```
        }
        int[] dp = new int[kChess];
        int res = 0;
        while (true) {
                res++;
                int previous = 0;
                for (int i = 0; i < dp.length; i++) {
                        int tmp = dp[i];
                        dp[i] = dp[i] + previous + 1;
                        previous = tmp;
                        if (dp[i] >= nLevel) {
                                return res;
                        }
                }
        }
}

public int log2N(int n) {
        int res = -1;
        while (n != 0) {
                res++;
                n >>>= 1;
        }
        return res;
}
```

画匠问题

【题目】

给定一个整型数组 arr，数组中的每个值都为正数，表示完成一幅画作需要的时间，再给定一个整数 num 表示画匠的数量，每个画匠只能画连在一起的画作。所有的画家并行工作，请返回完成所有的画作需要的最少时间。

【举例】

arr=[3,1,4]，num=2。

最好的分配方式为第一个画匠画 3 和 1，所需时间为 4。第二个画匠画 4，所需时间为 4。因为并行工作，所以最少时间为 4。如果分配方式为第一个画匠画 3，所需时间为 3。第二个画匠画 1 和 4，所需的时间为 5。那么最少时间为 5，显然没有第一种分配方式好。所以返回 4。

arr=[1,1,1,4,3]，num=3。

最好的分配方式为第一个画匠画前三个 1，所需时间为 3。第二个画匠画 4，所需时间

为 4。第三个画匠画 3，所需时间为 3。返回 4。

【难度】

校 ★★★☆

【解答】

方法一。如果只有 1 个画匠，那么对这个画匠来说，arr[0..j]上的画作最少时间就是 arr[0..j]的累加和。如果有 2 个画匠，对他们来说，画完 arr[0..j]上的画作有如下方案：

方案 1：画匠 1 负责 arr[0]，画匠 2 负责 arr[1..j]，时间为 Max{sum[0],sum[1..j]}。

方案 2：画匠 1 负责 arr[0..1]，画匠 2 负责 arr[2..j]，时间为 Max{sum[0..1],sum[2..j]}。

……

方案 *k*：画匠 1 负责 arr[0..k]，画匠 2 负责 arr[k+1..j]，时间为 Max{sum[0..k],sum[k+1..j]}。

方案 *j*：画匠 1 负责 arr[0..j-1]，画匠 2 负责 arr[j]。时间为 Max{sum[0..j-1],sum[j]}。

每一种方案其实都是一种划分，把 arr[0..j]分成两部分，第一部分由画匠 1 来负责，第二部分由画匠 2 来负责，两部分的累加和哪个大，哪个就是这种方案的所需时间。最后选所需时间最小的方案，就是答案。当画匠数量为 *i*（*i*>2）时，假设 dp[i][j]的值代表 *i* 个画匠搞定 arr[0..j]这些画所需的最少时间。那么有如下方案：

方案 1：画匠 1~*i*-1 负责 arr[0]，画匠 *i* 负责 arr[1..j] -> max{dp[i-1][0],sum[1..j]}。

方案 2：画匠 1~*i*-1 负责 arr[0..1]，画匠 *i* 负责 arr[2..j] -> max{dp[i-1][1],sum[2..j]}。

……

方案 *k*：画匠 1~*i*-1 负责 arr[0..k]，画匠 *i* 负责 arr[k+1..j] -> max{dp[i-1][k],sum[k+1..j]}。

方案 *j*：画匠 1~*i*-1 负责 arr[0..j-1]，画匠 *i* 负责 arr[j] -> max{dp[i-1][j-1] , sum[j]}。

哪种方案所需的时间最少，dp[i][j]的值就是那种方案所需的时间，即

dp[i][j] = min { max { dp[i-1][k] , sum[k+1..j] } (0<=k<j) }

具体过程参见如下代码中的 solution1 方法，此方法使用动态规划常见的空间优化技巧。因为 dp[i][j]的值仅依赖 dp[i-1][…]的值，所以我们不必生成规模为 Num×*N* 大小的矩阵，仅用一个长度为 *N* 的数组结构滚动更新、不断复用即可。

```
public int solution1(int[] arr, int num) {
        if (arr == null || arr.length == 0 || num < 1) {
                throw new RuntimeException("err");
        }
        int[] sumArr = new int[arr.length];
        int[] map = new int[arr.length];
```

```
        sumArr[0] = arr[0];
        map[0] = arr[0];
        for (int i = 1; i < sumArr.length; i++) {
                sumArr[i] = sumArr[i - 1] + arr[i];
                map[i] = sumArr[i];
        }
        for (int i = 1; i < num; i++) {
                for (int j = map.length - 1; j > i - 1; j--) {
                        int min = Integer.MAX_VALUE;
                        for (int k = i - 1; k < j; k++) {
                            int cur = Math.max(map[k], sumArr[j] - sumArr[k]);
                            min = Math.min(min, cur);
                        }
                        map[j] = min;
                }
        }
        return map[arr.length - 1];
}
```

画匠数目为 num，画作数量为 N，所以一共是 num×N 个位置需要计算，每一个位置都需要枚举所有的方案来找出最好的方案，所以方法一的时间复杂度为 $O(N^2\times num)$。

方法二，动态规划用四边形不等式优化后的解法。计算动态规划的每个值都需要去枚举，自然想到用“四边形不等式”及其相关猜想来做枚举优化。具体地说，假设计算 dp[i-1][j] 时，在最好的划分方案中，第 i-1 个画匠负责 arr[l..j]的画作。在计算 dp[i][j+1]时，在最好的划分方案中，第 i 个画匠负责 arr[m..j+1]的画作。那么在计算 dp[i][j]时，假设最好的划分方案是让第 i 个画匠负责 arr[k..j]，那么 k 的范围一定是[l,m]，而不可能在这个范围之外。四边形不等式的相关内容及其证明比较复杂且烦琐，本书因篇幅所限，不再详述，有兴趣的读者可以自行学习。利用四边形不等式对枚举过程的优化可以将时间复杂度从 $O(N^2\times num)$ 降至 $O(N^2)$。具体过程请参看如下代码中的 solution2 方法。

```
public int solution2(int[] arr, int num) {
        if (arr == null || arr.length == 0 || num < 1) {
                throw new RuntimeException("err");
        }
        int[] sumArr = new int[arr.length];
        int[] map = new int[arr.length];
        sumArr[0] = arr[0];
        map[0] = arr[0];
        for (int i = 1; i < sumArr.length; i++) {
                sumArr[i] = sumArr[i - 1] + arr[i];
                map[i] = sumArr[i];
        }
        int[] cands = new int[arr.length];
        for (int i = 1; i < num; i++) {
                for (int j = map.length - 1; j > i - 1; j--) {
                        int minPar = cands[j];
```

```
                    int maxPar = j == map.length - 1 ? j : cands[j + 1];
                    int min = Integer.MAX_VALUE;
                    for (int k = minPar; k < maxPar + 1; k++) {
                        int cur = Math.max(map[k], sumArr[j] - sumArr[k]);
                        if (cur <= min) {
                            min = cur;
                            cands[j] = k;
                        }
                    }
                    map[j] = min;
                }
            }
            return map[arr.length - 1];
        }
```

最优解。本题最优解反而是三个方法中最好理解的，先来重新思考这样一个问题，arr 数组中的值依然表示完成一幅画作需要的时间，但是规定每个画匠画画的时间不能多于 limit，那么要几个画匠才够呢？这个问题的实现非常简单，从左到右遍历 arr 的过程中做累加，一旦累加超过 limit，则认为当前的画（arr[i]）必须分给下一个画匠，那么就让累加和清零，并从 arr[i]开始重新累加。遍历的过程中如果发现有某一幅画的时间大于 limit，说明即使是单独分配一个画匠只画这一幅画，也不能满足每个画匠所需时间小于或等于 limit 这个要求。遇到这种情况就直接返回系统最大值，表示无论分多少个画匠，limit 都满足不了。这个过程请参看如下代码中的 getNeedNum 方法。如果 arr 的长度为 N，该方法的时间复杂度为 $O(N)$。

```
    public int getNeedNum(int[] arr, int lim) {
            int res = 1;
            int stepSum = 0;
            for (int i = 0; i != arr.length; i++) {
                    if (arr[i] > lim) {
                            return Integer.MAX_VALUE;
                    }
                    stepSum += arr[i];
                    if (stepSum > lim) {
                            res++;
                            stepSum = arr[i];
                    }
            }
            return res;
    }
```

理解了上面的小问题后，画匠问题最优解的思路就很好理解了——利用二分法。通过调整 limit 的大小，看看需要的画匠数目是大于画匠总数还是少于画匠总数，然后决定是将答案往上调整还是往下调整，那么 limit 的范围一开始为[0,arr 所有值的累加和]，然后不断

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。