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 — $O(n + m)$.

E. Сортировочная

(Автор задачи — *Киндер М.И.*)

Решение для первой группы тестов предполагает несложный перебор возможных вариантов расположения чисел и аккуратное программирование списка переставляемых пар (не более восьми шагов).

Решение для второй группы тестов. Если среди чисел исходного набора встречается единица, то переставить любые два элемента a и b, не меняя положение остальных чисел, можно за три шага:

Положение чисел:	*Переставляем пары:*
a … b … 1	a 1
1 … b … a	b 1
b … 1 … a	a 1
b … a … 1	

Теперь упорядочить числа набора можно, например, с помощью алгоритма сортировки пузырьком. Сначала найдём число 1 и поставим его на первое место. Затем найдём следующее (после 1) *наименьшее* число и поставим его на второе место указанным выше способом — с помощью трёх перестановок. При этом все остальные числа, включая 1, не меняют своего положения. Затем поставим на третье место следующее наименьшее число и так далее. Сложность этого алгоритма — $O(n^2)$. Понятно, что это решение можно улучшить, используя быструю сортировку, в этом случае его сложность будет $O(n \log n)$.

Полное решение. Рассмотрим граф, в котором вершины соответствуют данным числам, а рёбра — отношению делимости между ними. Другими словами, две вершины соединим ребром в том и только в том случае, если для соответствующих им чисел a и b или a делится на b или b делится на a. Создадим вектор-копию copy[] исходного набора и отсортируем его числа по возрастанию. Теперь легко определить, на какое место в графе нужно переместить каждое число исходного набора.

Пусть a и b — две вершины в графе, для которых соответствующие числа нужно поменять местами так, чтобы одно из этих чисел заняло требуемое место, определяемое отсортированной копией набора. С помощью обхода в ширину находим *кратчайший путь* от a до b. (Если такого пути не существует, то есть число в вершине a нельзя поставить на требуемое место b, выводим -1.) Пусть этот путь состоит из s рёбер:

$$a — v_1 — v_2 — v_3 — \ldots — v_{s-1} — b.$$

Опишем два *рекурсивных* алгоритма перестановки чисел, соответствующих вершинам a и b.

Алгоритм за $2^s - 1$ шагов. Несложно найти способы перестановки для небольших значений длины пути между вершинами. Например, если длина пути равна 1, то это можно сделать за один шаг. Если длина пути равна 2, это можно сделать за 3 шага. Предположим, что за k шагов мы умеем решать задачу перестановки для пути длиной s. Тогда в пути $a — v_1 — v_2 — v_3 — \ldots — v_{s-1} — v_s — b$ из $s + 1$ рёбер выделим путь от a до v_s, переставим числа в вершинах a и v_s (k шагов):

$$v_s \;\; v_1 — v_2 — v_3 — \ldots — v_{s-1} \;\; a \;\; b,$$

затем за 1 шаг переставим числа b и v_s:

$$b \;\; v_1 — v_2 — v_3 — \ldots — v_{s-1} \;\; a \;\; v_s.$$

Теперь мы можем ещё раз воспользоваться рекурсией и ещё за k шагов поменять числа в вершинах a и v_s пути $a — v_1 — v_2 — v_3 — \ldots — v_{s-1} — v_s$. Таким образом, после $k + 1 + k = 2k + 1$ шагов мы приходим к расположению:

$$b \;\; v_1 — v_2 — v_3 — \ldots — v_{s-1} — v_s \;\; a.$$

Несложно вывести формулу для подсчета количества шагов в общем случае. Для $s = 3$ получим $2 \cdot 3 + 1 = 7$ шагов, для $s = 4$ получим $2 \cdot 7 + 1 = 15$ шагов и так далее. Общее число шагов равно $2^s - 1$.

Алгоритм за $2s - 1$ шагов. Приведём другой рекурсивный — более эффективный — способ перестановки двух чисел, соответствующих вершинам a и b. Основная идея состоит в *циклическом* сдвиге чисел вдоль пути между вершинами. Проиллюстрируем её на примере пути длины $s = 3$: $\boldsymbol{a} — v_1 — v_2 — \boldsymbol{b}$. Сначала за 2 шага осуществим циклический сдвиг вправо первых трёх элементов (от $a — v_1 — v_2$ придём к $v_2 \;\; a — v_1$):

Положение чисел:	*Переставляем пары:*
$\boldsymbol{a} — \boldsymbol{v_1} — \boldsymbol{v_2} — b$	$\boldsymbol{a} \;\; \boldsymbol{v_1}$
$\boldsymbol{v_1} — \boldsymbol{a} \;\; \boldsymbol{v_2} — b$	$\boldsymbol{v_1} \;\; \boldsymbol{v_2}$
$\boldsymbol{v_2} \;\; \boldsymbol{a} — \boldsymbol{v_1} \;\; b$	

Теперь поменяем местами числа в вершинах v_2 и b: $b \;\; \boldsymbol{a} — \boldsymbol{v_1} — \boldsymbol{v_2}$, и затем ещё за 2 шага осуществим циклический сдвиг *влево* чисел вдоль пути $\boldsymbol{a} — \boldsymbol{v_1} — \boldsymbol{v_2}$, то есть получим расположение $b \;\; \boldsymbol{v_1} — \boldsymbol{v_2} \;\; \boldsymbol{a}$, общее число шагов $2 + 1 + 2 = 2 \cdot s - 1$.

Оценим сложность алгоритма. Сортировка исходного набора требует $O(n \log n)$ операций перестановок. Каждая перестановка двух элементов требует порядка $O(s) = O(n)$ шагов, поэтому итоговая сложность — $O(n^2 \log n)$.

Председатель жюри М.И. Киндер