Домашнее задание №1

Томинин Ярослав, 778 группа

3 марта 2018 г.

1.
a)Для этого докажем, что

$$\exists C, N \ \forall n > N : n < Cn \lg n$$

Возьмем $C = 1$, а $N = 10$
Так как $\ln n > 1$, если $n > 10 \Rightarrow n < n \lg n$
b) Переформулируем задачу

$$\exists \varepsilon > 0 : \exists C, N \ \forall n > N : Cn \lg n > n^{1+\varepsilon}$$

Если мы поделим обе части на n

$$C \lg n > n^{\varepsilon}$$

Если мы докажем, что

$$\forall \varepsilon > 0 : \forall C, N \ \exists n > N : C \lg n < n^{\varepsilon} \tag{1}$$

То наше предположение окажется неверным
Поймем, что степенная функция с положительным показателем возрастает быстрее лагарифмической, поэтому мы сможем найти такое n, начиная с которого $n^{\varepsilon} > \lg n$ Это означает, что мы доказали (**??**)
2.
1)
a)Да, может. Приведем пример
Пусть

$$f(n) = n \lg n$$

$$g(n) = 1$$

Тогда

$$f(n) = O(n^2)$$

$$g(n) = \Omega(1)$$

$$g(n) = O(n)$$

Поэтому $h(n) = n \lg n \Rightarrow h(n) = \theta(n \lg n)$
b) Нет, не может. Поймем, что

$$h(n) = \frac{O(n^2)}{\Omega(1)} \Rightarrow h(n) \leq Cn^2 \Rightarrow h(n) = O(n^2)$$

А мы уже умеем доказывать, что

$$\theta(n^3) \neq O(n^2)$$

2)
В первом пункте мы доказали, что

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

Если же

$$f(n) = n^2$$

$$g(n) = 1$$

То $h(n) = O(n^2)$
Сейчас приведем нижнюю оценку. Докажем, что $\exists h(n) = \Omega(0)$ Допустим $f(n) = 0, g(n) = 1 \Rightarrow h(n) = \Omega(0)$
3. a) Алгоритм: В одно число(s) будем записывать текущую сумму, а в число k количество обработанных чисел на данный момент

```
for (i=0;i<n;++i)
{
    s+=scanf();
    ++k;
}
```

$$m = \frac{s}{k}$$

Этот алгоритм коректен, потому что при любом n после цикла for в s будет записана сумма, а в k-количество чисел.
Посчитаем сложность алгоритма. Обозначим за c_1 время прибавления к s, а за c_2 время прибавления к k. Легко посчитать, что $f(n) = \Theta(n)$
b)

```
max=scanf();
k=0;
for (i=0;i<n-1;++i)
{
    a=scanf();
    if(max==a)
    {
        ++k;
    }
    else if(a > max)
    {
        k=1;
        max=a;
    }
}
```

Этот алгоритм коректен, потому что счетчик k обнуляется сразу же, если находится число, большее всех прочитанных, а k в любой момент содержит количество на данный момент max чисел.
Посчитаем сложность алгоритма. Обозначим за c_1 время прибавления к k, а за c_2 время присваивания значения m, за c_3 время считывания числа. Тогда сумарная сложность $f(n) = \Theta(n)$
в)

```
k=0;
d=0;
e=scanf();
for (i=0;i<n-1;++i)
{
    a=e;
    e=scanf();
    if(a==e)
    {
        ++k;
    }
    else
    {
        if(k>d)
        {
            d=k;
            k=1;
        }
        k=1;
    }
    if(k>d)
    {
        d=k;
        k=1;
    }
}
```

Этот алгоритм коректен, потому что в каждый момент в a хранится предыдущее прочитанное число, а в e вновь прочитанное,если же они равны, то счетчик увеличивает значение, иначе мы его обнуляем, но после каждого изменения k мы сравниваем его с d-числом,содержащим max последовательность одинаковых чисел, считанных на данных момент
Обозначим за c_1 время прибавления к k, за c_2 время присваивания значения a, за c_3 время считывания числа e, за c_4 время присваивания значения d. Каждая операция повторяется не более 2n раз. $\Rightarrow f(n) = \Theta(n)$

4.
Выберем первые числа в 1, 2, 3 массивах, положим их в переменные a, b, c сооитветственно. Сравним эти переменные, если есть только одно min значение, то добавим к флажку k еденицу, а значение переменной, в которую было записано min число, изменим на следующее число в этом массиве. Если есть хотя бы два min, то увеличиваем флажок на один и меняем все переменные(выбираем следующее число в массиве), которые содержали min значение. Если в какой-то момент значения в массиве закончились, то мы перестаем сравнивать эту переменную. Так делаем до тех пор, пока не переберем все значения. Докажем коректность этого алгоритма. Допустим, что какое-то число не было посчитано, но мы знаем, что в какой-то момент это число было в переменной и если мы поменяли значение этой переменной, то это означает, что мы учли это число.
Теперь допустим, что мы посчитали одно и то же число несколько раз.(Обозначим эти числа за a_1 и a_2) Если мы посчитали их несколько раз, то это были разные сравнения, так как в одном сравнении мы сразу выкидываем минимальные одинаковые числа и меняем флажок на $+1$. Тогда это означает, что числа a_1 и a_2 были выкинуты в разных сравнениях. Обозначим эти сравнения как i и i+j. Не нарушая общности предположим, что в i мы выкинули a_1. Тогда это означает, что переменная, которая отвечает за массив, которому принадлежит a_2, была меньше, чем $a_1 \Rightarrow$ мы не могли выкинуть a_1. Противоречие. Так как мы доказали, что каждое число мы посчитали хотя бы 1 раз и каждое число мы посчитали не более 1 раза $\Rightarrow$ каждое число мы посчитали ровно 1 раз.
Сложность алгоритма по времени: время сравнения трех чисел мы обозначим за c_1(операция проведена не более, чем n раз), время присваивания значения переменной c_2(операция проведена не более, чем n раз), время прибавления к числу k еденицы c_3(операция проведена не более, чем n раз). $\Rightarrow f(n) = \Theta(n)$
Сложность алгоритма по памяти: В нашем алгоритме в памяти хранятся массивы(занимают $c_1 n$), так же у нас есть 4 переменные.$\Rightarrow f(n) = \Theta(n)$
5.
a) Алгоритм:

```
int d[n];
int f[n];
int i,j,k,z;
for(i=0;i<n;++i)
{
    d[i]=1;
    f[i]=-1;
    for(j=0;j<i;++j)
```

```
    {
        if(a[j]<a[i])
        {
            if(d[i]<1+d[j])
            {
                d[i]=1+d[j];
                p[i]=j;
            }
        }
    }
}
for(i=0;i<n;++i)
{
    if(d[i]>k);
    {
        k=d[i];
        z=i;
    }
}
int MAX[k];
i=0;
while(z!=-1)
{
    MAX[k-i-1]=a[z];
    z=f[i];
    ++i;
}
```

Доказательство корректности алгоритма: Допустим, что наш алгоритм вывел последовательность, которая больше максимальной, тогда возьмем последний выведенный элемент (kn)и посмотрим на f[kn] - там лежит предыдущий элемент, который точно меньше нашего ⇒ Если мы будем выводить так дальше, то мы докажем, что последовательность, выведенная алгоритмом действительно является больше максимальной. Отсюда следует, что наше предположение неверно.

Допустим, что наш алгоритм выдал не максимальную подпоследовательность, тогда возьмем максимальную подпоследовательнось $[a_{k1}, a_{k2}, ..., a_{kn}]$. Посмотрим на первый элемент, $d[k1] >= 1$, посмотрим на второй элемент $d[k2] >= 2$, так как слева от него есть элемент $[a_{k1}$ и т.д. Получим, что $d[kn] >= n$ ⇒ Если алгоритм не вывел эту подпоследовательность, то он вывел большую. Противоречие. Асимтотика по времени: В нашем алгоритме используется двойной цикл for, в котором выполняются процеду-

ры присваивания и проверка условия. Так как эти операции повторяются $O(n^2)$ раз, то асимтотика этой части будет $O(n^2)$. Далее в алгоритме мы используем цикл for в котором проводятся n операций за константное время, следовательно эта часть не влияет на асимтотику. В конце алгоритма мы проделываем k раз операции за константное время. Так как $k \leqslant n$, то это не влияет на асимтотику $\Rightarrow$ Асимтотика алгоритма $\Theta = O(n^2)$.
Асимтотика по памяти: Мы используем 4 переменные типа int и 3 массива: два из них содержат n чисел типа int, оставшийся содержит k чисел того же типа. Следовательно $\Theta = O(n)$.
6.
Алгоритм: будем добавлять элемент в стек, если элемент, записанный в стеке равен ему и делать pop, если элемент не равен верхнему элементу, записанного в стеке. Тогда после всех операций в нашем стеке останутся только одинаковые числа, которые встречаются больше, чем $n/2$ раз.
Корректонсть алгоритма: Допустим, что в стеке остались числа, которых меньше, чем $n/2$ или вообще ничего не осталось. Тогда поймем, что те числа, которых больше $n/2$ частино были в стеке, а частично не были. Для тех чисел, которых не было в стеке мы знаем, что они удалили один элемент из стека, а для тех чисел, которые были в стеке мы знаем, что потребовалось столько же чисел, чтобы удалить их из стека. Тогда на вход должно было подаваться больше, чем n чисел. Противоречие.