分治——求逆序对的个数

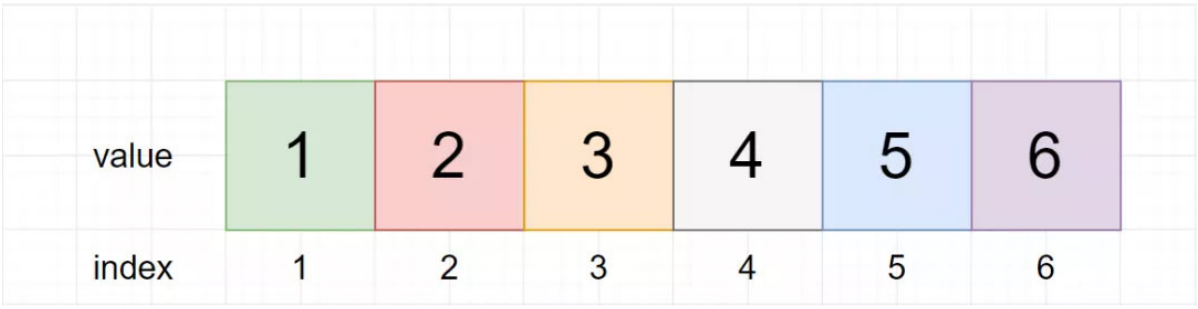
原创 田郑书媛 LOA算法学习笔记 2021-03-04 20:05

什么是逆序对呢？在数组中的两个数字，如果前面一个数字大于后面一个数字，那么这两个数字便构成了一个逆序对。逆序对反映了一个数组的有序程度，如下面图1~3所示。

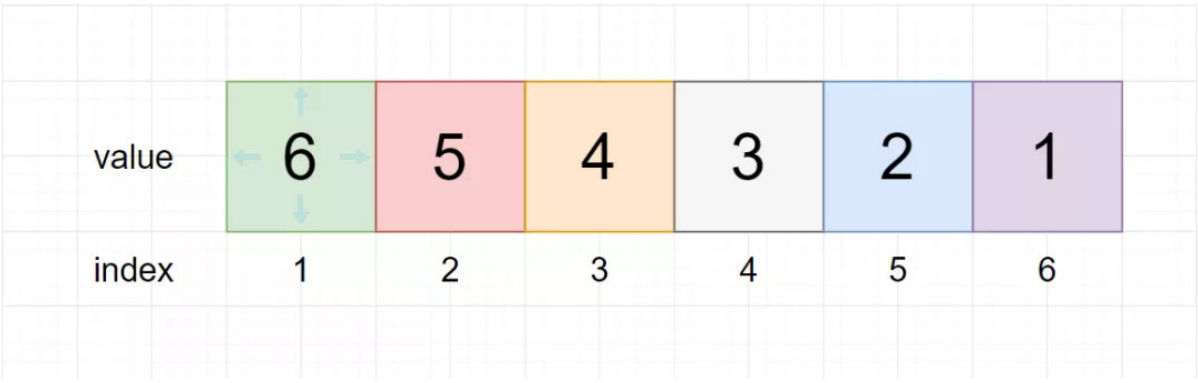
图1为顺序数组，其中逆序对的个数为0。

图2为逆序数组，其中逆序对一共有5+4+3+2+1=15个。

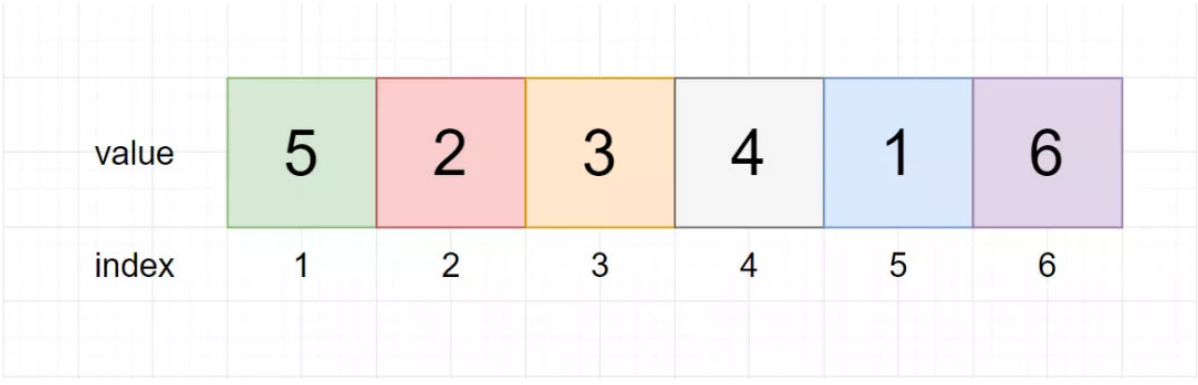
图3中的逆序对有(5,2),(5,3),(5,4),(5,1),(2,1),(3,1),(4,1)，一共是7个。



（图1:顺序数组不含逆序对）



（图2：逆序数组，任意两个元素构成逆序对）



（图3：逆序对7个）

下面介绍三种求解数组中逆序对的方法。

方法一：暴力求解法

1. 思路

每两个元素之间都要进行比较，且不进行任何存储记录。假设数组中一共有n个元素，则index为0的元素要和后面的n-1个数进行比较， index为1的元素要和后面的n-2个数进行比较……index为n-2的数要和后面的1个数进行比较，如果比后面的数大则逆序对的数量加一。

2. C语言代码

```
1 #include
2 #include
```

```c
#define MAX_NUM 100
char input[MAX_NUM*2];
int array[MAX_NUM];
int init(){
    int i,j=0;
    for(i=0;i<MAX_NUM;i++){
        array[i]=0;
    }
    printf("Please input the array\n");
    scanf("%s",&input);
    for(i=0;(i<MAX_NUM*2) && (input[i]!=0);i++){
        if(input[i]==','){
            j++;
        }else{
            array[j]=10*array[j]+input[i]-'0';
        }
    }
    return j+1;
}


int count_inversion(int array_num){
    int i,j;
    int count=0;
    for(i=0;i<array_num;i++){
        for(j=i+1;j<array_num;j++){
            if(array[i]>array[j])
                count++;
        }
    }
    return count;
}


int main(){
    int inversion_num,array_num;


    inversion_num=count_inversion(array_num);
    printf("There are totally %d inversions in this array\n",inversion_num);
    return 0;
}
```

上面是用暴力解法求逆序对的C语言代码，对输入格式的要求是每个元素用逗号隔开。init函数根据用户输入构造整数数组，count_inversion函数用暴力解法求数组中的逆序对个数。

3. 复杂度

设数组一共有n个元素，则一共需要比较的次数为$(n-1)+(n-2)+\cdots\cdots+1=n(n-1)/2=(n^2-n)/2$次，时间复杂度为$O(n^2)$

方法二：归并排序法

1. 思路

暴力求解法的思路很简单，但对应的时间复杂度也较高，我们希望通过分而治之的方法，降低求解逆序对的时间复杂度。那么对这个问题的分解与合并要如何实现呢？

(1) 分：将数组array平均分为前后两部分，前半部分为array[0...[n/2]-1],后半部分为array[[n/2]...n-1],分别计算前半部分的逆序对个数（即逆序的两个元素都在前半部分）和后半部分的逆序对个数（即逆序的两个元素都在后半部分）

(2) 合：计算一个元素在前半部分，另一个元素在后半部分的逆序对个数。

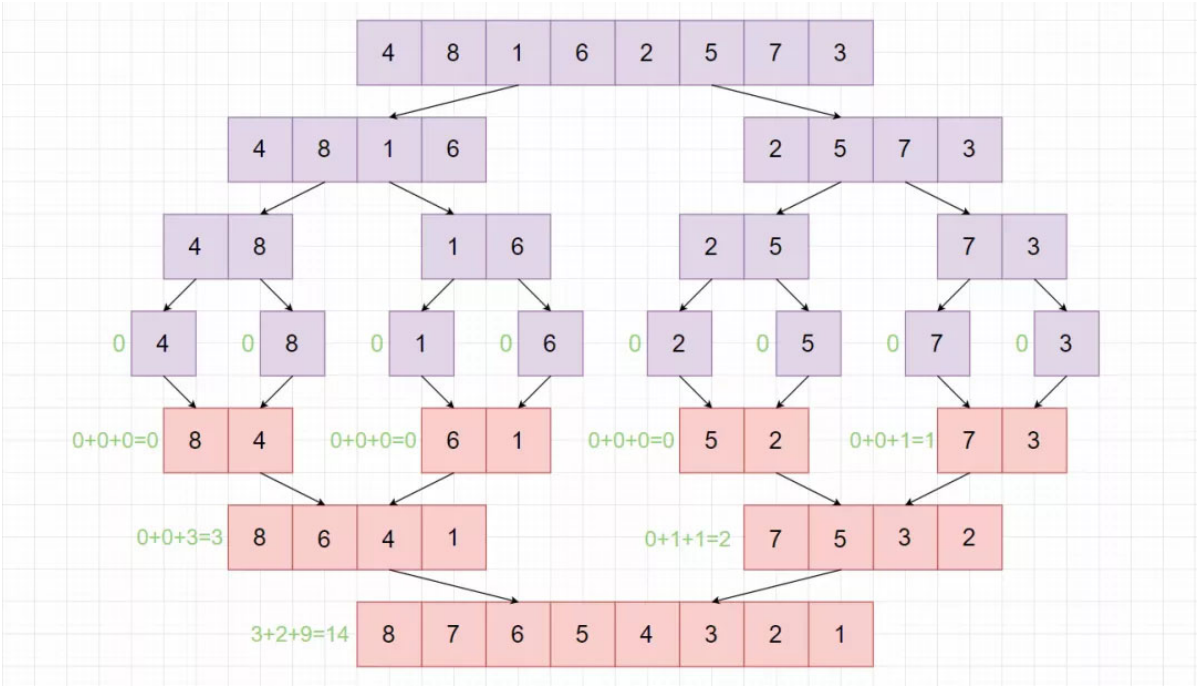
但是，问题到这里并没有结束，我们需要考虑"合"的两种情况：

① 前半部分和后半部分并没有一定的顺序结构：

在这种情况下，前半部分的每一个元素都要和后半部分的每一个元素进行大小比较，前半部分有n/2个元素，后半部分也有n/2个元素，一共需要比较 $n/2 \times n/2 = n^2/4$ 次，得到递推式 $T(n) = 2T(n/2) + n^2/4$ ，可以算出 $T(n) = O(n^2)$,并没有达到我们预期的降低时间复杂度的效果。

② 前半部分和后半部分有顺序结构：

如果前半部分和后半部分都是递减的，那问题就好办了。由于前半部分的索引值一定小于后半部分的索引值，所以只要我们发现了前半部分的一个元素A大于后半部分的一个元素B，那么A和B就一定可以构成逆序对。对于前半部分的元素A，当我们在后半部分扫描发现了第一个小于它的元素B时，就可以停止向后扫描了，因为B后面的元素都小于B，所以也一定小于A，即B和它后面的所有元素都可以和A构成逆序对。通过这个方法，我们可以大大较少比较的次数，在合并时对前半部分和后半部分只扫描一遍，从而将时间复杂度降到O(n logn)



(图4)

图4所示是归并排序求逆序对个数的过程，紫色部分为"分"，粉色部分是"合"，旁边用绿色展示计算当前逆序对个数的过程。

2. C语言代码

```
#include
#include
#define MAX_NUM 100
char input[MAX_NUM*2];
int array[MAX_NUM];
int init(){
    int i,j=0;
    for(i=0;i<MAX_NUM;i++){
        array[i]=0;
    }
    printf("Please input the array\n");
```

```
    scanf("%s",&input);
    for(i=0;(i<MAX_NUM*2) && (input[i]!=0);i++){
        if(input[i]==','){
            j++;
        }else{
            array[j]=10*array[j]+input[i]-'0';
        }
    }
    return j+1;
}


int merge(int left,int middle,int right){
    int p1=left,p2=middle+1,result=0,i;
    int temp[MAX_NUM];
    //初始化temp数组
    for(i=0;i<MAX_NUM;i++)
        temp[i]=0;
    //下面的while循环找到前半部分和后半部分的元素构成的逆序对
    while(p1<=middle && p2<=right){
        if(array[p1]>array[p2]){
            result=result+right-p2+1;
            p1++;
        }else{
            p2++;
        }
    }
    //下面将前半部分和后半部分合并为一个递减的数组temp
    p1=left;p2=middle+1;i=0;
    while(p1<=middle && p2<=right){
        if(array[p1]>array[p2])
            temp[i++]=array[p1++];
        else
            temp[i++]=array[p2++];
    }
    while(p1<=middle)
        temp[i++]=array[p1++];
    while(p2<=right)
        temp[i++]=array[p2++];
    //将temp中的内容拷贝到array里
    for(i=0;i<right-left+1;i++)
        array[left+i]=temp[i];
    return result;
}


int sort_count(int left,int right){
    int middle=(right+left)/2;
    int result=0;
    if(left==right){
```

```
        result=0;
    }else{
        result+=sort_count(left,middle);     //递归求前半部分的逆序对
        result+=sort_count(middle+1,right);  //递归求后半部分的逆序对
        result+=merge(left,middle,right);
    }
    return result;
}


int main(){
    int inversion_num,array_num;
    array_num=init();
    inversion_num=sort_count(0,array_num-1);
    printf("There are totally %d inversions in this array\n",inversion_num);
    return 0;
}
```

上面是用归并排序求逆序对的C语言代码，对输入格式的要求是每个元素用逗号隔开。

init函数根据用户输入构造整数数组。sort_count函数递归求解数组中的逆序对个数。将数组平均分为两半，首先调用sort_count递归求解前半部分内部的逆序对数量和后半部分内部的逆序对数量，然后调用merge函数。在merge函数中要做两件事：

① 寻找前半部分和后半部分之间的逆序对：

为前半部分设置一个指针p1，为后半部分设置一个指针p2，初始时p1和p2分别位于前半部分和后半部分的最左边。首先p1保持不动，p2向后扫描，当p2扫描到第一个比p1大的元素时，p2停止扫描，由于后半部分是递减的，故此时p2所指的元素及其后的所有元素均和p1所指元素构成逆序对，result要加上p2所指的元素及其后的所有元素的数量。然后将p1向后移动一个元素，p2从之前停下的位置开始扫描，这是因为p1此时所指的元素一定变小了，所以后半部分里比它小的第一个元素一定是在之前p2停下的地方后面，所以p2可以不用调整，直接从之前停下来的位置开始向后移动。不断重复这个过程，直到p1完成了对前半部分的扫描或p2完成了对后半部分的扫描。

② 对整个数组排序，使之按从大到小的顺序排列，从而方便之后对逆序对的寻找：

由于前半部分和后半部分本身已经有序，故可以先将p1和p2移到前半部分和后半部分最左边，然后每次选p1和p2所指的元素里面较大的放在数组temp的最后，将指向被选中元素的指针后移，然后继续比较。

3. 复杂度

因为merge函数中要做的两件事的复杂度均为O(n)，所以时间复杂度的递推式可以写为T(n)=2T(n/2)+O(n)，解得T(n)=O(n logn)

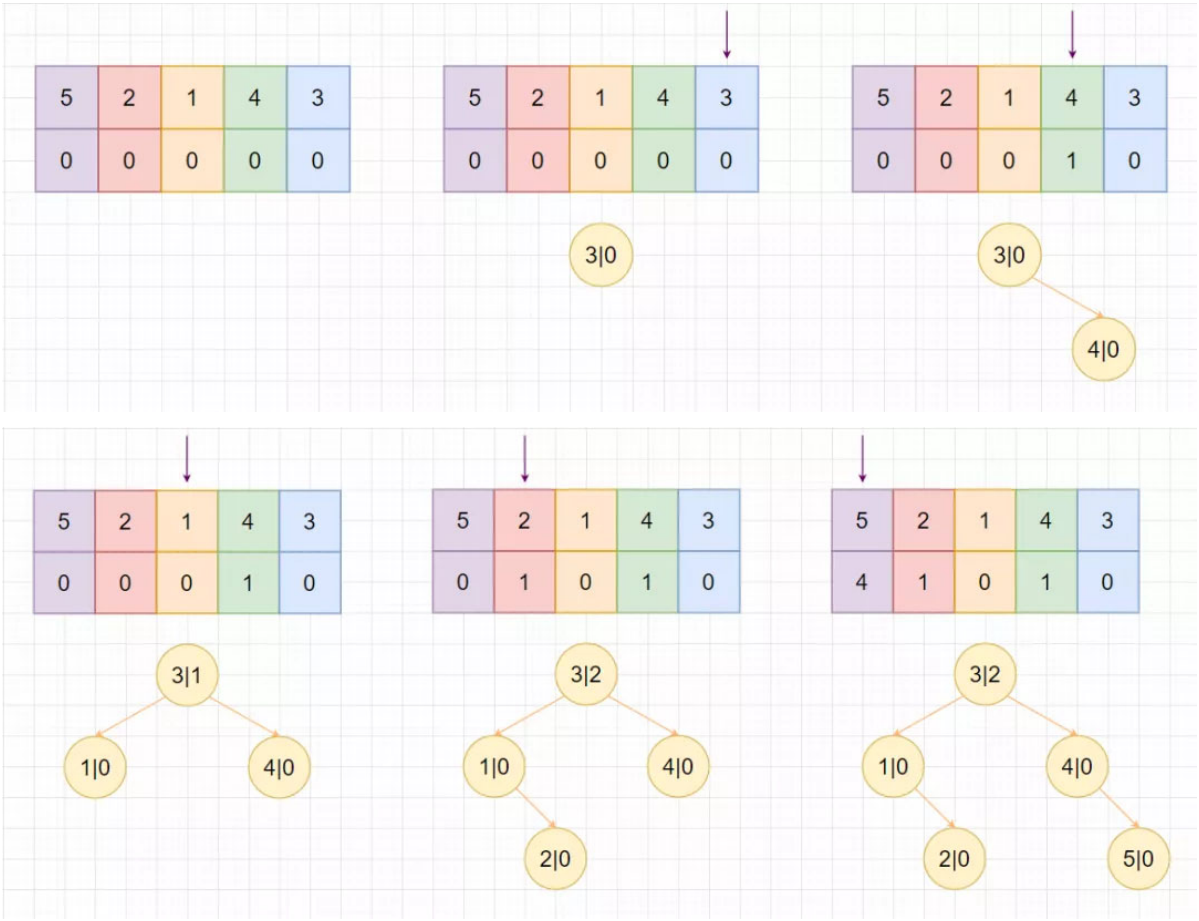
方法三：二叉搜索树

1. 思路

按照从后向前的顺序构造二叉搜索树，这样每当要新插入一个元素到二叉搜索树里的时候，都可以保证二叉查找树中当前存在的所有元素在数组中都位于新插入元素的后面。二叉搜索树的每个结点中除了有该结点的值value、指向左、右子树的指针外，还要添加一个用于记录左子树的元素总数的域subs，即当前树中数值小于该结点的元素总数。

当要给某个结点的左子树添加元素时，该结点的subs域的值要增加1。

当要给某个结点的右子树添加元素时，意味着当前结点和结点左子树中的所有结点都是小于新添元素的结点，并且在数组中位于新添元素的右侧，故都可以和新添元素构成逆序对。在递归过程中用一个专门的计数数组记录对应每个元素的逆序对，最后将每个元素对应的逆序对加起来即可得到数组中含有的逆序对总数。



（图5）

图5所示是从后向前构造数组{5，2，1，4，3}对应的二叉搜索树的过程，矩形内是数组中每个元素对应的逆序对个数，二叉树的每个结点里左侧的数字是该结点的值，右侧的数字是这个结点的左子树结点个数。

2. C语言代码

```
#include
#include
#define MAX_NUM 100
char input[MAX_NUM*2];
int array[MAX_NUM];
struct node{
    int val;
    int subs;
    struct node *left;
    struct node *right;
};


int init(){
    int i,j=0;
    for(i=0;i<MAX_NUM;i++){
        array[i]=0;
    }
    printf("Please input the array\n");
    scanf("%s",&input);
    for(i=0;(i<MAX_NUM*2) && (input[i]!=0);i++){
        if(input[i]==','){
            j++;
```

```
        }else{
            array[j]=10*array[j]+input[i]-'0';
        }
    }
    return j+1;
}


struct node *getTreeNode(int val){
    struct node *ret = (struct node *)malloc(sizeof(struct node));
    ret->left = NULL;
    ret->right = NULL;
    ret->val = val;
    ret->subs = 0;
    return ret;
}


struct node *insert(struct node *dest, int src, int *depth){
    if (dest == NULL){
        struct node *ret = getTreeNode(src);
        return ret;
    }
    if (src > dest->val){
        *depth += 1 + dest->subs;
        dest->right = insert(dest->right, src, depth);
    }
    if (src val){
        dest->subs+=1;
        dest->left = insert(dest->left, src, depth);
    }
    return dest;
}


int count_inversion(int *array, int array_num){
    int ret[array_num];
    int i = 0,result=0;
    for(i=0;i<array_num;i++)
        ret[i]=0;
    if (array_num <= 1)
        return 0;
    struct node *tree = NULL;
    for (i = array_num - 1; i >= 0; i--)
        tree = insert(tree, array[i], ret + i);
    for(i=0;i<array_num;i++)
        result+=ret[i];
    return result;
}
```

```
74
75 int main(){
76     int inversion_num,array_num;
77     array_num=init();
78     inversion_num=count_inversion(array,array_num);
79     printf("There are totally %d inversions in this array\n",inversion_num);
80     return 0;
81 }
```

上面是用二叉搜索树求逆序对的C语言代码，对输入格式的要求是每个元素用逗号隔开。

init函数根据用户输入构造整数数组。count_inversion函数按照从后往前的顺序将数组中的每一个元素插入到二叉搜索树里面，同时计算该元素对应的逆序对个数(即由于该元素大于数组中位于它后面的元素而产生的逆序对)，而这两步都是通过insert函数实现的。在insert函数中，如果输入的dest为空，则调用getTreeNode函数新建一个结点，否则继续考虑要插入的元素值的大小：

如果元素值大于dest结点的值，则将dest和它的左子树总的结点数加到新添加元素对应的逆序对个数里，并对dest的右子树递归调用insert函数；

如果元素值小于dest结点的值，则将dest的subs域加一，表示dest又多了一个左孩子，然后对dest的左子树递归调用insert函数。

3. 复杂度

统计逆序对数目的操作与插入结点的操作同步进行，一共要向二叉搜索树里面插入n个结点。如果最后构造的二叉树比较平衡，那么树的高度大约是logn，插入每个结点大约需要logn步，时间复杂度为O(nlogn)，但是如果构造的二叉搜索树很不平衡，那这个树可能会达到n的高度，这样的话时间复杂度就是 $O(n^2)$ 了，所以这个方法最好的时间复杂度是O(nlogn)，最差是 $O(n^2)$ 。

总结一下本文介绍的三种方法：暴力求解法、归并排序法和二叉搜索树解法。其中暴力求解法思路简单，但是效率也最低，只能达到 $O(n^2)$ 的时间复杂度。归并排序法用分治的思想，将大问题拆解为更易求解小问题，并且通过使数组有序实现小问题“合”为大问题时的加速，时间复杂度提升为O(nlogn)，而二叉搜索树法将统计逆序对数目的操作与插入结点的操作同步进行，通过“二叉搜索”的思想加速，在平衡的情况下时间复杂度最低，可以达到O(nlogn)，但是如果二叉树很不平衡的话，时间复杂度和暴力求解法相当。