A. Карточная игра

(Автор задачи — *Киндер М.И.*)

В этой игре первый игрок выиграет в том случае, когда общее количество сделанных ходов будет *нечётным*. Иначе выиграет второй игрок. Для заданного набора карточек подсчитаем количество возможных ходов. Для этого подсчитаем, сколько раз входит в исходный набор та или иная карточка. Это можно сделать двумя способами.

Решение $O(n^2)$. Выберем одну из карточек и будем просматривать *все* карточки, расположенные справа от неё. Если найдём совпадающую с ней, увеличим значение счётчика на единицу. После прохода по всему массиву мы будем знать, сколько раз число входит в данный набор. Поделив его на 2, найдём количество ходов, которое можно сделать с *этим числом*. Затем перейдём к следующей карточке и так далее, пока не просмотрим все числа. Просуммировав все количества ходов, мы определим *общее число ходов*. Осталось проверить его *чётность*. Это решение проходит первые две группы тестов.

Решение $O(n)$. Определим массив *count*[] из 10^5 чисел, в котором значение *count*[*i*] равно количеству чисел, равных *i*. Это значение будем подсчитывать *сразу* при считывании чисел, записанных на карточках набора. После этого, как уже отмечалось, нужно подсчитать общее число ходов и определить его чётность.

Приведём основной фрагмент кода на языке Pascal:

```
for i := 1 to n do begin
    read(a);
    inc(count[i]);
end;
sum := 0;
for a := 1 to 100000 do
sum := sum + count[a] div 2;
write(2 – sum mod 2);
```

B. Супердвоичная система счисления

(Автор задачи — *Киндер М.И.*)

Найдём запись в супердвоичной системе счисления нескольких первых натуральных чисел. Заметим, что

- если число n — чётное, то запись числа $n = 2m$ получается из супердвоичного представления числа m «сдвигом» на один разряд (то есть добавлением нуля к записи числа m).
- Если же n — нечётно, то $a_0 = \pm 1$, и $a_1 = 0$. Цифра a_0 выбирается так, чтобы число $n - a_0$ делилось на 4. Супердвоичная запись числа $n = 4m + a_0$ получается из записи меньшего числа m «сдвигом» на два разряда и добавлением справа цифры a_0.

Во всех этих случаях единственность представления числа n следует из единственности представления меньшего числа m. (Единственность представления числа 1 очевидна.)

n	1	2	2^2	2^3	2^4
1	+				
2		+			
3	–		+		
4			+		
5	+		+		
6		–		+	
7	–			+	
8				+	
9	+			+	
10		+		+	
11	–		–		+
12			–		+
13	+		–		+
14		–			+
15	–				+

```
k := 0;
while n > 0 do begin
    inc(k);
    r := n mod 2;
    if r = 1 then a[k] := – n mod 4 + 2;
    if r = 0 then a[k] := 0;
    n := (n – a[k]) div 2;
end;
for i := k downto 1 do
    write(a[i], ' ');
```

C. Популярный рейтинг

(*Модификация фольклорной задачи.*)

Решение $O(n^2)$. Выберем одно из чисел и будем просматривать *все* числа справа от него, подсчитывая на каждом шаге количество совпадающих с ним чисел. Как только обнаружим число, которое встречается более чем $n / 2$ раз, задача решена. Таким образом, понадобится порядка n^2 операций. Это решение проходит первые две группы тестов (40 *баллов*).

Решение $O(n \log n)$. Отсортируем исходный массив чисел и будем сравнивать *соседние* числа массива. В случае их совпадения будем увеличивать значение счётчика на единицу, подсчитывая, таким образом, количество вхождений в массив данного числа. Если значение счётчика оказывается больше $n / 2$, популярный рейтинг найден. В противном случае, переходим к просмотру следующего числа. Поскольку сортировка требует $n \log n$ операций, и затем еще n операций для просмотра элементов отсортированного массива, в итоге получаем алгоритм за $O(n \log n)$ операций. Это решение проходит первые три группы тестов (70 *баллов*).

Решение $O(n)$. [*Алгоритм Боера-Мура.*] Рассмотрим первое число массива, будем считать его «претендентом» на звание популярного числа.

1. Присвоим значение 1 счётчику числа вхождений этого числа в массив.

2. Увеличим значение счетчика на 1, если очередное считываемое число, совпадает с «претендентом» и уменьшаем на 1, если не совпадает. Так делаем, пока счётчик не станет равным 0, или не закончится поток.

3. Если после какого-то шага счетчик стал = 0, следующим шагом выполняем шаг 1.

4. То число, которое является «претендентом» в момент окончания потока, и есть искомый популярный элемент.

Это решение проходит все 4 группы тестов (100 *баллов*).
Приведём основной фрагмент кода на языке Pascal:

```
for i := 1 to n do begin
    read(r);
    if counter = 0 then begin
        majority := r;
        counter := 1;
    end
    else
        if r = majority then inc(counter)
        else dec(counter);
end;
write(majority);
```

D. Шестерёночки

(Автор задачи — *Киндер М.И.* По мотивам задачи Санкт-Петербургской олимпиады 1995 г.)

Сначала составим граф, описывающий соединение шестерёнок. Будем говорить, что две шестерёнки связаны между собой ребром, если они соединены друг с другом. Если одна из шестерёнок вращается по часовой стрелке, то все соединенные с ней шестерёнки должны вращаться против часовой стрелки. Шестерёнки, соединённые с ними, — снова по часовой, следующие — против, и так далее.
Перейдем к исследованию графа соединения шестерёнок. С помощью *обхода в ширину* или *обхода в глубину* назначим метки всем вершинам графа. Выберем одну из вершин в качестве начальной и присвоим ей метку 0. Вершины, соединённые с ней ребром, получат метку 1, а вершины, соединенные с ними, — снова метку 0; они вращаются в том же направлении, что и начальная шестерёнка. При входе в очередную вершину мы помечаем, что мы в ней были, и *рекурсивно* входим во все соседние с ней вершины, в которых ещё не были. Если мы попали в вершину, в которой уже побывали, сравниваем её метку с той, которую она должна получить. Если они не совпадают, система вращаться не может, и мы выводим -1. Иначе, мы получили *компоненту связности* графа, соответствующую исходной начальной вершине. (Это подсистема шестерёнок, которые вращаются после запуска начальной шестерёнки.) Выводим на печать номер начальной шестерёнки, с которой начинали обход графа, и продолжаем просмотр непомеченных вершин, пока не пометим все вершины графа.

Оценим сложность алгоритма. Для составления и считывания графа нам понадобится $O(m)$ шагов. Для нахождения всех вершин, связанных с заданной вершиной, то есть находящихся в одной с ней компоненте связанности, понадобится $O(n)$ операций. Такое же (по порядку) количество шагов нам придётся сделать, чтобы просмотреть все остальные компоненты связанности. Таким образом, итоговая сложность алгоритма — $O(n + m)$.

Председатель жюри М.И. Киндер