kSum 篇

这篇总结主要介绍 LeetCode 中几道关于求 kSum 的题目，主要要求是在数组中寻找 k 个数的和能否达到目标值。

LeetCode 关于 kSum 的主要题目有：

1. Two Sum
2. 3Sum
3. 3Sum Closest
4. 4Sum

1. Two Sum，这道题目提供了后面k>2的题目的基本思路。主要有两种思路，第一种是利用哈希表对元素的出现进行记录，然后进来新元素时看看能不能与已有元素配成target，如果哈希表的访问是常量操作，这种算法时间复杂度是O(n)，空间复杂度也是O(n)。

代码如下：

```
// Solution 1 - HashMap
public int[] twoSum(int[] numbers, int target) {
    if(numbers == null || numbers.length < 2){
        return null;
    }
    int[] res = new int[2];
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int i = 0; i < numbers.length; i++){
        if(map.containsKey(target - numbers[i])){
            res[0] = map.get(target - numbers[i]) + 1;
            res[1] = i + 1;
            return res;
        }
        map.put(numbers[i], i);
    }
    return res;
}
```

第二种思路则是先对数组进行排序，然后利用夹逼的方法找出满足条件的pair，原理是因为数组是有序的，那么假设当前结果比target大，那么左端序号右移只会使两个数的和更大，反之亦然。所以每次只会有一个选择，从而实现线性就可以求出结果。算法的复杂度是O(nlogn+n)=O(nlogn)，空间复杂度取决于排序算法。

代码如下：

```
// Solution 2 - 夹逼
public int[] twoSum(int[] numbers, int target) {
    if(numbers == null || numbers.length < 2){
        return null;
    }
    Arrays.sort(numbers);
    int[] res = new int[2];
    int l = 0;
    int r = numbers.length - 1;
    while(l < r){
        if(numbers[l] + numbers[r] == target){
            res[0] = numbers[l];
            res[1] = numbers[r];
            return res;
        }
        if(numbers[l] + numbers[r] < target){
            l++;
        } else {
            r--;
        }
    }
    return res;
}
```

做题时的感悟：

HashMap方法中的感悟：

1. 首先，检查一个集合中是否有某个元素使用的方法分别是：Map.containsKey() 和 Set.contains() 不要搞混淆。

2. 深入一些，Map.containsKey() 和 Set.contains()在我们的代码中不应该经常出现，他们的功能一般会被以后要用到的方法涵盖。详细解释和例子参见附录。

3. 在这道题中，不能先把所有元素都放入HashMap中，因为HashMap不能保存重复的元素，所以重复元素的下标会被替换掉。因此，需要一个一个向HashMap中添加。

4. 注意审题，题目要返回的是数组中第几个数，而不是数组中元素的下标，所以结果应该为下标+1，并且题目要求index1小于index2， 所以res[0]和res[1]对应的下标+1不能互换：

```
res[0] = map.get(target - numbers[i]) + 1;
res[1] = i + 1;
```

5. Iterator用法：

Fail-Fast机制：

int expectedModCount = modCount;这句代码，其实是集合迭代中的一种“快速失败”机制，这种机制提供迭代过程中集合的安全性。阅读源码就可以知道ArrayList中存在modCount对象，增删操作都会使modCount++，通过两者的对比迭代器可以快速的知道迭代过程中是否存在list.add()类似的操作，存在的话快速失败!

记录修改此列表的次数：包括改变列表的结构，改变列表的大小，打乱列表的顺序等使正在进行迭代产生错误的结果。Tips:仅仅设置元素的值并不是结构的修改。我们知道的是ArrayList是线程不安全的，如果在使用迭代器的过程中有其他的线程修改了List就会抛出ConcurrentModificationException异常，这就是Fail-Fast机制。

那么快速失败究竟是个什么意思呢?

在ArrayList类创建迭代器之后，除非通过迭代器自身remove或add对列表结构进行修改，否则在其他线程中以任何形式对列表进行修改，迭代器马上会抛出异常，快速失败。调用Iterator自身的remove()方法删除当前元素是完全没有问题的，因为在这个方法中会自动同步expectedModCount和modCount的值

迭代器的好处：

1、迭代器可以提供统一的迭代方式。

2、迭代器可以在对客户端透明的情况下，提供各种不同的迭代方式。

3、迭代器提供一种快速失败机制，防止多线程下迭代的不安全操作。

foreach循环：

foreach语法冒号后面可以有两种类型：一种是数组，另一种是是实现了Iterable接口的类。

夹逼方法中的感悟：

1. 原理是因为数组是有序的，那么假设当前结果比target大，那么左端序号右移只会使两个数的和更大，反之亦然。所以每次只会有一个选择，从而实现线性就可以求出结果。这种每次只会有一个选择，可以根据结果确定移动哪一边的题目，可以用夹逼的方法来做。
2. 在这里，输出结果改成了满足相加等于target的两个数，而不是他们的index。因为要排序，如果要输出index，需要对原来的数的index进行记录，方法是构造一个数据结构，包含数字的值和index，然后排序。
3. 数组有序是这种解法的核心要求，所以切记查找前要对数组进行排序。

2. 3Sum，这道题和 Two Sum 有所不同，使用哈希表的解法并不是很方便，因为结果数组中元素可能重复，如果不排序对于重复的处理将会比较麻烦，因此这道题一般使用排序之后夹逼的方法，总的时间复杂度为 O(n^2+nlogn)=(n^2),空间复杂度是 O(n)，注意，在这里为了避免重复结果，对于已经判断过的数会 skip 掉，这也是排序带来的方便。 这道题考察的点其实和 Two Sum 差不多，Two Sum 是 3Sum 的一个 subroutine。

代码如下：

```
public ArrayList<ArrayList<Integer>> threeSum(int[] num) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length < 3){
        return res;
    }
    Arrays.sort(num);
    for(int i = num.length - 1; i >= 2; i--){
        if(i < num.length - 1 && num[i] == num[i + 1]){
            continue;
        }
        ArrayList<ArrayList<Integer>> curRes = findTwoSum(num, -num[i], 0, i -
1);
        for(int j = 0; j < curRes.size(); j++){
            curRes.get(j).add(num[i]);
        }
        res.addAll(curRes);
    }
    return res;
}
```

```
    private ArrayList<ArrayList<Integer>> findTwoSum(int[] num, int target, int l, int r){
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        while(l < r){
            if(target == num[l] + num[r]){
                ArrayList<Integer> item = new ArrayList<Integer>();
                item.add(num[l]);
                item.add(num[r]);
                res.add(item);
                l++;
                r--;
                while(l < r && num[l] == num[l - 1]){
                    l++;
                }
                while(l < r && num[r] == num[r + 1]){
                    r--;
                }
            } else if(target > num[l] + num[r]){
                l++;
            } else {
                r--;
            }
        }
        return res;
    }
```

做题时的感悟:

1. 由于ArrayList.add()是从后面进行添加，而且题目要求结果数组中的数字升序，所以最后添加进ArrayList的一定要是最大的，所以for循环应该从num.length - 1扫到2.

2. 为了避免结果重复，我们要跳过已经判断过的数:

```
            if(i < num.length - 1 && num[i] == num[i + 1]){
                continue;
            }
```

这点很重要，在很多有重复的数组中，为了避免重复的结果产生，都需要使用这个判断。

3. 因为可能有多组 2 个数的和为-num[i]，所以在找到一组后，要跳过重复元素，然后继续寻找。

3. 3Sum Closest，这道题跟3Sum很类似，区别就是要维护一个最小的closest，求出和目标最近的三个和。brute force时间复杂度为O(n^3)，优化的解法是使用排序之后夹逼的方法，总的时间复杂度为O(n^2+nlogn)=(n^2),空间复杂度是O(n)。

代码如下：

```
public int threeSumClosest(int[] num, int target) {
    if(num == null || num.length < 3){
        return -1;
    }
    Arrays.sort(num);
    int closest = Integer.MAX_VALUE;
    for(int i = 0; i < num.length - 2; i++){
        int value = twoSumClosest(num, i + 1, num.length - 1, target - num[i]);
        if(Math.abs(closest) > Math.abs(value)){
            closest = value;
        }
    }
    return target + closest;
}

private int twoSumClosest(int[] num, int l, int r, int target){
    int closest = Integer.MAX_VALUE;
    while(l < r){
        int value = num[l] + num[r] - target;
        if(Math.abs(value) < Math.abs(closest)){
            closest = value;
        }
        if(value == 0){
            return 0;
        } else if(value > 0){
            r--;
        } else {
            l++;
        }
    }
    return closest;
}
```

做题时的感悟:

1. 审题，题目最后求的是最接近target的3个数的和。所以twoSum中根据target - num[i]求的closest就是与target的差值，所以最后return的target + closest即是他们3个数的和。

2. 注意，这里的closest是有正负的，不是绝对值后的结果，这样才能根据target + closest正确求出3个数的和。

4. 4Sum，这道题要求跟3Sum差不多，只是需求扩展到四个的数字的和了。所以可以按照3Sum中的解法，只是在外面套一层循环，相当于求n次3Sum。我们知道3Sum的时间复杂度是O(n^2)，所以如果这样解的总时间复杂度是O(n^3)。

代码如下:

```
public ArrayList<ArrayList<Integer>> fourSum(int[] num, int target) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length < 4){
        return res;
    }
    Arrays.sort(num);
    for(int i = num.length - 1; i >= 3; i--){
        if(i == num.length - 1 || num[i] != num[i + 1]){
            ArrayList<ArrayList<Integer>> curRes = threeSum(num, 0, i - 1, target - num[i]);
            for(int j = 0; j < curRes.size(); j++){
                curRes.get(j).add(num[i]);
            }
            res.addAll(curRes);
        }
    }
    return res;
}

public ArrayList<ArrayList<Integer>> threeSum(int[] num, int l, int r, int target) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length < 3){
        return res;
    }
    for(int i = r; i >= 2; i--){
        if(i == r || num[i] != num[i + 1]){
```

```
                    ArrayList<ArrayList<Integer>> curRes = twoSum(num, 0, i - 1, target
- num[i]);
                    for(int j = 0; j < curRes.size(); j++){
                        curRes.get(j).add(num[i]);
                    }
                    res.addAll(curRes);
                }
            }
            return res;
        }

        public ArrayList<ArrayList<Integer>> twoSum(int[] num, int l, int r, int target) {
            ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
            if(num == null || num.length < 2){
                return res;
            }
            while(l < r){
                if(num[l] + num[r] == target){
                    ArrayList<Integer> item = new ArrayList<Integer>();
                    item.add(num[l]);
                    item.add(num[r]);
                    res.add(item);
                    l++;
                    r--;
                    while(l < r && num[l] == num[l - 1]){
                        l++;
                    }
                    while(l < r && num[r] == num[r + 1]){
                        r--;
                    }
                } else if(num[l] + num[r] < target){
                    l++;
                } else {
                    r--;
                }
            }
            return res;
        }
```

这道题的比较优的解法是用求解一般kSum的解法进行层层二分，然后用Two Sum结合起来。我们知道Two Sum是一个排序算法加上一个线性的夹逼操作，并且所有pair的数量是O((n-1)+(n-2)+...+1)=O(n(n-1)/2)=O(n^2)。所以对O(n^2)的排序如果不用特殊线性排序算法是O(n^2*log(n^2))=O(n^2*2logn)=O(n^2*logn)，算法复杂度比上一个方法的O(n^3)是有提高的。思路虽然明确，不过细节上会多很多情况要处理。首先，我们要对每一个pair建一个数据结构来存储元素的值和对应的index，这样做是为了后面当找到合适的两对pair相加能得到target值时看看他们是否有重叠的index，如果有说明它们不是合法的一个结果，因为不是四个不同的元素。接下来我们还得对这些pair进行排序，所以要给pair定义comparable的函数。最后，当进行Two Sum的匹配时因为pair不再是一个值，所以不能像Two Sum中那样直接跳过相同的，每一组都得进行查看，这样就会出现重复的情况，所以我们还得给每一个四个元素组成的tuple定义hashcode和相等函数，以便可以把当前求得的结果放在一个HashSet里面，这样得到新结果如果是重复的就不加入结果集了。

做题时的感悟：

1．注意！if和else if有着本质的区别，例如下面代码：

```
if(num[l] + num[r] == target){
    ArrayList<Integer> item = new ArrayList<Integer>();
    item.add(num[l]);
    item.add(num[r]);
    res.add(item);
    l++;
    r--;
    while(l < r && num[l] == num[l - 1]){
        l++;
    }
    while(l < r && num[r] == num[r + 1]){
        r--;
    }
} else if(num[l] + num[r] < target){
    l++;
} else {
    r--;
}
```

如果上述代码中把else if改成了if，那就完全错误了！因为当第一个if符合条件时，就不会执行else if代码块了。但如果用的是if，就会进行判断。如果第一个if中没有改变l或r的值，num[l]

+ num[r] == target 和 num[l] + num[r] < target 是不能同时发生的。但恰巧其中改了l或r的值，所以可能会不正确的进入第二个if代码块或第三个代码块让l++或者r—，导致错误结果。

2. Comparable Vs Comparator

Comparable & Comparator 都是用来实现集合中元素的比较、排序的，只是 Comparable 是在集合内部定义的方法实现的排序，Comparator 是在集合外部实现的排序，所以，如想实现排序，就需要在集合外定义 Comparator 接口的方法或在集合内实现 Comparable 接口的方法。

自定义的类要在加入list容器中后能够排序，可以实现Comparable接口，若一个类实现了Comparable接口，就意味着“该类支持排序”。在用Collections类的sort方法排序时，如果不指定Comparator，那么就以自然顺序排序，如API所说：

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface

这里的自然顺序就是实现Comparable接口设定的排序方式。

而 Comparator 是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足你的要求时，你可以写一个比较器来完成两个对象之间大小的比较。若一个类要实现Comparator接口：它一定要实现compare(T o1，T o2) 函数，但这个类可以不实现equals(Object obj) 函数。因为任何类，默认都是已经实现了equals(Object obj)的。Java中的一切类都是继承于java.lang.Object，在Object.java中实现了equals(Object obj)函数；所以，其它所有的类也相当于都实现了该函数。

可以说一个是自己完成比较，一个是外部程序实现比较的差别而已。Comparable相当于“内部比较器”，而Comparator相当于“外部比较器”。

用 Comparator 是策略模式（strategy design pattern），就是不改变对象自身，而用一个策略对象（strategy object）来改变它的行为。

比如：你想对整数采用绝对值大小来排序，Integer 是不符合要求的，你不需要去修改 Integer 类（实际上你也不能这么做）去改变它的排序行为，只要使用一个实现了 Comparator 接口的对象来实现控制它的排序就行了。

Comparable接口只提供了int compareTo(T o)方法，也就是说假如我定义了一个Person类，这个类实现了Comparable接口，那么当我实例化Person类的person1后，我想比较person1 和一个现有的Person对象person2的大小时，我就可以这样来调用：

person1.comparTo(person2)，通过返回值就可以判断了；而此时如果你定义了一个 PersonComparator（实现了Comparator接口）的话，那你可以这样来进行比较：

PersonComparator comparator = new PersonComparator();

comparator.compare(person1, person2);

Reference:

http://blog.csdn.net/mageshuai/article/details/3849143

http://www.cnblogs.com/skywang12345/p/3324788.html