二分，即可缩小范围，最终确定 limit 到底是多少。具体过程参看如下代码中的 solution3 方法。

```
public int solution3(int[] arr, int num) {
        if (arr == null || arr.length == 0 || num < 1) {
                throw new RuntimeException("err");
        }
        if (arr.length < num) {
                int max = Integer.MIN_VALUE;
                for (int i = 0; i != arr.length; i++) {
                        max = Math.max(max, arr[i]);
                }
                return max;
        } else {
                int minSum = 0;
                int maxSum = 0;
                for (int i = 0; i < arr.length; i++) {
                        maxSum += arr[i];
                }
                while (minSum != maxSum - 1) {
                        int mid = (minSum + maxSum) / 2;
                        if (getNeedNum(arr, mid) > num) {
                                minSum = mid;
                        } else {
                                maxSum = mid;
                        }
                }
                return maxSum;
        }
}
```

假设 arr 所有值的累加和为 S，那么二分的次数为 log$S$，每次调用 getNeedNum 方法，然后进行二分，getNeedNum 方法的时间复杂度为 $O(N)$。所以 solution3 的时间复杂度为 $O(N\log S)$。

邮局选址问题

【题目】

一条直线上有居民点，邮局只能建在居民点上。给定一个有序整型数组 arr，每个值表示居民点的一维坐标，再给定一个正数 num，表示邮局数量。选择 num 个居民点建立 num 个邮局，使所有的居民点到邮局的总距离最短，返回最短的总距离。

【举例】

arr=[1,2,3,4,5,1000]，num=2。

第一个邮局建立在 3 位置，第二个邮局建立在 1000 位置。那么 1 位置到邮局距离为 2，2 位置到邮局距离为 1，3 位置到邮局距离为 0，4 位置到邮局距离为 1，5 位置到邮局距离为 2，1000 位置到邮局距离为 0。所以这种方案下的总距离为 6，其他任何方案的总距离都不会比该方案的总距离更短，所以返回 6。

【难度】

校 ★★★☆

【解答】

方法一，动态规划。首先解决一个问题，如果在 arr[i..j](0<=i<=j<N)区域上只能建一个邮局，并且这个区域上的居民点都前往这个邮局，要让 arr[i..j]上所有的居民点到邮局的总距离最短，这个邮局应该建在哪里？如果 arr[i..j]上有奇数个民居点，邮局建在中点位置会使总距离最短；如果 arr[i..j]上有偶数个民居点，此时认为中点有两个，邮局建在哪个中点上都行，都会使总距离最短。根据这种思路，我们先生成一个规模为 $N \times N$ 的矩阵 w，w[i][j](0<=i<=j<N)的值代表如果在 arr[i..j](0<=i<=j<N)区域上只建一个邮局，这一区间上的总距离为多少。因为始终有 i<=j 的要求，所以我们求 w 矩阵的时候，实际上只求 w 矩阵的一半。

求 w 矩阵的过程。在求每一个位置 w[i][j]的时候，求法并不是把区间 arr[i..j]上的每个位置到中点的距离求出后累加，这样求虽然肯定正确，但会很慢。更快速的求法是如果已经求出了 w[i][j-1]的值，那么 w[i][j]=w[i][j-1]+arr[j]-arr[(i+j)/2]。解释一下这是为什么，如果 arr[i..j-1]上有奇数个点，那么中点是 arr[(i+j-1)/2]，加上 arr[j]之后，arr[i..j]有偶数个点，第一个中点是 arr[(i+j)/2]。在这种情况下，(i+j-1)/2 和(i+j)/2 其实是同一个位置。比如，arr[i..j-1]=[4,15,26]，中点是 15。arr[i..j]=[4,15,26,47]，第一个中点是 15。所以，此时 w[i][j]比 w[i][j-1]多出来的距离就是 arr[j]到 arr[(i+j)/2]的距离，即 w[i][j]=w[i][j-1]+arr[j]-arr[(i+j)/2]。如果 arr[i..j-1]上有偶数个点，中点有两个，无论选在哪一个，w[i][j-1]的值都是一样的。加上 arr[j]之后，arr[i..j]有奇数个点，中点是 arr[(i+j)/2]。在这种情况下，arr[i..j-1]上的第二个中点和 arr[i..j]上唯一的中点其实是同一个位置。比如，arr[i..j-1]=[4,15,26,47]，第二个中点是 26。arr[i..j]=[4,15,26,47,53]，唯一的中点是 26。所以，

此时 w[i][j] 比 w[i][j-1] 多出来的距离还是 arr[j] 到 arr[(i+j)/2] 的距离，即 w[i][j]=w[i][j-1]+arr[j]-arr[(i+j)/2]。所以 *w* 矩阵求解的代码片段如下：

```
int[][] w = new int[arr.length + 1][arr.length + 1];
for (int i = 0; i < arr.length; i++) {
        for (int j = i + 1; j < arr.length; j++) {
                w[i][j] = w[i][j - 1] + arr[j] - arr[(i + j) / 2];
        }
}
```

如上代码中让把 *w* 申请成规模(N+1)×(N+1)的原因是为了在接下来的代码实现中，省去很多越界的判断，实际上 *w* 的有效区域就是 w[0..N][0..N]中的一半，剩下的部分都是 0。

有了 *w* 矩阵之后，接下来介绍动态规划的过程。dp[a][b]的值代表如果在 arr[0..b]上建设 *a*+1 个邮局，总距离最少是多少。所以 dp[0][b]的值代表如果在 arr[0..b]上建设 1 个邮局，总距离最少是多少。很明显，总距离最少是 w[0][b]。那么 dp[0][0..N-1]上的所有值都可以直接赋值，即如下的代码片段：

```
int[][] dp = new int[num][arr.length];
for (int j = 0; j != arr.length; j++) {
        dp[0][j] = w[0][j];
}
```

当 arr[0..b]上可以建设不止 1 个邮局时，即 dp[a][b](a>0)时，应该如何计算？举例说明，比如 arr=[-3,-2,-1,0,1,2]，要计算 dp[2][5]的值，即可以在 arr[0..5]上建立 3 个邮局的情况下，最少的最距离是多少，并且此时已经有 dp[0..1][0..5]的所有值。

方案 1：邮局 1、2 负责[-3]，邮局 3 负责[-2,-1,0,1,2]，距离 dp[1][0]+w[1][5]。

方案 2：邮局 1、2 负责[-3,-2]，邮局 3 负责[-1,0,1,2]，距离 dp[1][1]+w[2][5]。

方案 3：邮局 1、2 负责[-3,-2,-1]，邮局 3 负责[0,1,2]，距离 dp[1][2]+w[3][5]。

方案 4：邮局 1、2 负责[-3,-2,-1,0]，邮局 3 负责[1,2]，距离 dp[1][3]+w[4][5]。

方案 5：邮局 1、2 负责[-3,-2,-1,0,1]，邮局 3 负责[2]，距离 dp[1][4]+w[5][5]。

方案 6：邮局 1、2 负责[-3,-2,-1,0,1,2]，邮局 3 负责[]，距离 dp[1][5]+w[6][5](w 越界为 0)。

枚举所有的划分方案，选一个距离最短的即可，所以，dp[a][b] = Min { dp[a - 1][k] + w[k + 1] [b] (0<=k<N) }。

方法一的全部过程请参看如下代码中的 minDistances1 方法。

```
public int minDistances1(int[] arr, int num) {
        if (arr == null || num < 1 || arr.length < num) {
                return 0;
        }
```

```
        int[][] w = new int[arr.length + 1][arr.length + 1];
        for (int i = 0; i < arr.length; i++) {
                for (int j = i + 1; j < arr.length; j++) {
                        w[i][j] = w[i][j - 1] + arr[j] - arr[(i + j) / 2];
                }
        }
        int[][] dp = new int[num][arr.length];
        for (int j = 0; j != arr.length; j++) {
                dp[0][j] = w[0][j];
        }
        for (int i = 1; i < num; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = 0; k <= j; k++) {
                    dp[i][j] = Math.min(dp[i][j], dp[i - 1][k] + w[k + 1][j]);
                }
            }
        }
        return dp[num - 1][arr.length - 1];
}
```

w 矩阵的求解过程 $O(N^2)$，动态规划的求解过程 $O(N^2\times\text{num})$。所以方法一总的时间复杂度为 $O(N^2)+O(N^2\times\text{num})$，即 $O(N^2\times\text{num})$。

方法二，用四边形不等式优化动态规划的枚举过程，使整个过程的时间复杂度降低至 $O(N^2)$。在方法一中求解 dp[a][b]的时候，几乎枚举了所有的 dp[a-1][0..b]，但这个枚举过程其实是可以得到加速的。具体解释为：

1．当邮局为 a-1 个，区间为 arr[0..b]时，如果在其最优划分方案中发现，邮局 1~a-2 负责 arr[0..l]，邮局 a-1 负责 arr[l+1..b]。那么当邮局为 a 个，区间为 arr[0..b]时，如果想得到最优方案，邮局 1~a-1 负责的区域不必尝试比 arr[0..l]小的区域，只需尝试 arr[0..k](k>=l)。

2．当邮局为 a 个，区间为 arr[0..b+1]时，如果在其最优划分方案中发现，邮局 1~a-1 负责 arr[0..m]，邮局 a 负责 arr[m+1..b+1]。那么当邮局为 a 个，区间为 arr[0..b]时，如果想得到最优方案，邮局 1~a-1 负责的区域不必尝试比 arr[0..m]大的区域，只尝试 arr[0..k](k<=m)。

本题为何能用四边形不等式进行优化的证明略。有兴趣的读者可以自行学习“四边形不等式”的相关内容。有了这个枚举优化过程后，在算 dp[a][b]时，只用在 dp[a-1][b]的最优尝试位置 l 和 dp[a][b+1]的最优尝试位置 m 之间进行枚举，其他的位置一概不用再试。具体过程请参看如下代码中的 minDistances2 方法。

```
public int minDistances2(int[] arr, int num) {
        if (arr == null || num < 1 || arr.length < num) {
                return 0;
```

```
        }
        int[][] w = new int[arr.length + 1][arr.length + 1];
        for (int i = 0; i < arr.length; i++) {
                for (int j = i + 1; j < arr.length; j++) {
                        w[i][j] = w[i][j - 1] + arr[j] - arr[(i + j) / 2];
                }
        }
        int[][] dp = new int[num][arr.length];
        int[][] s = new int[num][arr.length];
        for (int j = 0; j != arr.length; j++) {
                dp[0][j] = w[0][j];
                s[0][j] = 0;
        }
        int minK = 0;
        int maxK = 0;
        int cur = 0;
        for (int i = 1; i < num; i++) {
            for (int j = arr.length - 1; j > i; j--) {
                minK = s[i - 1][j];
                maxK = j == arr.length - 1 ? arr.length - 1 : s[i][j + 1];
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = minK; k <= maxK; k++) {
                    cur = dp[i - 1][k] + w[k + 1][j];
                    if (cur <= dp[i][j]) {
                        dp[i][j] = cur;
                        s[i][j] = k;
                    }
                }
            }
        }
        return dp[num - 1][arr.length - 1];
}
```