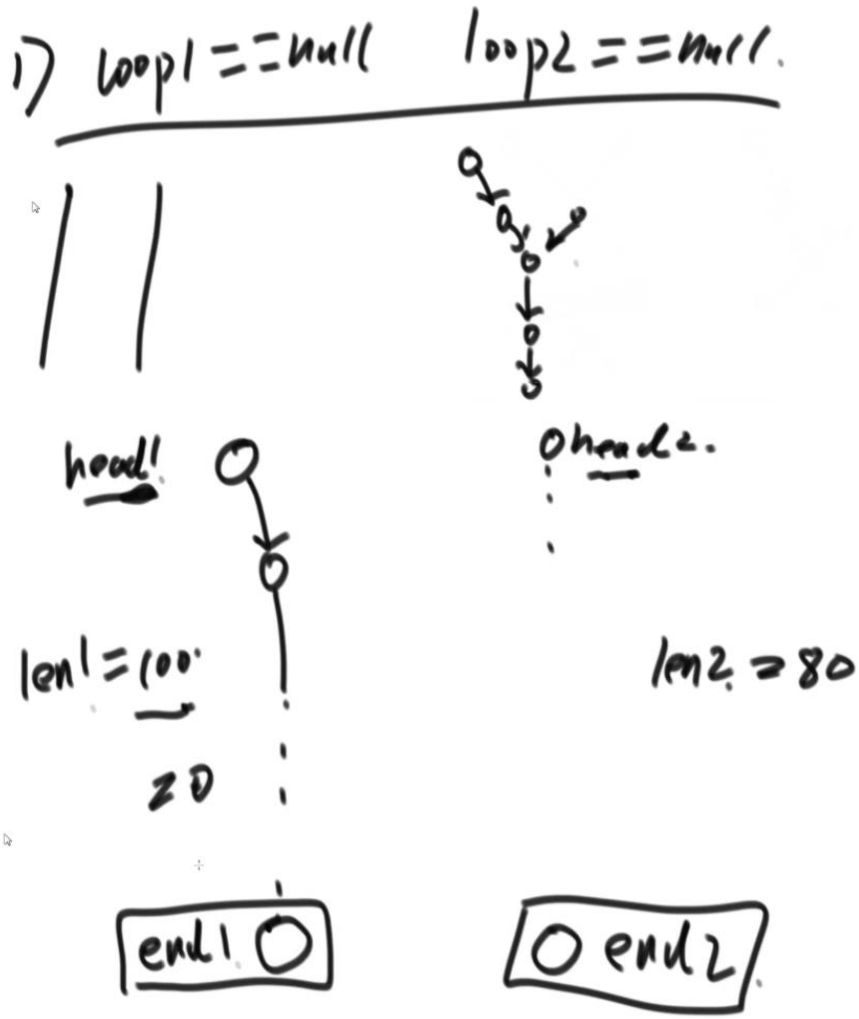
**方法一：**

Hashset，遍历下去，没遇过就存到 set 里，走到 null 则无环；如遇到在 set 里的了，就有环了。

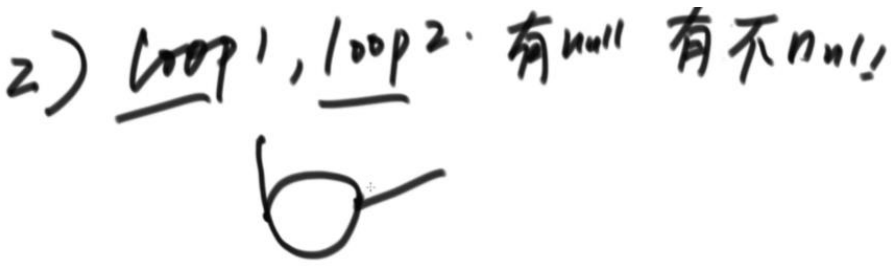
**方法二：快慢指针**

如果无环，快指针会走到空；如果有环，快慢指针会在环上相遇。相遇后快指针回到 head，接下来两个指针每个都只走 1 步，一定会在入环节点相遇。

### 4.3.2 判断是否相交

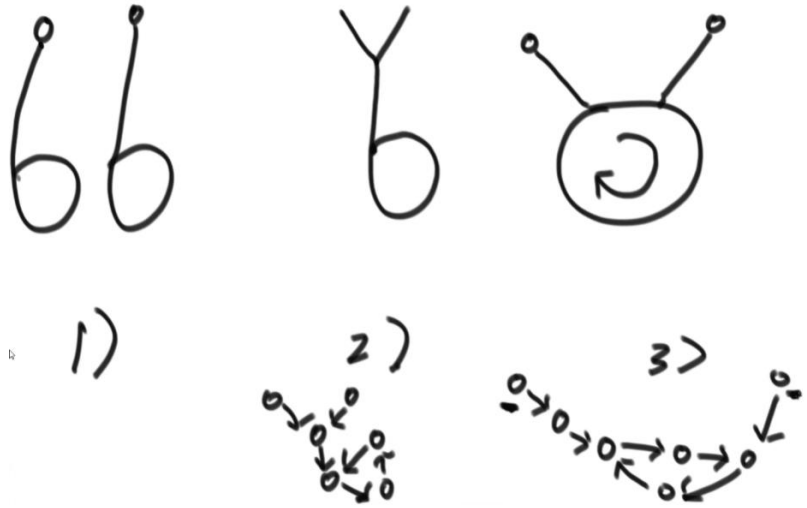



end1 == end2，长链表先走差值步，再两个链表一起走，在相交节点相遇




如果一个链表有环，一个链表无环，则这两个链表不可能相交。

(3) 两个链表都有环




情况 2：loop1 == loop2，然后把 loop 的位置当作终止节点，又变回了无环链表相交问题。

情况 1、3：loop1 继续往下走，loop2 不动，在转回到自己之前能遇到 loop2，就是情况 3；没遇到 loop2，就是情况 1。

如果是情况 1，返回 null。

如果是情况 3，返回 loop1 或 loop2 都对。

```
#include<iostream>
#include<cmath>

struct ListNode
{
    int val;
    ListNode* next;
};

class FindFirstIntersectNode
{
public:
    ListNode* getIntersectNode(ListNode* head1, ListNode* head2)
    {
        if (head1 == NULL || head2 == NULL)
        {
            return NULL;
        }
        ListNode* loop1 = getLoopNode(head1);
        ListNode* loop2 = getLoopNode(head2);
        if (loop1 == NULL && loop2 == NULL)
        {
```

```
        return noLoop(head1, head2);
    }
    if (loop1 != NULL && loop2 != NULL)
    {
        return bothLoop(head1, loop1, head2, loop2);
    }
    return NULL;
}

//找到链表第一个入环节点，如果无环，返回NULL
ListNode* getLoopNode(ListNode* head)
{
    if (head == NULL || head->next == NULL || head->next->next == NULL)
    {
        return NULL;
    }
    ListNode* slow = head->next;
    ListNode* fast = head->next->next;
    while (slow != fast)
    {
        if (fast->next == NULL || fast->next->next == NULL)
        {
            return NULL;
        }
        slow = slow->next;
        fast = fast->next;
    }
    fast = head;    //从头节点重新走
    while (slow != fast)
    {
        slow = slow->next;
        fast = fast->next;
    }
    return slow;
}

ListNode* noLoop(ListNode* head1, ListNode* head2)
{
    if (head1 == NULL && head2 == NULL)
    {
        return NULL;
    }
    ListNode* cur1 = head1;
    ListNode* cur2 = head2;
```

```
        int n = 0;
        while (cur1->next)
        {
            n++;
            cur1 = cur1->next;
        }//结束后cur1就是end1
        while (cur2->next)
        {
            n--;
            cur2 = cur2->next;
        }
        if (cur1 != cur2)
        {
            return NULL;
        }
        cur1 = n > 0 ? head1 : head2;  //长的头节点变成cur1
        cur2 = cur1 == head1 ? head2 : head1; //短的头节点变成cur2
        n = abs(n);
        while (n)
        {
            n--;
            cur1 = cur1->next;
        }
        while (cur1 != cur2)
        {
            cur1 = cur1->next;
            cur2 = cur2->next;
        }
        return cur1;
    }

    ListNode* bothLoop(ListNode* head1, ListNode* loop1, ListNode* head2,
ListNode* loop2)
    {
        ListNode* cur1 = NULL;
        ListNode* cur2 = NULL;
        if (loop1 == loop2)
        {
            cur1 = head1;
            cur2 = head2;
            int n = 0;
            while (cur1 != loop1)
            {
                n++;
```

```
                cur1 = cur1->next;
            }
            while (cur2 != loop2)
            {
                n--;
                cur2 = cur2->next;
            }
            cur1 = n > 0 ? head1 : head2;
            cur2 = cur1 == head1 ? head2 : head1;
            n = abs(n);
            while (n != 0)
            {
                n--;
                cur1 = cur1->next;
            }
            while (cur1 != cur2)
            {
                cur1 = cur1->next;
                cur2 = cur2->next;
            }
            return cur1;
        }
        else
        {
            cur1 = loop1->next;
            while (cur1 != loop1)
            {
                if (cur1 == loop2)
                {
                    return loop1;
                }
                cur1 = cur1->next;
            }
            return NULL;
        }

    }
};
```