1. Алгоритм[1] получает на вход число n (в десятичной записи) и создаёт массив $A[2,\ldots,n]$, заполненный нулями. Далее алгоритм выполняет следующую процедуру, пока массив не окажется заполнен единицами. Идёт по массиву от 2 до n пока не встретит первый ноль. Пусть ноль оказался в ячейке с номером k. Тогда алгоритм выводит k и заполняет все ячейки с номерами, кратными k, единицами: идёт по массиву дальше с шагом один и через каждые k клеток записывает в ячейку единицу.

1. Какую последовательность чисел выводит алгоритм?

Последовательность простых чисел, не превосходящих n

2. Оцените временную сложность алгоритма.

$\theta(nlogn)$, n**, поскольку алгоритм ровно 1 раз проходится по массиву, и** $logn$**, поскольку нам на каждом простом числе надо будет записывать 1 в кратные** k **ячейки.**

3. Является ли алгоритм полиномиальным?

Является, поскольку $\exists$ полином p степени 2, что алгоритм при любых входных данных будет работать не более чем за (nlogn)

2. Докажите, что для произвольной константы $c > 0$ функция $g(n) = 1 + c + c^2 + \ldots + c^n$ есть

1 $\Theta(1)$, если $c < 1$;

2 $\Theta(n)$, если $c = 1$;

3 $\Theta(c^n)$, если $c > 1$.

1) В таком случае $g(n) = \frac{1}{1-c} = \Theta(1)$
2) В этом случае $g(n) = 1 \cdot (n+1) = n + 1 = \Theta(n)$
3) В этом случае $g(n) > c^n$ **и** $g(n) < 2 \cdot c^{2n} \Rightarrow g(n) = \Theta(c^n)$

3. Верно ли, что **а)** $n = O(n \log n)$? - **Нет, т.к.** $n = O(n)$. **б)** $\exists \varepsilon > 0 : n \log n = \Omega(n^{1+\varepsilon})$?

- **Нет, т.к.** $\forall k > 0 \lim_{n\to\infty} \frac{ln(n)}{n^k} = 0$ **(доказывается, к примеру, с помощью правила Лопиталя)**

4. Известно, что $f(n) = O(n^2), g(n) = \Omega(1), g(n) = O(n)$. Положим

$$h(n) = \frac{f(n)}{g(n)}.$$

1. Возможно ли, что **а)** $h(n) = \Theta(n \log n)$ - **Да**; **б)** $h(n) = \Theta(n^3)$? – **нет**

Доказательство для всех трех случаев: из исходных данных следует, что $O(n) \leqslant h(n) \leqslant O(n^2) \Rightarrow \Theta(n) \leqslant h(n) \leqslant \Theta(n^2)$

[1] Алгоритм в этой задаче можно оптимизировать, но это не должно вас смущать. В нашем курсе (как и стандартно в математических курсах) нужно решать сформулированную задачу, поэтому, пожалуйста, удержитесь от порывов оптимизировать алгоритм и оценивать его сложность, если этого не требуется в условии.

2. Приведите наилучшие (из возможных) верхние и нижние оценки на функцию $h(n)$ и приведите пример функций $f(n)$ и $g(n)$ для которых ваши оценки на $h(n)$ достигаются.

$h(n) = \Omega(n) = O(n^2)$

examples:

1) $f(n) = n^2, g(n) = \frac{n+4}{n-7}$
2) $f(n) = n^2, g(n) = \frac{n^2+1}{n+2}$

5. Дана программа

```
for (bound = 1; bound < n; bound *= 2 ) {
   for (i = 0; i < bound; i += 1) {
      for (j = 0; j < n; j += 2)
         печать ("алгоритм")
      for (j = 1; j < n; j *= 2)
         печать ("алгоритм")
   }
}
```

Пусть $g(n)$ обозначает число слов "алгоритм", которые напечатает соответствующая программа. Найдите Θ–асимптотику $g(n)$.

$g(n) = \Theta(nlogn(n + logn)) = \Theta(n^2 logn)$

Пояснения: входим в цикл с экспоненциальным ростом → $logn$.
Далее в него вложен цикл с шагом 1 → умножаем на n.
Далее в него вложено 2 цикла, один из которых идет с шагом 2 (отсюда n), а другой – с шагом, умножающимся на 2 (отсюда $logn$). Поскольку циклы идут последовательно, то их сложность просто складывается.

Во всех задачах ниже мы полагаем, что арифметические операции стоят $O(1)$.

6 [Шень 1.3.1 (а,б,г)]. Постройте линейный по времени онлайн-алгоритм, который вычисляет следующие функции или укажите индуктивные расширения для следующих функций:

а) среднее арифметическое последовательности чисел;

int avg = 0; – среднее арифметическое
int counter = 0; – количество считанных чисел
while (cin » number) {
avg = avg * counter; – умножаем текущее среднее арифметическое на количество чисел.
counter++; – инкрементируем количество чисел.
avg = (avg + number) / counter; – получаем новое среднее арифметическое.
cout « avg; – выводим на экран.
}

б) число элементов последовательности целых чисел, равных её максимальному элементу;

int max = absolute minimum for int number; – максимальный элемент
int counter = 1; – количество чисел, равных максимальному элементу

while (cin » number) {
if (number > max) { – если нашлось число больше максимума, то записываем его в max и делаем счетчик равным 1.
max = number;
counter = 1;
}
else if (number == max) { – если число равно максимуму, инкрементируем счетчик
counter++;
}
cout « counter;
}

в) максимальное число идущих подряд одинаковых элементов;
int current = nan; – текущие подряд идущие элементы (или 1 элемент)
int counter = 1; – количество подряд идущих одинаковых элементов.
int max = 1; – максимальное количество подряд идущих одинаковых элементов.
while (cin » number) {
if (number == current) counter++; – если одинаковые числа подряд, то инкрементируем счетчик.
else { – в противном случае приравниваем счетчик к 1 и обновляем текущий элемент.
counter = 1;
current = number;
}
if (counter > max) max = counter;
cout « max;
}

7. Дано три отсортированных по возрастанию массива, внутри каждого массива все элементы различные. Предложите[2] линейный алгоритм нахождения числа различных элементов в объединении массивов.
array[p] first;
array[q] second;
array[r] third;
int counter = p + q + r;
int min array elem = 0;
int i = 0, j = 0, k = 0;
if (first[i] == second[j]) {
counter–;
j++;
}
if (first[i] == third[k]) {
counter–;
k++;

[2]Здесь и всюду далее мы требуем не только описание алгоритма, но и доказательство его корректности, а также доказательство оценок на время работы алгоритма.

```
}
while (хотя бы номера не +inf) {
    min array elem = min number(first[i], second[j], third[k]);
    switch(min array elem) {
        case 1:
            if (first[i] == second[j]) counter–;
            if (first[i] == third[k]) counter–;
            if (i == p) first[i] = +inf;
            else i++;
            break;
        case 2:
            if (second[j] == first[i]) counter–;
            if (second[j] == third[k]) counter–;
            if (j == q) second[j] = +inf;
            else j++;
            break;
        case 3:
            if (third[k] == first[i]) counter–;
            if (third[k] == second[j]) counter–;
            if (k == r) third[k] = +inf;
            else k++;
            break;
    }
}
```

8 [Шень 1.3.4]. Дана последовательность целых чисел $a_1, a_2, \ldots, a_n$. Необходимо найти её самую длинную строго возрастающую подпоследовательность. Предложите **а)** $O(n^2)$ алгоритм (докажите его корректность и асимптотику); **б)** $O(n \log n)$ алгоритм.

а)

int[n] sizes – массив максимальных длин возрастающей подпоследовательности от 0-ого элемента до элемента с номером, равным индексу массива.

```
for (int i = 0; i < n; ++i) {
    int current = a[i];
    for (int j = i - 1; j >= 0; −−j) {
        if (a[j] < current) {
            sizes[i]++; – заполняем созданный нами массив.
            current = a[j];
        }
    }
}
```

int max = max index(sizes); – индекс элемента с максимально возможной возрастающей подпоследовательностью.

int subseq size = sizes[max]; – длина искомой подпоследовательности.

result[subseq size−−] = a[max]; – массив с членами искомой подпоследовательности.

```
for (int i = max - 1; i >= 0; − − i) {
```

```
        if (sizes[i] == subseq size) {
            result[subseq size−−] = a[i];
        }
    }
```

Доказательство корректности тривиально: мы находим номер члена последовательности, на котором завершается максимально возможная возрастающая подпоследовательность, после чего, начиная с конца, запоминаем члены искомой последовательности так, чтобы до i-того члена подпоследовательности можно было построить возрастающую подпоследовательность с максимальной длиной i.

Асимптотика доказывается наличием двух циклов for с максимальной длиной n, один из которых вложен в другой.

б)

```
int max = 1;
for (int i = 1; i < n; ++i) {
    int j = 0;
    for (; j < max + 1; ++j) {
        int q = j + (max - j) / 2;
        if (a[i] <= tmp[q]) max = s;
        else j = q;
    }
    if (j == max) {
        max++;
        tmp[max + 1] = a[i];
    }
    else tmp[j + 1] = a[i];
}
```

9*. На вход подаётся последовательность натуральных чисел $x_1, \dots x_n$ в которой один из элементов встречается строго больше, чем $\frac{n}{2}$ раз. Постройте алгоритм, который находит этот элемент, и при этом может использовать в качестве внешней памяти только стек (в который можно помещать только элементы последовательности), операции со стеком стоят $O(1)$ времени; в оперативной памяти программа использует $O(1)$ битов памяти и $O(1)$ регистров (в каждом из которых может храниться число x_i).

Числа x_i идут потоком данных на вход и каждое доступно для считывания только один раз — вернуться обратиться к прочитанным ранее числам можно, только если сохранить их в памяти.

```
int current number = 0; – текущее число
int popular number = 0; – самое популярное число (которое мы ищем)
cin » current number;
popular number = current number;
int count = 1; – количество самых популярных чисел
while (cin » current number) {
    if (current number == popular number) count++;
    else count−−;
    if (count == 0) {
        popular number = current number;
```

```
        count = 1;
    }
}
cout « popular number;
```

P.S. Сложность по времени – $\Theta(n)$, т.к. один раз проходим по последовательности. Сложность по памяти – $O(1)$, т.к. использовали только 3 регистра.

Более оптимального алгоритма (по асимптотике) не может существовать, поскольку все данные у нас подаются последовательно и нам надо считать минимум $\lceil \frac{n}{2} \rceil$ элементов для ответа, поэтому сомнений быть не должно.

Корректность алгоритма вполне тривиальна: если мы считали искомый элемент, то мы добавляем к "счетчику этого элемента"$+1$ (то есть либо $+1$, если текущий популярный элемент он и есть, либо -1, если в данный момент времени "самым частым"является другой элемент). Таким образом, поскольку "популярный"элемент встречается чаще чем в половине случаев, то его счетчик будет $\geqslant 1 \Rightarrow$ именно он и будет выведен на экран.