数组与字符串：排序与求余

最简单的数据结构要数数组了，数组是由一组连续内存模块构成的。对于给定的一个数组 A, 它含有 n 个元素，A[i]表示存在于数组中的第 i 个元素。获取或更新元素 A[i]所需要的时间是 O(1),但由于数组的空间是固定的，因此要想添加新的元素，就比较麻烦。要实现对第 i 个元素的删除，我们可以再使用一个对应的 boolean 型数组，该数组的第 i 个元素是 false, 表明 A[i]已经被删除。

对数组来说，插入元素比较麻烦，一般来说，要向后插入一个元素，我们需要重新分配一块比原数组大的内存，然后将原数组的元素以及新元素拷贝到新分配的内存中。这种插入的方法看似非常耗时，但如果我们做一下改动，使得每次分配的内存是原数组的两倍，那么如果插入操作即使很频繁，但由于一次分配的内存够大，插入时不需要每次都发生元素的拷贝动作，因此算下来，平均的插入时间可以是 O(1).

接下来我们看看两道关于数组操作的面试算法题。
对数组的排序及相关操作，始终是算法面试的一大考点，下面这道题就是数组排序操作的一个变种，而且也是面试中出现较为频繁的考题。

题目是这样的：写一个函数，输入是一个数组 A，以及下标 i,要求函数将数组的元素进行调整，使得所有比 A[i]小的元素排在前面，接着是所有等于 A[i]的元素，最后排列的是所有大于 A[i]的元素。例如给定 A = 6 ,5 ,5, 7, 9,4, 3, 3, 4, 6, 8, 4,7,9, 2, 1. i = 5. 于是得到调整后的数组为：1,2,3,3,4,4,4,6,8,7,7,9,5,5,6,9

要求算法的空间复杂度是 O(1),时间复杂度是 O(n).

这个问题的解决,我们先用一个变量 pivot 来存储 A[i]的值，然后分两步走，第一步是将数组分成两部分，开头那部分所有元素都小于 pivot,第二部分所有元素都大于等于 pivot.

接着对数组第二部分，我们再做一次调整，使得数组第二部分又分成两部分,第一部分所有元素等于 pivot,第二部分所有元素大于 pivot.

经过这两部调整后，所得的结果符合题目条件。
我们先看看第一步怎么做，我们用两个指针，begin 和 end, begin 指向队列开头，end 指向队列末尾，如果 A[begin] >= pivot. 那么将 A[begin]与 A[end]调换，然后 end--, 如果 A[begin] < pivot, 那么 begin++, 当 begin >= end 时，

循环结束，由于当 A[begin] > pivot 时，我们都把 A[begin] 的值放到 A[end]上，于是循环结束后，我们能保证对任意 j，j >= end，都有 A[j] >= pivot，这样我们第一步就实现了。我们用题目中的例子来走一遍上面的算法，begin = 0，end = 15，绿色的数字表示 A[begin]，红色表示 A[end]，pivot = A[5] = 4;

6 ,5 ,5, 7, 9,4, 3, 3, 4, 6, 8, 4,7,9, 2, 1

由于 A[begin] > pivot，于是 A[begin]和 A[end]互换，end--:

1 ,5 ,5, 7, 9,4, 3, 3, 4, 6, 8, 4,7,9, 2, 6

由于 A[being] < pivot，于是 begin++:

1 ,5 ,5, 7, 9,4, 3, 3, 4, 6, 8, 4,7,9, 2, 6

由于 A[begin] > pivot，于是 A[begin]和 A[end]互换，end--:

1 ,2 ,5, 7, 9,4, 3, 3, 4, 6, 8, 4,7,9, 5, 6

由于 A[being] < pivot, 于是 begin++:

1 ,2 ,5, 7, 9,4, 3, 3, 4, 6, 8, 4,7,9, 5, 6

由于 A[begin] > pivot, 于是 A[begin]和 A[end]互换，end--:

1 ,2 ,9, 7, 9,4, 3, 3, 4, 6, 8, 4,7,5, 5, 6

由于 A[begin] > pivot, 于是 A[begin]和 A[end]互换，end--:

1 ,2 ,7, 7, 9,4, 3, 3, 4, 6, 8, 4,9,5, 5, 6

由于A[begin]>pivot, 于是A[begin]和A[end]互换，end—

1 ,2 ,4, 7, 9,4, 3, 3, 4, 6, 8, 7,9,5, 5, 6

由于 A[begin] >= pivot, 于是 A[begin]和 A[end]互换，end—

1 ,2 ,8, 7, 9,4, 3, 3, 4, 6, 4, 7,9,5, 5, 6

由于A[begin]＞pivot，于是A[begin]和A[end]互换，end—

1 ，2 ，6， 7， 9，4， 3， 3， 4， 8， 4， 7，9，5， 5， 6

由于A[begin]＞pivot，于是A[begin]和A[end]互换，end—

1 ，2 ，4， 7， 9，4， 3， 3， 6， 8， 4， 7，9，5， 5， 6

由于 A[begin] >= pivot， 于是 A[begin]和 A[end]互换， end—

1 ，2 ，3， 7， 9，4， 3，4， 4， 6， 8， 4， 7，9，5， 5， 6

由于 A[begin] ＜ pivot， 于是 begin++:

1 ，2 ，3， 7， 9，4， 3，4， 4， 6， 8， 4， 7，9，5， 5， 6

由于 A[begin] >= pivot， 于是 A[begin]和 A[end]互换， end—

1 ，2 ，3， 3， 9， 4， 7，4， 4， 6， 8， 4， 7，9，5， 5， 6

由于 A[begin] < pivot, 于是 begin++:

1 ,2 ,3, 3, 9, 4, 7,4, 4, 6, 8, 4, 7,9,5, 5, 6

由于 A[begin] >= pivot, 于是 A[begin]和 A[end]互换, end—

1 ,2 ,3, 3, 4, 9, 7,4, 4, 6, 8, 4, 7,9,5, 5, 6

此时，end < begin, 第一次调整结束，我们看到走完第一步后，从 end 往后所有的元素都大于等于 pivot,接着执行第二部调整，此次调整大家作为练习。

需要注意的是，两次调整时，判断是否交互的条件不一样，第一次调整时，交换的条件是 A[being] >= pivot, 第二次调整时交换的条件是 A[begin] > pivot.

由于每一次调整，只需要一个循环，并且不需要分配多余的空间，所以时间复杂度是 O(n), 空间复杂度是 O(1).

接下来我们看看代码实现，我们把数组元素交互的逻辑实现放在一个函数中：

```
int[] reArrangeByPivot(int[] A, int begin, int end, int pivot, boolean checkEqual) {
    if (end <= begin) {
        return A;
    }

        while (begin < end) {

            if ( (checkEqual && A[begin] >= pivot)  || (checkEqual == false && A[begin] > pivot)){
                int temp = A[begin];
                A[begin] = A[end];
                A[end] = temp;
                end--;
            } else {
                begin++;
            }
        }

        return A;
    }
```

函数输入除了数组本身外，还有begin和end两个下标，用来表明要调整的数组部分，checkEqual用来判断，元素交互时判断条件是小于等于还是严格小于。

有了该辅助函数后，我们看看主函数的实现：

```
public int[] reArrangeArray(int[] A, int i) {
      if (A.length <= 1) {
         return A;
      }

      int pivot = A[i];
      A = reArrangeByPivot(A, 0, A.length - 1, pivot,
true);
      int j = 0;
      for (j = 0; j < A.length; j++) {
         if (A[j] >= pivot) {
            break;
         }
      }
```

```
        A = reArrangeByPivot(A, j, A.length - 1, pivot, false);

        return A;
    }
```

在主函数中，第一次调用 reArrangePivot 执行算法描述中的第一步调整，然后通过一个 for 循环找到调整后数组的第二部分，然后再调用 reArrangePivot 对第二部分进行算法描述的第二步调整，最终代码实现，可见后面的代码调试演示。

接着我们看看第二道有点难度的题目。

假设 A 是一个整数数组,长度为 n，数组中的元素可能是重复的。设计一个算法，找到一系列下标的集合 I = {i(0), i(1), i(2)….i(n)}. 使得(A[i(0)] + A[i(1)] + … A[i(n)]) mod n = 0.例如假定 A = {711, 704, 427, 995, 334, 62, 763, 98, 733, 721}, 于是 I = {0,1,3}, 因为(A[0] + A[1] + A[3]) mod 10 = 0.

请给出一个有效的算法找到满足条件的集合 I, 无论 A 的元素如何取值，这样的集合总是存在的。

这道题有些难度，主要在于前面我们提到的一些思维模式用不上，我们使用小样例分析法，假定 A 只有一个，或两个元素，例如 A = {1, 2}, A = { 11, 13}，我们发现，确实总存在集合 I，满足题目中的条件，但这些小样本实例无法揭示出怎样找到集合 I.

我们也提过，一旦遇到数组，或集合，我们要排序后看看是否有线索或头绪。对于这道题，即使排序后，任然看不出有什么结果。最坏的是，原来有用的暴力枚举法，在这里也用不上，因为你不知道 I 会包含几个元素，所以要暴力枚举，我们不知道需要几个循环间套。

此类题目一般用来选取团一个团队技术领导。它就纯粹的考察你是否具备足够水平的逻辑推导能力。

我们看看，随便选取若干个 A 中的元素相加后对 n 求余，看看有什么启发，假定我们选取 k 个元素，例如是 A[1]，A[2]…A[k]. 如果有

(A[1] + A[2] +…+A[k]) mod n = 0

那么 1,2 …k 就是满足条件的 I 集合。如果不等于 0，假设等于 t, t < n. 如果存在一个 l, 1 <= l <= k, 使得 A[l] mod n = t. 那么集合{1, 2,… l -1, l+1,…k} 就是满足条件

的集合。因为((A[1] + …A[l-1] + A[l+1] +…A[k]) + A[l]) mod n = t = A[l] mod n,也就是我们找到了一个等价条件：

((A[1] + …A[l-1] + A[l+1] +…A[k]) + A[l]) mod n ⇔ A[l] mod n

由此我们可以推导出(A[1] + …A[l-1] + A[l+1] +…A[k]) mod n = 0.

这样，我们就得到一个可行的算法：

```
int sum = 0;
for (int i = 0; i < n; i++) {

    sum += A[i];
    int t = sum % n;
    if (t == 0 ) {
       return {0,1,2, ….i};
    }

    if (存在一个 l, 使得 A[l] %n == t) {
        return {0, 1,…l - 1, l + 1, …i};
    }
}
```

接下来的问题就是如何判断是否已经存在一个这样的 l，我们使用另外一个长度为 n 的数组 B，先把 B 每个元素都初始化为 0，当 for 循环到 i 时，我们设置 B[A[i] mod n] = 1,如果有 B[t] = 1，也就是 t = A[i] mod n,那就意味着存在一个 l，使得 A[l] % n == t. 于是我们再将 0 到 i 中的每一个元素都拿出来，看看哪一个元素满足对 n 求余后等于 t，满足条件的元素就是 A[l].