Задача A4. Разные алгоритмы решения одной* задачи

Даны три алгоритма, в рамках которых выполняется обработка целочисленного массива A, содержащего n элементов, а sort(A) соответствует сортировке массива некоторым способом:

```
algorithm1:
  c = 0
  ind = -1

  for i = 0 to n
      c1 = 0
      for j = 0 to n
          if A[i] = A[j]
              c1 = c1 + 1
      if c1 > c
          c = c1
          ind = i

  if c > n / 2 return A[ind]
```

```
algorithm2:
  c = 1, ind = 0

  for i = 1 to n
      if A[ind] = A[i]
          c = c + 1
      else
          c = c - 1

      if c = 0
          ind = i
          c = 1

  return A[ind]
```

```
algorithm3:
  if n = 1
      return A[0]

  c = 1
  sort(A)

  for i = 1 to n
      if A[i - 1] = A[i]
          c = c + 1
      else
          if c > n / 2
              return A[i - 1]
          c = 1
```

1. Утверждается, что представленные алгоритмы должны решать одну и ту же задачу, т. е., выдавать один и тот же ответ на одинаковых входных данных.
• Согласны ли вы с этим утверждением? Результаты работы каких алгоритмов из представленных могут отличаться? Какую задачу решает каждый алгоритм?
• Приведите примеры входных данных, при которых результаты работы алгоритмов могут отличаться, а также совпадать. Поясните свой ответ с помощью трассировки (частичной) работы алгоритмов.

Не согласен. Алгоритмы 1 и 3 решают задачу поиска элемента, который встречается в массиве более чем n/2 раз, и возвращают этот элемент только если он существует. Алгоритм 2, однако, всегда возвращает некоторый элемент из массива, независимо от того, является ли он встречающимся более чем n/2 раз, так как не выполняет проверку наличия большинства.

Отличаться могут результаты алгоритма 2 по сравнению с алгоритмами 1 и 3.

Алгоритм 1 перебирает все элементы и подсчитывает количество появлений каждого, возвращая элемент, если он встречается более чем n/2 раз.

Алгоритм 2 пытается найти кандидата на элемент, который может встречаться более чем n/2 раз, используя механизм увеличения и уменьшения счётчика во время обхода массива, но не проверяет, действительно

ли этот кандидат появляется более чем n/2 раз.

Алгоритм 3 сначала сортирует массив и затем ищет элемент, который появляется более чем n/2 раз подряд, возвращая его, если такой найден.

Примеры входных данных и пояснения:

Пример, когда результаты отличаются:
Входной массив: A = [1, 2, 3, 4], (n = 4)
Алгоритм 1: Перебирает все элементы, находит, что максимальное количество появлений любого элемента равно 1. Так как 1 <= n/2, алгоритм не возвращает значение.
Алгоритм 2: Инициализирует c = 1, ind = 0. Проходит по массиву, обновляя ind и c. В конце возвращает A[ind], что равно 4.
Алгоритм 3: Сортирует массив: [1, 2, 3, 4]. Перебирает элементы, но ни один не встречается более чем n/2 раз подряд. Алгоритм не возвращает значение.
Итог: Алгоритм 2 возвращает 4, тогда как алгоритмы 1 и 3 не возвращают ничего.

Пример, когда результаты совпадают:
Входной массив: A = [2, 2, 1, 2, 2, 3, 2] (n = 7)
Алгоритм 1: Находит, что число 2 встречается 5 раз. Так как 5 > n/2, возвращает 2.
Алгоритм 2: После прохождения массива, ind указывает на элемент со значением 2. Возвращает 2.
Алгоритм 3: Сортирует массив: [1, 2, 2, 2, 2, 2, 3]. Находит, что число 2 встречается 5 раз подряд. Возвращает 2.
Итог: Все три алгоритма возвращают 2.

Пояснение:
Алгоритм 1 точно определяет элемент встречающийся более чем n/2 раз путем полного перебора и подсчета.
Алгоритм 2 находит кандидата на элемент встречающийся более чем n/2 раз, но без окончательной проверки; поэтому может вернуть неверный результат, если элемента встречающигося более чем n/2 раз нет.
Алгоритм 3 использует сортировку для упрощения поиска элемента встречающимся более чем n/2 раз, проверяя последовательные элементы.
Таким образом, алгоритмы 1 и 3 решают задачу поиска элемента

встречающимся более чем n/2 раз с проверкой его существования, тогда как алгоритм 2 может вернуть кандидата без гарантии его большинства.

2. Вычислите асимптотическую верхнюю границу $O(f(n))$ временной сложности для каждого алгоритма. Обоснуйте свой ответ. Представлять полный расчет точного выражения функции временной сложности $T(n)$ не нужно.

Алгоритм 1 анализ:

Внешний цикл выполняется для каждого i от 0 до n-1, то есть n итераций.

Внутренний цикл для каждого i внутренний цикл по j от 0 до n-1, то есть n итераций.

Операции внутри внутренних циклов: сравнение и инкремент — операции O(1).

Временная сложность:

Общее число операций: $n \times n = n^2$.

Итого временная сложность: $O(n^2)$.

Алгоритм 2 анализ:

Цикл выполняется для каждого i от 1 до n-1, то есть n-1 итераций.

Операции внутри цикла: сравнение, инкремент/декремент, проверка и возможное обновление ind и c — все O(1) операций.

Временная сложность:

Общее число операций: пропорционально n.

Итого временная сложность: $O(n)$.

Алгоритм 3 анализ:

Сортировка массива занимает $O(n \log n)$ времени с использованием эффективных алгоритмов сортировки (например, быстрая сортировка, сортировка слиянием).

Цикл после сортировки выполняется для каждого i от 1 до n-1, то есть n-1 итераций.

Операции внутри цикла: сравнение и инкремент/сброс счётчика — операции O(1).

Временная сложность:

Сортировка: $O(n \log n)$.

Цикл после сортировки: $O(n)$.

Итого временная сложность: $O(n \log n)$, так как сортировка доминирует.

Обоснование:

Алгоритм 1 использует вложенные циклы по n итераций каждый, что приводит к квадратичной сложности.

Алгоритм 2 выполняет единственный проход по массиву с постоянными операциями внутри цикла, что даёт линейную сложность.

Алгоритм 3 включает этап сортировки, который требует $O(n \log n)$ времени, после чего следует линейный проход по массиву.

Вывод:

Алгоритм 1 неэффективен для больших массивов из-за квадратичной временной сложности $O(n^2)$.

Алгоритм 2 является самым эффективным с линейной временной сложностью $O(n)$.

Алгоритм 3 имеет промежуточную сложность $O(n \log n)$ из-за этапа сортировки.

3. Какие алгоритмы и каким образом необходимо доработать, чтобы в результате все представленные алгоритмы решали одну и ту же задачу? В ответе представьте инструкции, которые необходимо добавить или удалить. *Изменять представленные алгоритмы полностью не предполагается.*

Чтобы все три алгоритма решали одну и ту же задачу и выдавали одинаковые результаты на одинаковых входных данных, необходимо внести изменения в алгоритмы, чтобы они все находили элемент, который встречается в массиве более чем n/2 раз, и возвращали его только если он действительно является им. Если такого элемента нет, алгоритмы не должны возвращать значение.

Алгоритм 2:

Алгоритм пытается найти кандидата на элемент, который может встречаться более чем n/2 раз, используя механизм увеличения и уменьшения счётчика во время обхода массива, но не проверяет, действительно ли этот кандидат появляется более чем n/2 раз. Чтобы исправить это, необходимо добавить этап верификации после основного цикла.

Добавить проверку после основного цикла, которая подсчитывает количество вхождений кандидата и проверяет, превышает ли оно n/2.

Доработанный алгоритм 2:

```
c = 1
ind = 0

for i = 1 to n - 1
    if A[ind] = A[i]
        c = c + 1
    else
        c = c - 1

    if c = 0
        ind = i
        c = 1

// Добавляем этап верификации
count = 0
for i = 0 to n - 1
    if A[i] = A[ind]
        count = count + 1

if count > n / 2
    return A[ind]
// Иначе, элемента встречающимся более чем n/2 раз нет
```

Алгоритм 3:

Алгоритм 3 после сортировки массива ищет последовательные элементы и возвращает элемент, если он встречается более чем n/2 раз подряд. Однако, если мажоритарный элемент находится в конце массива, текущая реализация может его пропустить, так как проверка if c > n / 2 происходит только внутри цикла при смене значения.

Добавить финальную проверку после цикла, чтобы учесть возможность мажоритарного элемента в конце массива.

Доработанный алгоритм 3:

```
if n = 1
    return A[0]

c = 1
sort(A)

for i = 1 to n - 1
    if A[i - 1] = A[i]
        c = c + 1
    else
        if c > n / 2
            return A[i - 1]
        c = 1

// Добавляем финальную проверку после цикла
if c > n / 2
    return A[n - 1]
// Иначе, элемента встречающегося более чем n/2 раза нет.
```

Алгоритм 1:

Алгоритм 1 уже корректно определяет элемент, который может встречаться более чем n/2 раз и не требует доработок.

4. Докажите, что представленные вами доработки не ухудшают ранее вычисленные в п. 2 асимптотические верхние границы временной сложности. Представьте расчеты асимптотических верхних границ сложности доработанных алгоритмов.

Доказательство сохранения временной сложности:

Доработанный алгоритм 2:

Изначальная временная сложность: $O(n)$

Анализ доработок:

Основной цикл выполняется $n-1$ раз, временная сложность $O(n)$.

Этап верификации: дополнительный цикл от $i = 0$ до $n-1$, временная сложность $O(n)$.

Операции внутри цикла: сравнение и инкремент — $O(1)$ на итерацию.

Общая временная сложность:

Суммарно: $O(n) + O(n) = O(2n)$

Асимптотически: $O(n)$

Временная сложность осталась $O(n)$, то есть доработка не ухудшила временную сложность алгоритма.

Доработанный алгоритм 3:
Изначальная временная сложность: $O(n \log n)$

Анализ доработок:
Сортировка массива: $O(n \log n)$
Основной цикл выполняется $n - 1$ раз, временная сложность $O(n)$.
Финальная проверка после цикла: Операция сравнения и возможный возврат значения — $O(1)$.

Общая временная сложность:
Суммарно: $O(n \log n) + O(n) + O(1)$
Асимптотически: $O(n \log n)$ (так как $O(n \log n)$ доминирует)

Временная сложность осталась $O(n \log n)$, доработка не повлияла на асимптотическую сложность.

Таким образом, внесенные доработки не ухудшили асимптотические верхние границы временной сложности алгоритмов.