实验一：用分治法求解数组的中位数和最大子集 报告

姓名：陈矛　　学号：SA22225116

一、实验原理

1. 分治法求解两个数组中位数的流程图：

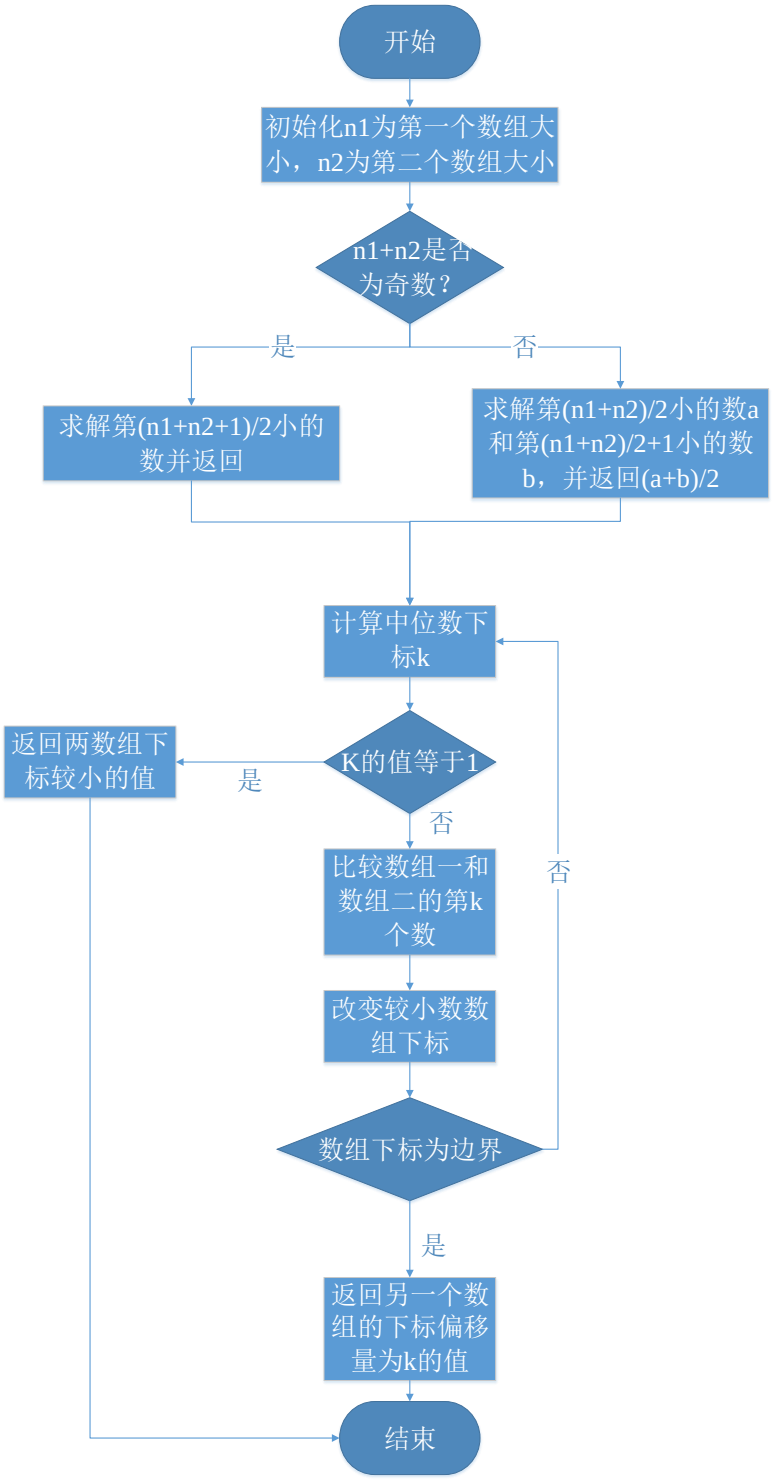

图 算法流程图

2. 分治法求最大子集的算法流程图

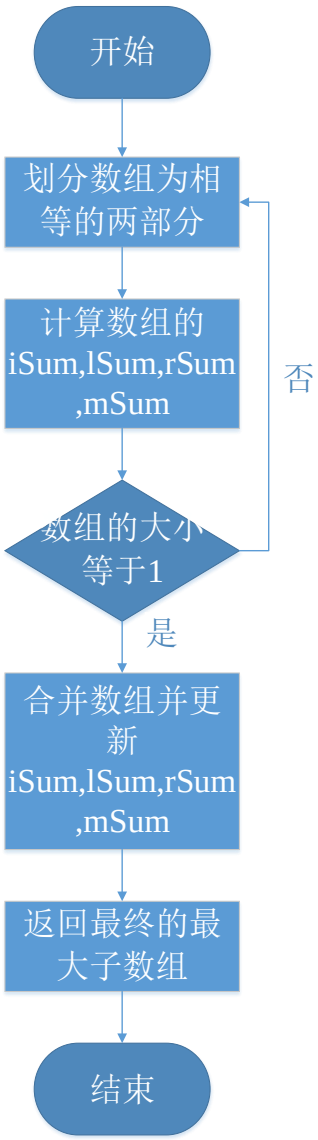

图 算法流程图

3. 分治法的概念

分治算法的基本思想是将一个规模为 N 的问题,分解成 K 个规模较小的子问题,这些子问题相互独立且月原问题性质相同。求解出子问题的解,合并得到原问题的解。分治法在每一层递归上有 3 个基本步骤。

（1）分解:将原问题逐层分解为若干个较小规模,相互独立且与原问题表示形式相同的子问题;

（2）求解:若分解后的子问题规模较小且求解容易则直接求解,否则将相应的子问题再不断分解为更小的子问题,直到容易求解为止;

（3）合并:将已经求解出的各子问题的解逐步合并,最后得到原问题的解。

2.适用分治策略解决的问题

对于一个输入规模为 n 并且取值比较大的问题,若能满足下面条件,使用分治法的思想就可以提高解决问题的效率。

（1）保证这 n 个数据能分解为 k 个不同的子集合,同时得到的 k 个子集合为可以独立求解的子问题（$1 \le k \le n$）;
（2）分解后的各子问题和原问题具有相似的表示形式,便于使用循环或递归机制;
（3）得到这些子问题的解之后,可以方便递推出原问题的解。

二、运行结果及分析

分治法求解数组的中位数：

1. 运行结果

```
int[] nums1 = {1,2};
int[] nums2 = {3,4};
```

```
E:\University\Java\develop_tools\jdk1.
2.5

Process finished with exit code 0
```

```
int[] nums1 = {1};
int[] nums2 = {3,4};
```

```
E:\University\Java\develop_tools\jdk1.
3.0

Process finished with exit code 0
```

2. 算法思想：

对于找出两个有序数组的中位数的问题，可以使用分治法来解决。

首先，需要将问题转化为求解两个有序数组的第 k 小数的问题。因为中位数就是两个有序数组中第(m+n)/2 小的数，其中 m 和 n 分别是两个数组的长度。

然后，我们可以使用分治法的思想来求解这个问题。我们可以先取出两个数组的第 k/2 小的数，设为 a 和 b。如果 a=b，那么这两个数就是第 k 小的数；如果 a<b，那么我们可以将第一个数组中的前 k/2 个数排除，因为这些数都不可能是第 k 小的数；如果 a>b，那么我们可以将第二个数组中的前 k/2 个数排除。我们可以递归地调用这个算法，直到找到第 k 小的数为止。

首先，我们需要定义一个函数 findKth，用于求解两个有序数组的第 k 小的数。这个函数的输入参数包括两个数组 nums1 和 nums2，以及一个数 k。在函数内部，我们需要做如下的操作：

a) 如果 nums1 数组的长度为 0，那么第 k 小的数就是 nums2 数组的第 k 小的数；如果 nums2 数组的长度为 0，那么第 k 小的数就是 nums1 数组的第 k 小的数。
b) 取出 nums1 数组的第 k/2 小的数 a 和 nums2 数组的第 k/2 小的数 b。
c) 如果 a=b，那么 a 和 b 就是第 k 小的数。
d) 如果 a<b，那么我们可以将 nums1 数组的前 k/2 个数排除，因为这些数都不

可能是第 k 小的数。然后我们递归地调用 findKth 函数，求解剩余的 nums1 和 nums2 数组的第 k-k/2 小的数。

e) 如果 a>b，那么我们可以将 nums2 数组的前 k/2 个数排除，然后递归地调用 findKth 函数，求解剩余的 nums1 和 nums2 数组的第 k-k/2 小的数。

findMedianSortedArrays 函数的输入参数是两个有序数组 nums1 和 nums2，其目的是求解这两个数组的中位数。

首先，我们需要计算两个数组的总长度 n1+n2。如果两个数组的总长度是奇数，那么中位数就是第(n1+n2+1)/2 小的数；如果两个数组的总长度是偶数，那么中位数就是第(n1+n2)/2 小的数和第(n1+n2)/2+1 小的数的平均数。

我们可以使用 findKth 函数求解第 k 小的数，所以我们可以直接调用 findKth 函数来求解中位数。

对于奇数的情况，我们直接调用 findKth 函数求解第(n1+n2+1)/2 小的数，然后将结果返回即可。

对于偶数的情况，我们先调用 findKth 函数求解第(n1+n2)/2 小的数 a，然后再调用 findKth 函数求解第(n1+n2)/2+1 小的数 b，最后将 a 和 b 的平均数返回即可。

最终，findMedianSortedArrays 函数就可以用来求解两个有序数组的中位数。

3. 算法流程图：

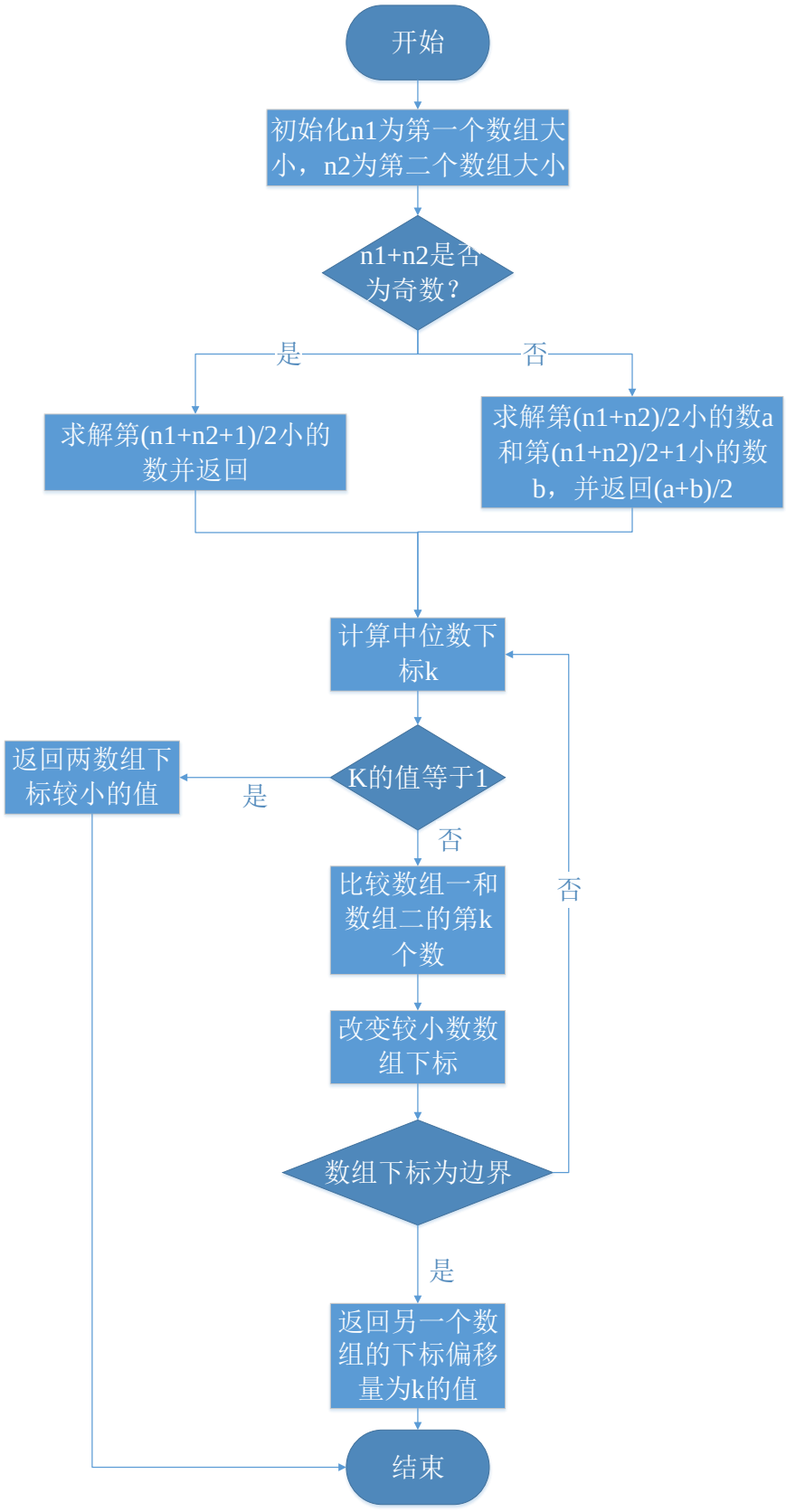

图 算法流程图

4. 关键代码：

```java
public static int findKth(int[] nums1, int[] nums2, int k) {

    // 如果nums1 数组为空，返回nums2 数组的第k 小的数
    if (nums1.length == 0) {
        return nums2[k - 1];
    }
    // 如果nums2 数组为空，返回nums1 数组的第k 小的数
    if (nums2.length == 0) {
        return nums1[k - 1];
    }
    // 如果k=1，返回nums1 数组的第1 小的数和nums2 数组的第1 小的数的较小
值
    if (k == 1) {
        return Math.min(nums1[0], nums2[0]);
    }
    // 取出nums1 数组的第k/2 小的数a 和nums2 数组的第k/2 小的数b
    int a = Integer.MAX_VALUE;
    int b = Integer.MAX_VALUE;
    if (k / 2 - 1 < nums1.length) {
        a = nums1[k / 2 - 1];
    }
    if (k / 2 - 1 < nums2.length) {
        b = nums2[k / 2 - 1];
    }

    // 如果a=b，返回a
    if (a == b) {
        return a;
    }
    // 如果a<b，将nums1 数组的前k/2 个数排除，递归地求解剩余的nums1 和
nums2 数组的第k-k/2 小的数
    if (a < b) {
        int[] newNums1 = Arrays.copyOfRange(nums1, k / 2,
nums1.length);
        return findKth(newNums1, nums2, k - k / 2);
    }
    // 如果a>b，将nums2 数组的前k/2 个数排除，递归地求解剩余的nums1 和
nums2 数组的第k-k/2 小的数
    else {
        int[] newNums2 = Arrays.copyOfRange(nums2, k / 2,
nums2.length);
        return findKth(nums1, newNums2, k - k / 2);
    }
}
```

```java
public static double findMedianSortedArrays(int[] nums1, int[]
nums2) {
    // 计算两个数组的总长度
    int n1 = nums1.length;
    int n2 = nums2.length;
    int n = n1 + n2;
    // 如果两个数组的总长度是奇数，求解第(n1+n2+1)/2 小的数并返回
    if (n % 2 == 1) {
        return findKth(nums1, nums2, (n1 + n2 + 1) / 2);
    }
    // 如果两个数组的总长度是偶数，求解第(n1+n2)/2 小的数 a 和第
(n1+n2)/2+1 小的数 b，并返回(a+b)/2
    else {
        int a = findKth(nums1, nums2, (n1 + n2) / 2);
        int b = findKth(nums1, nums2, (n1 + n2) / 2 + 1);
        return (a + b) / 2.0;
    }
}
```

分治法求解数组的最大子集：

1. 运行结果：

```
E:\University\java\develop_tools\jdk1.8.0_131\bin\java.exe ...
最大子数组为：[4, -1, 2, 1]
最大子数数组和为：6

Process finished with exit code 0
```

2. 算法思想：

使用分治法，首先需要把问题分解成子问题，然后用递归的方式，依次解决这些子问题。对于求最大子集这个问题，如果从数组中间位置将数组划分为两个子数组，那么问题可以分解为三个部分：

1) 左边部分的最大子数组。
2) 右边部分的最大子数组。
3) 穿过中间，包含左边部分最右边的那个元素以及右边部分最左边那个元素的，最大子数组。

那么整个问题就可以分解为四步：

1) 在数组的中间位置将数组划分为两个子数组。
2) 分别求解两个子数组的最大子数组。
3) 求出包含中间位置的最大子数组，这个最大子数组的左端点在左子数组内，右端点在右子数组内。
4) 合并三个最大子数组，得到整个数组的最大子数组。

3. 算法流程图：

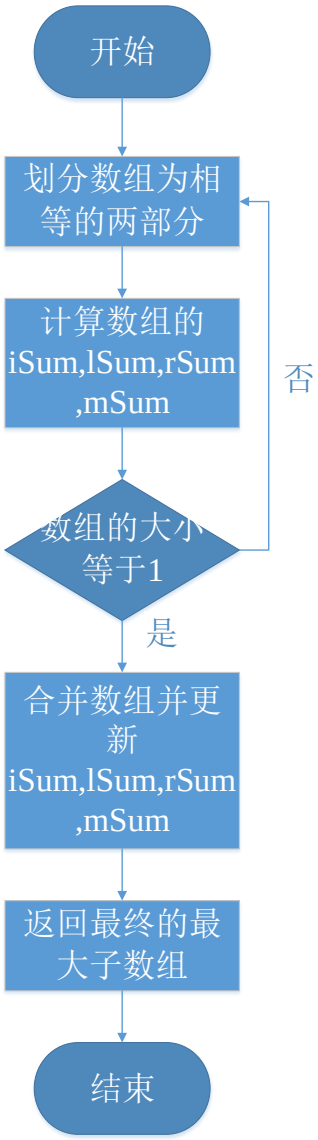

图 算法流程图

4. 关键代码：

```
package SolutionForLab1;

import java.util.Arrays;

/**
 * @author ChenMao
 * @create 2022-12-19 22:38
 */
public class MaximumSubarray {

    public static class Result {
        public int low;
        public int high;
```

```
        public int sum;

        public Result(int low, int high, int sum) {
            this.low = low;
            this.high = high;
            this.sum = sum;
        }
    }

    public static Result findMaximumSubarray(int[] A, int low, int
high) {
        if (low == high) {
            // 如果数组只有一个元素，则最大子数组就是这个元素本身
            return new Result(low, high, A[low]);
        } else {
            // 将数组划分为两个子数组
            int mid = (low + high) / 2;
            Result leftResult = findMaximumSubarray(A, low, mid);
            Result rightResult = findMaximumSubarray(A, mid + 1,
high);
            Result crossResult = findMaxCrossingSubarray(A, low, mid,
high);
            //找出三个部分的最大子数组

            if (leftResult.sum >= rightResult.sum &&
leftResult.sum >= crossResult.sum) {
                return leftResult;
            } else if (rightResult.sum >= leftResult.sum &&
rightResult.sum >= crossResult.sum) {
                return rightResult;
            } else {
                return crossResult;
            }
        }
    }

    public static Result findMaxCrossingSubarray(int[] A, int low,
int mid, int high) {
        int leftSum = Integer.MIN_VALUE;
        int sum = 0;
        int maxLeft = 0;
        //找到左边部分的最大总和，每找到一个更大的总和，就更新maxLeft
        for (int i = mid; i >= low; i--) {
            sum += A[i];
```

```
            if (sum > leftSum) {
                leftSum = sum;
                maxLeft = i;
            }
        }
        //找到右边部分的最大总和，每找到一个更大的总和，就更新 maxRight
        int rightSum = Integer.MIN_VALUE;
        sum = 0;
        int maxRight = 0;
        for (int i = mid + 1; i <= high; i++) {
            sum += A[i];
            if (sum > rightSum) {
                rightSum = sum;
                maxRight = i;
            }
        }

        return new Result(maxLeft, maxRight, leftSum + rightSum);
    }
```

三、实验总结

分治法是一种常用的求解问题的方法，它通过递归地把一个大问题分成多个小问题来解决。本次实验中，我们使用分治法来求解两个问题：

一是找出两个有序数组的中位数。二是找出一个数组中的最大子集。

在第一个问题里面，首先，需要将问题转化为求解两个有序数组的第 k 小数的问题。因为中位数就是两个有序数组中第(m+n)/2 小的数，其中 m 和 n 分别是两个数组的长度。

然后，我们可以使用分治法的思想来求解这个问题。我们可以先取出两个数组的第 k/2 小的数，设为 a 和 b。如果 a=b，那么这两个数就是第 k 小的数；如果 a<b，那么我们可以将第一个数组中的前 k/2 个数排除，因为这些数都不可能是第 k 小的数；如果 a>b，那么我们可以将第二个数组中的前 k/2 个数排除。我们可以递归地调用这个算法，直到找到第 k 小的数为止。在代码的细节里面，还需要注意奇数和偶数不同情况的讨论、下标越界的问题、还有当其中某一个数组的长度不足 k/2 时的情况的处理。

在第二个问题里面，为了求解数组的最大子集，我们首先要找到数组的中间位置，然后求解左半部分和右半部分的最大子集。我们还要考虑跨越中间位置的子集，即包含左半部分的最后一个元素和右半部分的第一个元素的子集。最后，我们取左半部分的最大子集、右半部分的最大子集和跨越中间位置的最大子集的最大值，作为整个数组的最大子集。

第二个问题的代码部分，我这里求解穿过分界线的最大子数组的方法是依次从交界线往前或往后累加，依次比较大小。这样子的方法可以解决问题，但肯定不是最好的。同时在这个实验里，其实也可以使用动态规划的方法或者使用累加数组的方法，同样可以解决问题，而且免去了反复的递归过程，时间复杂度会更优秀。

在第一个实验中，我们使用了两组示例数组，分别用总长度为奇数和偶数的情况都进行了测试，以检验算法在这两种情况下都可以实现功能。

在第二个实验中，我们使用了一个长度为 9 的整数数组[−2, 1, −3, 4, −1, 2, 1, −5, 4]，它的最大子集为[4, −1, 2, 1]，最大子集的和是 6。我们可以通过改变数组的元素值来验证算法的正确性。

总的来说，分治法是一种高效的求解问题的方法，它的时间复杂度为 O(log n)，比其他算法更快。但它也有一些缺点，比如需要额外的空间来存储子问题的答案，以及在处理某些问题时，需要额外的处理。因此，我们要根据实际情况选择合适的算法来求解问题。

实验日期：2022 年 11 月 18 日

附录

实验源代码：

MedianSortedArrays.java

```java
package SolutionForLab1;

import java.util.Arrays;

/**
 * @author ChenMao
 * @create 2022-11-18 16:09
 */
public class MedianSortedArrays {

    public static void main(String[] args) {
        int[] nums1 = {1};
        int[] nums2 = {3,4};

        System.out.println(findMedianSortedArrays(nums1,nums2));
    }

    public static int findKth(int[] nums1, int[] nums2, int k) {

        // 如果nums1 数组为空，返回nums2 数组的第k 小的数
        if (nums1.length == 0) {
```

```
            return nums2[k - 1];
        }
        // 如果nums2数组为空，返回nums1数组的第k小的数
        if (nums2.length == 0) {
            return nums1[k - 1];
        }
        // 如果k=1，返回nums1数组的第1小的数和nums2数组的第1小的数的较小值
        if (k == 1) {
            return Math.min(nums1[0], nums2[0]);
        }
        // 取出nums1数组的第k/2小的数a和nums2数组的第k/2小的数b
        int a = Integer.MAX_VALUE;
        int b = Integer.MAX_VALUE;
        if (k / 2 - 1 < nums1.length) {
            a = nums1[k / 2 - 1];
        }
        if (k / 2 - 1 < nums2.length) {
            b = nums2[k / 2 - 1];
        }

        // 如果a=b，返回a
        if (a == b) {
            return a;
        }
        // 如果a<b，将nums1数组的前k/2个数排除，递归地求解剩余的nums1和nums2数组的第k-k/2小的数
        if (a < b) {
            int[] newNums1 = Arrays.copyOfRange(nums1, k / 2,
nums1.length);
            return findKth(newNums1, nums2, k - k / 2);
        }
        // 如果a>b，将nums2数组的前k/2个数排除，递归地求解剩余的nums1和nums2数组的第k-k/2小的数
        else {
            int[] newNums2 = Arrays.copyOfRange(nums2, k / 2,
nums2.length);
            return findKth(nums1, newNums2, k - k / 2);
        }
    }

    public static double findMedianSortedArrays(int[] nums1, int[]
nums2) {
        // 计算两个数组的总长度
```

```
        int n1 = nums1.length;
        int n2 = nums2.length;
        int n = n1 + n2;
        // 如果两个数组的总长度是奇数，求解第(n1+n2+1)/2 小的数并返回
        if (n % 2 == 1) {
            return findKth(nums1, nums2, (n1 + n2 + 1) / 2);
        }
        // 如果两个数组的总长度是偶数，求解第(n1+n2)/2 小的数a 和第
(n1+n2)/2+1 小的数b，并返回(a+b)/2
        else {
            int a = findKth(nums1, nums2, (n1 + n2) / 2);
            int b = findKth(nums1, nums2, (n1 + n2) / 2 + 1);
            return (a + b) / 2.0;
        }
    }


}
```

MaximumSubarray.java

```
package SolutionForLab1;

import java.util.Arrays;

/**
 * @author ChenMao
 * @create 2022-12-19 22:38
 */
public class MaximumSubarray {

    public static class Result {
        public int low;
        public int high;
        public int sum;

        public Result(int low, int high, int sum) {
            this.low = low;
            this.high = high;
            this.sum = sum;
        }
    }

    public static Result findMaximumSubarray(int[] A, int low, int high)
{
        if (low == high) {
```

```
            // 如果数组只有一个元素，则最大子数组就是这个元素本身
            return new Result(low, high, A[low]);
        } else {
            // 将数组划分为两个子数组
            int mid = (low + high) / 2;
            Result leftResult = findMaximumSubarray(A, low, mid);
            Result rightResult = findMaximumSubarray(A, mid + 1, high);
            Result crossResult = findMaxCrossingSubarray(A, low, mid,
high);
            //找出三个部分的最大子数组

            if (leftResult.sum >= rightResult.sum && leftResult.sum >=
crossResult.sum) {
                return leftResult;
            } else if (rightResult.sum >= leftResult.sum &&
rightResult.sum >= crossResult.sum) {
                return rightResult;
            } else {
                return crossResult;
            }
        }
    }

    public static Result findMaxCrossingSubarray(int[] A, int low, int
mid, int high) {
        int leftSum = Integer.MIN_VALUE;
        int sum = 0;
        int maxLeft = 0;
        //找到左边部分的最大总和，每找到一个更大的总和，就更新maxLeft
        for (int i = mid; i >= low; i--) {
            sum += A[i];
            if (sum > leftSum) {
                leftSum = sum;
                maxLeft = i;
            }
        }
        //找到右边部分的最大总和，每找到一个更大的总和，就更新maxRight
        int rightSum = Integer.MIN_VALUE;
        sum = 0;
        int maxRight = 0;
        for (int i = mid + 1; i <= high; i++) {
            sum += A[i];
            if (sum > rightSum) {
                rightSum = sum;
```

```
                maxRight = i;
            }
        }

        return new Result(maxLeft, maxRight, leftSum + rightSum);
    }

    public static void main(String[] args) {
        int[] A = {-2,1,-3,4,-1,2,1,-5,4};
        Result result = findMaximumSubarray(A, 0, A.length - 1);
        System.out.println("最大子数组为: " +
Arrays.toString(Arrays.copyOfRange(A, result.low, result.high + 1)));
        System.out.println("最大子数数组和为: "+ result.sum);
    }
}
```