

```

        disArr[i] = 0; // 重要
        while (true) {
            index = -index;
            if (disArr[index] > -1) {
                disArr[index]++;
                break;
            } else {
                int nextIndex = disArr[index];
                disArr[index] = 1;
                index = nextIndex;
            }
        }
    }
    disArr[0] = 1;
}

```

paths 转成距离数组的过程中，每一个城市只经历跳出去和跳回来两个过程，距离数组转成统计数组的过程也是如此，所以时间复杂度为 $O(N)$ ，整个过程没有使用额外的数据结构，只使用了有限几个变量，所以额外空间复杂度为 $O(1)$ 。全部过程请参看如下代码中的 pathsToNums 方法，这也是主方法。

```

public void pathsToNums(int[] paths) {
    if (paths == null || paths.length == 0) {
        return;
    }
    // citiesPath -> distancesArray
    pathsToDistans(paths);

    // distancesArray -> numArray
    distansToNums(paths);
}

```

正数数组的最小不可组成和

【题目】

给定一个正数数组 arr，其中所有的值都为整数，以下是最小不可组成和的概念：

- 把 arr 每个子集内的所有元素加起来会出现很多值，其中最小的记为 min，最大的记为 max。
- 在区间[min,max]上，如果有数不可以被 arr 某一个子集相加得到，那么其中最小的那个数是 arr 的最小不可组成和。
- 在区间[min,max]上，如果所有的数都可以被 arr 的某一个子集相加得到，那么

$\text{max}+1$ 是 arr 的最小不可组成和。

请写函数返回正数数组 arr 的最小不可组成和。

【举例】

$\text{arr}=[3,2,5]$ 。子集 $\{2\}$ 相加产生 2 为 min ，子集 $\{3,2,5\}$ 相加产生 10 为 max 。在区间 $[2,10]$ 上，4、6 和 9 不能被任何子集相加得到，其中 4 是 arr 的最小不可组成和。

$\text{arr}=[1,2,4]$ 。子集 $\{1\}$ 相加产生 1 为 min ，子集 $\{1,2,4\}$ 相加产生 7 为 max 。在区间 $[1,7]$ 上，任何数都可以被子集相加得到，所以 8 是 arr 的最小不可组成和。

【进阶题目】

如果已知正数数组 arr 中肯定有 1 这个数，是否能更快地得到最小不可组成和？

【难度】

尉 ★★★☆☆

【解答】

解法一为暴力递归的方法，即收集每一个子集的累加和，存到一个哈希表里，然后从 min 开始递增检查，看哪个正数不在哈希表中，第一个不在哈希表中的正数就是结果。具体请参见如下代码中的 `unformedSum1` 方法。

```
public int unformedSum1(int[] arr) {
    if (arr == null || arr.length == 0) {
        return 1;
    }
    HashSet<Integer> set = new HashSet<Integer>();
    process(arr, 0, 0, set); // 收集所有子集的和
    int min = Integer.MAX_VALUE;
    for (int i = 0; i != arr.length; i++) {
        min = Math.min(min, arr[i]);
    }
    for (int i = min + 1; i != Integer.MIN_VALUE; i++) {
        if (!set.contains(i)) {
            return i;
        }
    }
    return 0;
}

public void process(int[] arr, int i, int sum, HashSet<Integer> set) {
```

```

        if (i == arr.length) {
            set.add(sum);
            return;
        }
        process(arr, i + 1, sum, set); // 包含当前数 arr[i] 的情况
        process(arr, i + 1, sum + arr[i], set); // 不包含当前数 arr[i] 的情况
    }
}

```

如果 `arr` 长度为 N ，那么子集的个数为 $O(2^N)$ ，所以暴力递归方法的时间复杂度为 $O(2^N)$ ，收集子集和的过程中，递归函数 `process` 最多有 N 层，所以额外空间复杂度为 $O(N)$ 。

解法二是动态规划的方法。假设 `arr` 所有数的累加和为 `sum`，那么 `arr` 子集的累加和必然都在 $[0, \text{sum}]$ 区间上。于是生成长度为 `sum+1` 的 `boolean` 型数组 `dp[]`，`dp[j]` 如果为 `true`，则表示 `j` 这个累加和能够被 `arr` 的子集相加得到，如果为 `false`，则表示不能。如果 `arr[0..i]` 这个范围上的数组成的所有子集可以累加出 `k`，那么 `arr[0..i+1]` 这个范围上的数组成的所有子集则必然可以累加出 `k+arr[i+1]`。具体过程请参看如下代码中的 `unformedSum2` 方法。

```

public int unformedSum2(int[] arr) {
    if (arr == null || arr.length == 0) {
        return 1;
    }
    int sum = 0;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i != arr.length; i++) {
        sum += arr[i];
        min = Math.min(min, arr[i]);
    }
    boolean[] dp = new boolean[sum + 1];
    dp[0] = true;
    for (int i = 0; i != arr.length; i++) {
        for (int j = sum; j >= arr[i]; j--) {
            dp[j] = dp[j - arr[i]] ? true : dp[j];
        }
    }
    for (int i = min; i != dp.length; i++) {
        if (!dp[i]) {
            return i;
        }
    }
    return sum + 1;
}

```

更新 `dp[]` 时，从 `arr[0..i]` 的子集和状态更新到 `arr[0..i+1]` 的子集和状态的过程中， $0 \sim \text{sum}$ 的累加和都要看是否能被加出来，所以每次更新的时间复杂度为 $O(\text{sum})$ 。子集和状态从 `arr[0]` 的范围增长到 `arr[0..N-1]`，所以更新的次数为 N 。所以解法二的时间复杂度为 $O(N \times \text{sum})$ ，额外空间就是 `dp[]` 的长度，即额外空间复杂度为 $O(N)$ 。

进阶问题，如果正数数组 `arr` 中肯定有 1 这个数，求最小不可组成和的过程可以得到很好的优化，优化后可以做到时间复杂度为 $O(\log N)$ ，额外空间复杂度为 $O(1)$ 。具体过程为：

1. 把 `arr` 排序，排序之后则必有 `arr[0]=1`。
2. 从左往右计算每个位置 i 的 `range(0 \leq i < N)`。`range` 代表当计算到 `arr[i]` 时，`[1, range]` 区间内的所有正数都可以被 `arr[0..i-1]` 的某一个子集加出来，初始时，`arr[0]=1`，`range=0`。
3. 如果 `arr[i]>range+1`，因为 `arr` 是有序的，所以 `arr[i]` 往后的数都不会出现 `range+1`，所以直接返回 `range+1`。如果 `arr[i] \leq range+1`，说明 `[1, range+arr[i]]` 区间上的所有正数都可以被 `arr[0..i]` 的某一个子集加出来，所以令 `range+=arr[i]`，继续计算下一个位置。
4. 如果所有的位置都没有出现 `arr[i]>range+1` 的情况，直接返回 `range+1`。

步骤 1 的时间复杂度为 $O(\log N)$ ，步骤 2~步骤 4 的时间复杂度为 $O(N)$ 。所以整个过程的时间复杂度为 $O(\log N)$ ，额外空间复杂度为 $O(1)$ 。

举例说明一下，`arr=[3,8,1,2]`，排序后为`[1,2,3,8]`，计算开始前 `range=0`。

计算到 1 时，`range` 更新成 1，表示`[1,1]`区间上的正数都可以被 `arr[0]` 的某个子集加出来。

计算到 2 时，`range` 更新成 3，表示`[1,3]`区间上的正数都可以被 `arr[0..1]` 某个子集加出来。

计算到 3 时，`range` 更新成 6，表示`[1,6]`区间上的正数都可以被 `arr[0..2]` 某个子集加出来。

计算到 8 时，第一次出现 `8>range+1`，此时可知 7 这个数永无可能被得到，直接返回 7。

具体过程请参看如下代码中的 `unformedSum3` 方法。

```
public int unformedSum3(int[] arr) {
    if (arr == null || arr.length == 0) {
        return 0;
    }
    Arrays.sort(arr); // 把 arr 排序
    int range = 0;
    for (int i = 0; i != arr.length; i++) {
        if (arr[i] > range + 1) {
            return range + 1;
        } else {
            range += arr[i];
        }
    }
    return range + 1;
}
```