

```

1 algorithm1:
2   c = 0
3   ind = -1
4
5   for i = 0 to n
6     c1 = 0
7     for j = 0 to n
8       if A[i] = A[j]
9         c1 = c1 + 1
10    if c1 > c
11      c = c1
12      ind = i
13
14  if c > n / 2 return A[ind]

```

асимптотика  
 $O(n^2)$

$n$

Алгоритм 1 ищет элемент массива, который встречается в массиве A более  $\frac{n}{2}$  раз.

Алгоритм проходится по каждому элементу и считает, сколько раз он совпадает с другими элементами. Для этого рассматриваются возможные пары, из-за чего асимптотика равна  $O(n^2)$ . Если данный элемент встречался чаще, чем предыдущий (т.е.  $c_1 > c$ ), то индекс и значение наибольших совпадений (переменные  $C$ ) перезаписываются. Если  $c > \frac{n}{2}$ , то возвращается элемент, иначе ничего. Т.к. если эл. встречается  $> \frac{n}{2}$  раз, то другого эл. с большим кол-вом быть не может.

```

1 algorithm2:
2   c = 1, ind = 0
3
4   for i = 1 to n
5     if A[ind] = A[i]
6       c = c + 1
7     else
8       c = c - 1
9
10    if c = 0
11      ind = i
12      c = 1
13
14  return A[ind]

```

асимптотика  
 $O(n)$

$n$

Алгоритм 2 проходит по элементам и сравнивает их с элементом  $A[ind]$ . Если они равны, то переменная  $C$  увеличивается на 1, иначе уменьшается на 1. Дальше, если  $C=0$ , т.е. если среди рассмотренных элементов кол-во равных и неравных одинаково, то  $ind$  меняется на  $i$ , а  $C$  сбрасывается до 1. Таким образом, алгоритм возвращает последний элемент, после которого одинаковое кол-во равных и неравных элементов.

```

1 algorithm3:
2   if n = 1
3     return A[0]
4
5   c = 1
6   sort(A)
7
8   for i = 1 to n
9     if A[i - 1] = A[i]
10      c = c + 1
11    else
12      if c > n / 2
13        return A[i - 1]
14      c = 1

```

асимптотика  
 $O(\text{sort}(n))$

$\text{sort}(n)$

$n$

Алгоритм 3 сортирует список, а дальше проходится по элементам. Если элемент равен предыдущему, то переменная  $C$  увеличивается на 1. Иначе, если переменная  $C > \frac{n}{2}$ , возвращается элемент, который встретился  $> \frac{n}{2}$  раз, и  $C$  сбрасывается до 1. Таким образом, находим элемент, который встретился  $> \frac{n}{2}$  раз.

**1.** Не согласен. Алгоритмы 1 и 3 работают похоже — они возвращают элемент, встретившийся  $>\frac{n}{2}$  раз. Алгоритм 2 возвращает последний элемент, после которого однократное кол-во разных и первых ему элементов. Более подробно каждый алгоритм описан правее них на первой странице.  
 Алгоритм 1 наиболее корректный. Проблема алгоритма 3 в том, что если нужный элемент самый большой, то он не вернётся, т.к. условием выхода является то, что необходим следующий элемент, который не будет равен текущему. Идея алгоритма 2 несовсем верная.

Примеры входных данных:

№	ПРИМЕР	1	2	3
1	$A = \{1, 2, 2, 5, 2\}$	2	2	2
2	$A = \{1, 3, 3, 3, 2, 2, 3\}$	3	3	-
3	$A = \{1, 2, 3, 4, 5, 6\}$	-	6	-

Пример 1: Алгоритм 1 (A1) находит элемент 2, который встретился 3 раза ( $>\frac{5}{2}$  раз). A2 на I итерации получает  $c=0$ , т.к.  $1 \neq 2 \Rightarrow \text{ind}=1$ , а дальше  $c$  увеличивается 2 раза и уменьшается 1 раз, но  $c \neq 0$ . А тогда возвращается  $A[1]=2$ . A3 возвращает 2, т.к. встречается 3 раза ( $>\frac{5}{2}$ ), а также есть элемент  $\geq h$ .

Пример 2: A1 возвращает 3, т.к. этот элемент встречается 4 раза ( $>\frac{7}{2}$ ). A2 возвращает 3, т.к. на I итерации  $c=0$  и  $\text{ind}=1$ . Дальше, аналогично примеру 1,  $c$  не будет равна 0, поэтому вернётся  $A[1]=3$ . A3 ничего не вернёт, несмотря на то, что 3 встретился  $4 > \frac{7}{2}$  раза. Но т.к. после 7 нет элемента  $\neq 7$ , то в условие выхода алгоритм не попадёт.

Пример 3: A1 и A3 ничего не вернут, т.к. нет элементов, встречающихся  $>\frac{n}{2}$  раз. A2 вернёт 6, т.к. на последней итерации  $c$  станет равно 0.

**2.** Временная сложность A1 —  $O(n^2)$ , т.к. есть внешний цикл на  $n$  итераций и внутренний на  $n$  итераций, в которых содержатся операции за  $O(1)$ . В итоге получает  $O(n^2)$ .

Временная сложность А2 -  $O(n)$ , т.к. есть 1 цикл на  $n$  итераций, в которых есть только операции за  $O(1)$ .

Временная сложность А3 -  $O(\text{sort}(n))$ . Сортировка выполнится за  $O(\text{sort}(n))$ , а в цикле есть  $n$  итераций, в которых содержатся операции за  $O(1)$ , т.е. цикл выполнится за  $O(n)$ . Следовательно, сортировка займет большее время, т.к. она выполнится не раньше, чем за цикл.

3. А1 работает хорошо, поэтому его не меняем. Единственное - в конце можно добавить `return -1`, если нужный элемент не найден.
4. Асимптотика не поменяется никак.

```
5  template<typename T>
6  int algorithm2(std::vector<T>& A) {
7      int n = A.size();
8
9      int c = 1;
10     int ind = 0;
11
12     for (int i = 1; i < n; ++i) {
13         if (A[ind] == A[i]) {
14             c = c + 1;
15         } else {
16             c = c - 1;
17         }
18
19         if (c == 0) {
20             ind = i;
21             c = 1;
22         }
23     }
24
25     int res = 0;
26     for (int i = 0; i < n; ++i) {
27         if (A[ind] == A[i]) {
28             ++res;
29         }
30     }
31     if (res > n / 2) {
32         return A[ind];
33     }
34     return -1;
35 }
```

```
38  template<typename T>
39  int algorithm3(std::vector<T> A) {
40      int n = A.size();
41
42      if (n == 1) {
43          return A[0];
44      }
45
46      int c = 1;
47      std::sort(first: A.begin(), last: A.end());
48
49      for (int i = 1; i < n; ++i) {
50          if (A[i - 1] == A[i]) {
51              c = c + 1;
52          } else {
53              if (c > n / 2) {
54                  return A[i - 1];
55              }
56              c = 1;
57          }
58      }
59      if (c > n / 2) {
60          return A[n - 1];
61      }
62      return -1;
63 }
```

В А2 добавляем проверку, что найденный элемент действительно встречается  $> \frac{n}{2}$  раз. Для этого пройдемся по циклу и посчитаем кол-во + проверка `if`. Асимптотика как была  $O(n)$ , так и осталась, т.к. добавился лишь цикл за  $O(n)$ .

В А3 добавим проверку после цикла, что последний элемент является искомым. Асимптотика не поменялась, т.к. добавлено действие за  $O(1)$ .