第一题

正如试题解析所说，是一道约瑟夫环问题，先根据递推关系式求出最后存活的人（在最后一轮的编号为 0）在最初的编号 t,然后计算好人的初始编号为 t 的概率，

$$P(goodManWin) = \Sigma_{i\in\{i|a[i]==1\}} P(i\ initial\_number\_is\_t)$$

$$= \Sigma_{i\in\{i|a[i]==1\}} P((i-t+n)\%n\ initial\_number\_is\_0)$$

$$= \Sigma_{i\in\{i|a[i]==1\}} w[(i-t+n)\%n]/\Sigma_{i=0}^{n-1} w[i]$$

代码：

```
class GoodManWin {//question 1
    private double probability;
    GoodManWin(int[] a, int[] w, int m) {
    int n = a.length;
    int noOfWinner = joseph(m,n);
    int dividend = 0;
    int divisor = 0;
    for (int i = 0; i < n; i ++) {
        divisor += w[i];
        if (a[i] == 1) {//good man
        dividend += w[(i + n - noOfWinner) % n];
        }
    }
    probability = 1.0*dividend/divisor;
    }
    private int joseph(int m, int n) {
    //no_in_round_i-1 = (no_in_round_i + m) % total_i-1;
    //total_i = n - i
    //no_in_round_n-1 = 0
    //thus, no_in_round_i=(no_in_round_i+1 + m) % (n - i);
    //we need to find no_in_round_0
    int no = 0;//initial:no_in_round_n-1=0
    for (int i = n - 2; i >= 0; i --) {
        no = (no + m) % (n - i);
    }
    return no;
    }
    double getProbability() {
    return probability;
    }
}
```

第二题

难点在于想到按以不同元素 numbers[i]作为最大值元素的情况对区间进行分类统计，因为可

能会有重复元素的存在，所以规定在区间内有多个值相等，均为最大值的情况下，以 index 最小的元素作为最大值元素。对于元素 numbers[i],若 numbers[s_i]是其左侧第一个大于等于它的元素，numbers[l_i]是其右侧第一个大于它的元素，则所有左端点在[s_i+1,i]，右端点在[l,l_i-1]之间的区间都是以 numbers[i]为最大值元素的区间，所有左右端点不符合这个条件的区间都不是以 numbers[i]为最大元素的区间，即以 numbers[i]为最大值元素的区间共有(i – s_i)*(l_i-i)个。因此，$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}P[i][j] = \sum_{i=0}^{n-1}arr[i] * (i - s_i) * (l_i - i)$ 可以用单调栈求 s_i 和 l_i。

代码：

```
class MaxSum {//question 2
    int sum;
    MaxSum(int[] numbers) {//count the number of intervals where numbers[i] is the maximum
    int[] firstLargerAndEqual = new int[numbers.length];
    firstLargerAndEqual[0] = -1;
    Stack<Integer> st = new Stack<>();
    st.push(0);
    long total = 0;
    for (int i = 1; i < numbers.length; i ++) {
        while (!st.isEmpty() && numbers[st.peek()] < numbers[i]) {
        st.pop();
        }
        firstLargerAndEqual[i] = (st.isEmpty())? -1:st.peek();
        st.push(i);
    }
    st = new Stack<>();
    int rightLarger = numbers.length;
    for (int i = numbers.length - 1; i >= 0; i --) {

        while (!st.isEmpty() && numbers[st.peek()] <= numbers[i]) {
        st.pop();
        }
        rightLarger = (st.isEmpty())? numbers.length:st.peek();
        total = (total + (long)(rightLarger - i)*(i - firstLargerAndEqual[i])
* numbers[i]%1000000007) %1000000007;
        st.push(i);
    }
    sum = (int) total;
    }
    int getSum() {
    return sum;
    }
}
```

第三题

用 BFS 做，矩阵 minimum[i][j]中存当前发现的到（i,j）所需要移除的最小障碍数，当发现新的仅需移除更少障碍的路径时，更新 minimum，同时在新路径的基础上继续探索到别的点所需要移除的最小障碍数（即将坐标（i,j）放入队列中），直到所有 minimum[i][j]都不再更新（队列为空）。

代码：

```
class RemoveObstacle {//question 3
    private int removed;
    RemoveObstacle(int[][] playground) {
    int m = playground.length;
    if (m == 0) return;
    int n = playground[0].length;
    int[][] minimum = new int[m][n];
    boolean[][] inqueue = new boolean[m][n];
    for (int i = 0; i < m; i ++) {
        for (int j = 0; j < n; j ++) {
        minimum[i][j] = m*n + 1;
        }
    }
    minimum[0][0] = playground[0][0];
    Queue<Integer> q = new LinkedList<>();
    q.add(0);
    inqueue[0][0] = true;
    int[][] moves = {{0,-1},{0,1},{-1,0},{1,0}};
    while (!q.isEmpty()) {
        int cur = q.poll();
        int x = cur/n;
        int y = cur % n;
        inqueue[x][y] = false;
        for (int[] move:moves) {
        int newx = x + move[0];
        int newy = y + move[1];
        if (newx >= 0 && newx < m && newy >= 0 && newy < n && minimum[x][y] +
playground[newx][newy] < minimum[newx][newy]) {
            minimum[newx][newy] = minimum[x][y] + playground[newx][newy];
            if (!inqueue[newx][newy]) {
            q.add(newx*n+newy);
            inqueue[newx][newy] = true;
            }
        }
        }
    }
    removed = minimum[m - 1][n - 1];
```

```
    }
    int getMinimumRemoved() {
    return removed;
    }
}
```

第四题

时间复杂度为 O(n^2k)的动态规划解法比较直观，dp[j][i]为将 0~i 的元素划分成 j+1 段可以得到的最大兴趣值，w[p][i]为 p~j 间不同元素个数(兴趣值)的话，则 dp[j][i] = max_{j<=p<=i} dp[j-1][p-1]+w[p][i]。dp[k-1][n-1]为最终答案。

时间复杂度为 O(nklogn)的线段树解法需要理解一下。本题的目标是要求出 dp[j][i]，令 arr_{j,i}[p]=dp[j-1][p] + w[p][i]，则根据递推关系式 dp[j][i] = max_{j<=p<=i} dp[j-1][p-1]+w[p][i] = max_{j<=p<=i}arr_{j,i}[p]，求 dp[j][i] 就是求数组 arr_{j,i}(下标（j,i）表示该数组的值会随着 j,i 的变动而变动，一对（j,i）对应一个数组)在区间[j,i]上的最大值。又因为 arr_{j,i}[p] = 1 + dp[j-1][i-1](p=i),同时，假设在 ids[i]左侧第一个与其相等的为 ids[q]，则对于 i - 1 >= p > q,arr_{j,i}[p] = dp[j-1][p] + w[p][i] = dp[j-1][p] + w[p][i-1] + 1 = arr_{j,i-1}[p] + 1, 对于 j<=p<=q,arr_{j,i}[p] = dp[j-1][p]+w[p][i] = dp[j-1][p]+w[p][i-1]=arr_{j,i-1}。因此，对每一个 j 建一棵可以对区间上各元素进行统一 update 的最大值查询线段树，i 从 j 开始遍历，对每一个 i,首先为线段树区间[i,i]上各元素（即 arr_{j,i}[i]）加上 1+dp[j-1][i-1],再为线段树区间[q+1,i-1]上各元素加上 1，然后查询[j,i]上的最大值，即为 dp[j][i]。

关于如何建可对区间各元素进行统一 update 的最大值查询线段树（建树复杂度 O(n),查询及更新复杂度 O（logn）），可以使用 lazy propagation，详情可见 https://cp-algorithms.com/data_structures/segment_tree.html#toc-tgt-10 有一系列对于线段树的讲解。

代码：（包含了简单的动态规划解法和线段树解法）

```
class FindPartition {//question 4
    private int greatestInterest = -1;
    private int dpgreatest = -1;
    private int[] ids;
    private int k;
    class SegmentTree {
    class Node {
        int l;
        int r;
        int max;
        int append;
        Node left;
        Node right;
        Node(int l, int r) {
```

```
    this.l = l;
    this.r = r;
    }
}
Node root;
SegmentTree(int l, int r){
    root = buildSegmentTree(l,r);
}
private Node buildSegmentTree(int l, int r) {
    Node root = new Node(l,r);
    if (l == r) return root;
    int m = l + (r - l)/2;
    root.left = buildSegmentTree(l,m);
    root.right = buildSegmentTree(m + 1, r);
    return root;
}
void push(Node e) {
    e.left.append += e.append;
    e.right.append += e.append;
    e.left.max += e.append;
    e.right.max += e.append;
    e.append = 0;
}
void update(int l, int r, int toappend) {
    if (l > r) return;
    update(root,l,r,toappend);
}
void update(Node root, int l, int r, int toappend) {
    if (root.l == l && root.r == r) {
    root.append += toappend;
    root.max += toappend;
    } else {
    push(root);
    int m = root.l + (root.r - root.l)/2;
    if (r <= m) {
        update(root.left,l,r,toappend);
    } else {
        if (l > m) {
        update(root.right,l,r,toappend);
        } else {
        update(root.left,l,m,toappend);
        update(root.right,m+1,r,toappend);
        }
    }
```

```
        root.max = Math.max(root.left.max, root.right.max);
        }
    }
    int findMax(int l, int r) {
        return findMax(root,l,r);
    }
    private int findMax(Node root, int l, int r) {
        if (l == root.l && root.r == r) {
        return root.max;
        }
        push(root);
        int m = root.l + (root.r - root.l)/2;
        if (r <= m) {
        return findMax(root.left,l,r);
        } else {
        if (l > m) {
            return findMax(root.right,l,r);
        } else {
            return
Math.max(findMax(root.left,l,m),findMax(root.right,m+1,r));

        }
        }
    }
    }
  public FindPartition(int[] ids, int k) {
      this.ids = ids;
      this.k = k;

  }
  public int getGreatestInterestInNKLOGN() {//O(nklogn)解法
      if (greatestInterest == -1) {
      int n = ids.length;
          if (n == 0 || k > n) {
          greatestInterest = n;
          return greatestInterest;
          }
          int[][] maxInterest = new int[k][n];
          int[] prev = new int[n];//prev index where ids[prev[i]] = ids[i]
          prev[0] = -1;
          HashMap<Integer,Integer> prevdict = new HashMap<>();
          prevdict.put(ids[0], 0);
          for (int i = 1; i < n; i ++) {
          if (prevdict.containsKey(ids[i])) {
```

```java
                prev[i] = prevdict.put(ids[i], i);
            } else {
                prev[i] = -1;
                prevdict.put(ids[i], i);
            }
            }
            maxInterest[0][0] = 1;
            for (int i = 1; i < n; i ++) {
            if (prev[i] == -1) {
                maxInterest[0][i] = maxInterest[0][i - 1] + 1;
            } else {
                maxInterest[0][i] = maxInterest[0][i - 1];
            }
            }
            for (int j = 1; j < k; j ++) {
            SegmentTree st = new SegmentTree(j,n-1);
            for (int i = j; i < n; i ++) {
                st.update(i, i, maxInterest[j - 1][i - 1] + 1);//update p == i
                st.update(Math.max(j, prev[i] + 1), i - 1, 1);//update p >= q + 
1 && p < i
                maxInterest[j][i] = st.findMax(j, i);//j<=p<=i
            }
            }
            greatestInterest = maxInterest[k - 1][n - 1];
        }
        return greatestInterest;
    }
    public int getGreatestInterestInNKN() {//O(n^2k)解法
        if (dpgreatest == -1) {
        int n = ids.length;
        if (n == 0 || k >= n) {
            dpgreatest = n;
            return n;
        }
        int[][] max = new int[k][n];
        int[][] interest = new int[n][n];
        for (int i = 0; i < n; i ++) {
            HashSet<Integer> s = new HashSet<>();
            s.add(ids[i]);
            interest[i][i] = 1;
            for (int j = i + 1; j < n; j ++) {
            if (s.contains(ids[j])) {
                interest[i][j] = interest[i][j - 1];
            } else {
```

```
                s.add(ids[j]);
                interest[i][j] = interest[i][j - 1] + 1;
            }
            }
        }
        for (int i = 0; i < n; i ++) {
            max[0][i] = interest[0][i];
        }
        for (int j = 1; j < k; j ++) {
            for (int i = j; i < n; i ++) {
            max[j][i] = interest[j][i] + j;
            for (int p = j + 1; p <= i; p ++ ) {
                max[j][i] = Math.max(max[j][i], max[j - 1][p - 1] +
interest[p][i]);
            }
            }
        }
        dpgreatest = max[k - 1][n - 1];
        }
        return dpgreatest;
    }
}
```