Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Мегафакультет трансляционных информационных технологий

Факультет инфокоммуникационных технологий

Дисциплина: Алгоритмы и структуры данных

**Отчет по Лабораторной работе №4**

Выполнила: Микулина Алиса Романовна

Группа: K3143, 1 курс

Преподаватель: Харьковская Татьяна Александровна

Санкт-Петербург

29.06.2022

**Описание задания**

**Задание 1.**

Наивно найти подстроку в строке. Долго, конечно, но надо

**Задание 2.**

В строке найти все возможные палиндромы из трех букв, которые можно получить вычеркиванием букв в строке.

**Задание 3.**

Опять ищем подстроку в строке, но по оптимизированному алгоритму Рабина-Карпа.

**Задание 4.**

С помощью магии, приворотов и хеширования нужно сравнить две подстроки в строке, и вывести, равны ли они.

**Задание 5.**

Вроде как обычная префикс функция, самая обыкновенная без всяких там приколов.

**Задание 6.**

Нужно написать z-функцию. Тут, скорее, сложно понять, что такое эта функция, а когда поймешь уже не сложно.

**Задание 7.**

Бинарным методом ищем наибольшую общую подстроку двух строк, еще и с помощью хеширования. Не так сложно, как звучит на самом деле, норм.

**Задание Сдвиг Текста.**

Из конца строки взяли сколько-то символов, перевернули и поставили в начало. Надо определить, сколько.

**Задание Поиск Подстроки.**

Тоже ищем подстроку, но очень быстро. Быстрее всего (по крайней мере тесты прошло) решение с помощью префиксов.

**Описание решения и исходный код**

**Задача 1.**

Все начинается так просто… Просто наивный алгоритм для нахождения подстроки. Мы знаем, какой длины строку ищем, поэтому просто циклом проходимся по строке, смотря на все вхождения такой длины, и сравниваем их с искомым) По сути, все)))

def **find\_instances**(to\_find, text):

    inst = ''

    count = 0

    len\_f = **len**(to\_find)

    len\_t = **len**(text)

    iterable = len\_t - (len\_f - 1)

    for i in **range**(iterable):

        comparable = text[i:i + len\_f]

        if comparable == to\_find:

            count += 1

            inst += **str**(i + 1) + ' '

    inst = inst[:-1]

    return [count, inst]

with **open**('input.txt') as f:

    to\_find = f.**readline**().**strip**()

    text = f.**readline**().**strip**()

ans = **find\_instances**(to\_find, text)

with **open**('output.txt', 'w') as d:

    d.**write**(**str**(ans[0]) + '\n')

    d.**write**(ans[1])

**Задача 2.**

Тут забавная задачка. Нужно найти все существующие палиндромы длиной в 3 символа, которые можно получить путем вычеркивания остальных символов.

На самом деле тоже очень просто, мы делаем сортировку подсчетом по символам и при этом храним индекс каждого вхождения символа. Потом мы просто смотрим индексы одинаковых элементов попарно и вычисляем количество букв между ними. Так мы, как раз, и найдем все микропалиндромы))

def **find\_three**(line):

    count = 0

    first = **ord**('a')

    last = **ord**('z')

    full = []

    left = []

    for i in **range**(first, last + 1):

        full.**append**(0)

        left.**append**(0)

    for char in line:

        id\_ = **ord**(char)

        full[id\_ - first] += 1

    for char in line:

        id\_ = **ord**(char) - first

        left[id\_] += 1

        for i in **range**(**len**(full)):

            if i == id\_:

                right = full[i] - left[i]

                cur\_left = left[i] - 1

                count += cur\_left \* right

            else:

                right = full[i] - left[i]

                count += left[i] \* right

    return count

with **open**('input.txt') as f:

    line = f.**readline**().**split**()

    text = ''

    for word in line:

        text += word

ans = **find\_three**(text)

with **open**('output.txt', 'w') as d:

    d.**write**(**str**(ans))

**Задача 3.**

Так как вариант нахождения подстроки из первой задачи далеко неэффективен, нам нужно написать алгоритм Робина-Карпа, основанный на хешировании.

Все, что я могу сказать о хешировании: это больно. Очень. Хеши можно написать, через день забыть, как написал. В итоге это почему-то работающий черный ящик, в который мы кидаем хомячка и получаем корги на выходе…

В общем, в чем суть алгоритма. Мы знаем длину подстроки. Дальше мы находим все хеши строк этой длины. Но это медленно, если считать каждый хеш отдельно, ведь там такие размеры строк могут быть!!! Для этого есть оптимизация этого алгоритма. Оно называется скользящий хеш.

Чтобы посчитать хешем все подстроки данной длины, “ручками” надо найти только первую такую подстроку, а дальше считать по формулке. Мы, получается вычитаем предыдущий элемент, домножаем все на основание и прибавляем следующий, ну и все дружно делим на Большое Простое Число)))

Строки с одинаковыми хешами будут, с огромной вероятностью, одинаковыми, вот и все)))

А теперь поговорим о боли… Я, когда находила вхождения с одинаковым хешем, добавляла циферку не в массив, а сразу в строку, ведь все равно потом сдавать в файлик таким макаром…

Да? Ну чтож, смотрю я, тесты валятся по времени. Думаю, надо оптимайзить. Стала считать степени заранее, а не каждый раз, чтобы не перегружать питон возведением в степень, убрала проверку на то, одинаковы ли строки буквенно, если они одинаковы хешами, так как шанс ошибки крайне мал… Вся равно не проходит… Спустя два часа я решила делать сначала массив, а потом уже его печатать текстово.

Разница в 0.5 секунды! Пол секунды были скушаны строками… Я даже и не думала, что они так тормозят прогресс…………… Поплакали, можно решать дальше))

with **open**('input.txt') as f:

    to\_find = f.**readline**().**strip**()

    text = f.**readline**().**strip**()

*# print(len(text))*

mult = 1019

primary = 10 \*\* 9 + 7

deg = [1 % primary]

find\_ln = **len**(to\_find)

for i in **range**(1, find\_ln):

    deg.**append**((1019 \* deg[i - 1]) % primary)

def **hash**(line, primary):

    num = 0

    for i in **range**(**len**(line)):

        letter = **ord**(line[i])

        num += letter \* deg[**len**(line) - i - 1]

        num = num % primary

    hashed = num % primary

    return hashed

def **precompute\_hashes**(find\_ln, line, mult, primary):

    hashes = [0 for i in **range**(**len**(line) - find\_ln + 1)]

    first = line[:find\_ln]

    hashes[0] = **hash**(first, primary)

    for i in **range**(**len**(hashes) - 1):

        old\_hash = hashes[i]

        old\_symb = **ord**(line[i])

        b\_degreed = deg[find\_ln - 1]

        new\_symb = **ord**(line[i + find\_ln])

        all\_old = ((old\_hash - old\_symb \* b\_degreed) \* mult) % primary

        new\_hash = all\_old + new\_symb

        hashes[i + 1] = new\_hash

    return hashes

def **RobinKarp**(to\_find, line):

    cnt = 0

    ans = []

    primary = 10 \*\* 9 + 7

    mult = 1019

    find\_ln = **len**(to\_find)

    hashed\_find = **hash**(to\_find, primary)

    hashed\_all = **precompute\_hashes**(find\_ln, line, mult, primary)

    for i in **range**(**len**(hashed\_all)):

        if hashed\_all[i] != hashed\_find:

            continue

        else:

            ans.**append**(**str**(i + 1))

            cnt += 1

    return (**str**(cnt), ' '.**join**(ans))

if **len**(to\_find) > **len**(text):

    ans = 0

else:

    ans = **RobinKarp**(to\_find, text)

with **open**('output.txt', 'w') as d:

    if **len**(to\_find) > **len**(text):

        d.**write**(**str**(ans))

    else:

        d.**write**(**str**(ans[0]) + '\n')

        d.**write**(ans[1])

**Задача 4.**

Эту задачу я села решать сразу после третьей, часов в 10 вечера. Написала, она падает. Падает и все. При этом хеши считаются, вроде все адекватно (но нет).

В итоге, эту задачу я решила к 9 вечера следующего дня, при этом еще успела решить все остальные в этом промежутке. Хуже этой задачи, мне кажется, нет ничего на свете. Если в прошлый раз мы учились квакать, то тут бы надо не квакнуться пока пишешь…

Я даже специально для отчета оставила микрокомметнарий в процессе решения для осознания всей боли…

“Пятый час сижу с этой задачей. Плачу. Не понимаю как это работает, а решить надо… вот бы мне внезапно пришло понимание того где что нужно умножать возводить в степень и вычитать. Я всю ночь сидела с задачами в коворкинге и все что мне сейчас хочется это плакать. Кристина принесла мне пледик, хоть что-то хорошее в этой жизни…”

Ладно, достаточно печали, надо отчет отчетить. Берем, получается, считаем первую подстроку строки. Дальше считаем на основе первого хеща все остальные подстроки, увеличивая количество буковок. Получается что-то типа скользящий односторонний хеш. Чтобы нат хеш подстроки, нам нужно вычесть из префикса, оканчивающейся на последнюю букву подстроки, префикс, заканчивающийся на одну букву раньше, чем первый символ подстроки. Че-то там перемножить, че-то там возвести в степень, и методом волшебного тыка подбираем комбинацию, с которой это все начинает волшебным образом работать!

Очень не люблю эту задачку, я прям в расстройстве с нее, но решила.

Кстати от того, что решила ее, никакого удовлетворения. Хотя обычно я решенным задачам прям радуюсь.

MAX\_SIZE = 2 \* 10 \*\* 6

MULT = 1019

PRIMARY = 10 \*\* 9 + 7

deg = [1 % PRIMARY]

find\_ln = MAX\_SIZE

for i in **range**(1, find\_ln):

    deg.**append**((1019 \* deg[i - 1]) % PRIMARY)

def **hash**(line):

    ln\_line = **len**(line)

    hashed\_pefixes = [0 for i in **range**(ln\_line + 1)]

    hashed\_pefixes[0] = 0

    for i in **range**(1, ln\_line + 1):

        hashed\_pefixes[i] = (hashed\_pefixes[i - 1] + **ord**(line[i - 1]) \* deg[i]) % PRIMARY

    return hashed\_pefixes

def **whether\_equal**(first, second, ln, full\_ln, hashed):

    first\_hashed = ((hashed[first + ln] - hashed[first]) \* deg[full\_ln - first]) % PRIMARY

    second\_hashed = ((hashed[second + ln] - hashed[second]) \* deg[full\_ln - second]) % PRIMARY

    if first\_hashed == second\_hashed:

        return 'YES'

    return 'NO'

with **open**('input\_4.txt') as f:

    line = f.**readline**().**strip**()

    hashed = **hash**(line)

    num\_checks = **int**(f.**readline**())

    for line in f.**readlines**():

        first, second, ln = **list**(**map**(**int**, line.**split**()))

**print**(**whether\_equal**(first, second, ln, **len**(line), hashed))

**Задача 5.**

Одна из тех задач, что я решала для морального успокоения. Описание есть полностью в лекции. Делается это все чудо за линию за счет того, что у нас есть два указателя, которые мы передвигаем.

Получается, мы либо увеличиваем значение j, если строка от j равна строке от i-1, а если нет, то j смещаем на начало строки и сравниваем заново. И так до конца.

Несложно и быстро, причем очень быстро)))

import **time**

t\_start = **time**.**perf\_counter**()

def **prefix**(line):

    stripped\_line = **list**(line)

    len\_line = **len**(line)

    pref = [0 for i in **range**(len\_line)]

    j = 0

    for i in **range**(2, len\_line + 1):

        while j > 0 and stripped\_line[j] != stripped\_line[i - 1]:

            j = pref[j - 1]

        if stripped\_line[j] == stripped\_line[i - 1]:

            j += 1

        pref[i - 1] = j

    return pref

with **open**('input\_5.txt') as f:

    line = f.**readline**().**strip**()

ans = **prefix**(line)

with **open**('output\_5.txt', 'w') as d:

    for elem in ans:

        d.**write**(**str**(elem) + ' ')

**print**("Время работы: %s секунд " % (**time**.**perf\_counter**() - t\_start))

**Задача 6.**

И ещееее одна задача, которую я решала для морального успокоения. Чем-то похожа на префикс, но не префикс.

Вообще z-функция показывает для каждого элемента с индексом i, какой длины подстрока, начинающаяся с этого элемента, может быть префиксом всей строки.

Мем какой-то конечно, но почему бы и нет, если все так просто…

Использем, получается, z-блоки. Z-блок начинается с I и длина у него z[i]. Конец и начало блока храним в R (right) и L (left). В зависимости от того, меньше i чем right или нет мы либо бегаем внутри отрезка, либо снаружи, добавляя значения ячейкам когда видим, что дальше идет подходящее значение. Так пробегаемся по всей строке, и в принципе все)))

def **zed**(line):

  zedded = [0] \* **len**(line)

  L, R = 0, 0

  for i in **range**(1, **len**(line)):

    zedded[i] = **max**(0, **min**(zedded[i - L], R - i))

    while i + zedded[i] < **len**(line) and line[zedded[i]] == line[i + zedded[i]]:

      zedded[i] += 1

    if i + zedded[i] > R:

      L, R = i, i + zedded[i]

  return zedded

with **open**('input.txt') as f:

    line = f.**readline**().**strip**()

ans = **zed**(line)[1:]

with **open**('output.txt', 'w') as d:

    for elem in ans:

        d.**write**(**str**(elem) + ' ')

**Задача 7.**

Я бы сказала, что эта задача Больше красивая, чем сложная. Бинарный поиск очень хорошо влезает в эту задачу. С самими хешами нет никаких проблем, потому что они такие же, как в старых задачах, а вот binary search тут прекрасен. Мы делим с самого начала длину строки пополам, и ищем все подстроки длиной в половину строки. Если находим, то записываем координаты и идем искать больше, если нет – то меньше. Опять пополамим и ищем так пока не дойдем то ограничителей.

Очень красивая задачка, мне прям понравилась 😊

def **hash**(line, primary):

    global deg

    num = 0

    for i in **range**(**len**(line)):

        letter = **ord**(line[i])

        num += letter \* deg[**len**(line) - i - 1]

        num = num % primary

    hashed = num % primary

    return hashed

def **precompute\_hashes**(find\_ln, line, mult, primary):

    global deg

    hash\_dict = {}

    hashes = [0 for i in **range**(**len**(line) - find\_ln + 1)]

    first = line[:find\_ln]

    hashes[0] = **hash**(first, primary)

    hash\_dict[hashes[0]] = 0

    for i in **range**(**len**(hashes) - 1):

        old\_hash = hashes[i]

        old\_symb = **ord**(line[i])

        b\_degreed = deg[find\_ln - 1]

        new\_symb = **ord**(line[i + find\_ln])

        all\_old = ((old\_hash - old\_symb \* b\_degreed) \* mult) % primary

        new\_hash = all\_old + new\_symb

        hashes[i + 1] = new\_hash

        if new\_hash in hash\_dict.**keys**():

            continue

        else:

            hash\_dict[new\_hash] = i + 1

    return hash\_dict

def **binary\_search**(smaller, bigger, old\_dividor, very\_old\_dividor, border, direction):

    global CURRENT\_BIGGEST

    if old\_dividor == very\_old\_dividor:

        return

    if direction == 'right':

        if (very\_old\_dividor + old\_dividor) % 2 != 0:

            new\_splitline = (very\_old\_dividor + old\_dividor) // 2 + 1

        else:

            new\_splitline = (very\_old\_dividor + old\_dividor) // 2

    else:

        if (very\_old\_dividor - (very\_old\_dividor - old\_dividor)) % 2 != 0:

            new\_splitline = (very\_old\_dividor - (very\_old\_dividor - old\_dividor)) // 2 + 1

        else:

            new\_splitline = (very\_old\_dividor - (very\_old\_dividor - old\_dividor)) // 2

    if new\_splitline == 0 or new\_splitline == border:

        return

    all\_smol = **precompute\_hashes**(new\_splitline, smaller, mult, primary)

    all\_big = **precompute\_hashes**(new\_splitline, bigger, mult, primary)

    found = (**find\_same**(all\_smol, all\_big))

    if found[0] == True:

        CURRENT\_BIGGEST = [found[1], found[2], new\_splitline]

        direction = 'right'

    else:

        direction = 'left'

**binary\_search**(smaller, bigger, new\_splitline, old\_dividor, border, direction)

def **find\_same**(all\_smol, all\_big):

    for key in all\_smol.keys():

        if key in all\_big.keys():

            id\_1, id\_2 = all\_smol[key], all\_big[key]

            return [True, id\_1, id\_2]

    return [False]

def **find\_biggest\_subline**(first, second):

    global mult, primary, CURRENT\_BIGGEST

    if **len**(first) < **len**(second):

        smaller, bigger = first, second

    else:

        smaller, bigger = second, first

    tiny\_len = **len**(smaller)

    if tiny\_len % 2 != 0:

        first\_splitter = tiny\_len // 2 + 1

    else:

        first\_splitter = tiny\_len // 2

    all\_smol = **precompute\_hashes**(first\_splitter, smaller, mult, primary)

    all\_big = **precompute\_hashes**(first\_splitter, bigger, mult, primary)

    first\_found = (**find\_same**(all\_smol, all\_big))

    if first\_found[0] == True:

        CURRENT\_BIGGEST = [first\_found[1], first\_found[2], first\_splitter]

        direction = 'right'

    else:

        direction = 'left'

**binary\_search**(smaller, bigger, first\_splitter, tiny\_len, tiny\_len, direction)

    return

mult = 1019

primary = 10 \*\* 9 + 7

with **open**('input\_7.txt') as f:

    for line in f.**readlines**():

        CURRENT\_BIGGEST = [0, 0, 0]

        first, second = line.**split**()

        len\_one, len\_two = **len**(first), **len**(second)

        find\_ln = **max**(len\_one, len\_two)

        deg = [1 % primary]

        for i in **range**(1, find\_ln):

            deg.**append**((1019 \* deg[i - 1]) % primary)

**find\_biggest\_subline**(first, second)

**print**(CURRENT\_BIGGEST)

**Задача Сдвиг Текста.**

Пока я писала сам отчет, я поняла, что можно было бы решить ее с помощью z-функции, но мне уже некогда писать, да и тем более это решение хоть и наивное, но тоже зашло в проверку. Тут мы буквально поочередно берем букву с конца и ставим в начало строки, а потом сравниваем с тем, что получилось в условии задачи. Считаем количество изменений, потом их выводим. Если получить такую строку такими вращениями не получается, то значит нужно вывести -1.

Z-функцией бы было быстрее, но я даже не уверена что оно работает, но мне кажется работает))))

def **cycle**(initial, final):

    if initial == final:

        return 0

    len\_str = **len**(initial)

    temp = initial

    for i in **range**(len\_str):

        temp = temp[-1:] + temp[:-1]

        if temp == final:

            return i + 1

    return -1

with **open**('input\_cycle.txt') as f:

    initial = f.**readline**().**strip**()

    final = f.**readline**().**strip**()

ans = **cycle**(initial, final)

with **open**('output\_cycle.txt', 'w') as d:

    d.**write**(**str**(ans))

**Задача Поиск Подстроки.**

Тут все просто и сердито. Берем p-функцию и с омощью нее решаем. Это, вроде как, самый быстрый вариант. Мы берем строку, которую ищем, после нее ставим разделитель # и саму строку в которой будем искать.

После прохождения функцией всего массива, то части, описывающей все после # будет храниться информация о том, где есть нужные нам вхождения. Так как мы знаем длину искомой нам строки, то мы буквально можем посмотреть, где лежит эта длина, и это и будут наши индексы))) Только нам нужно вычесть длину искомой подстроки, а то будет неправильно)

def **find\_subline**(line, fin\_len):

    stripped\_line = **list**(line)

    len\_line = **len**(line)

    pref = [0 for i in **range**(len\_line)]

    j = 0

    for i in **range**(2, len\_line + 1):

        while j > 0 and stripped\_line[j] != stripped\_line[i - 1]:

            j = pref[j - 1]

        if stripped\_line[j] == stripped\_line[i - 1]:

            j += 1

        pref[i - 1] = j

    ans = ''

    for i in **range**(**len**(pref)):

        if pref[i] == fin\_len:

            ans += **str**(i - 2 \* fin\_len) + ' '

    return ans

with **open**('input.txt') as f:

    line = f.**readline**().**strip**()

    to\_find = f.**readline**().**strip**()

    fin\_len = **len**(to\_find)

    zhaba = to\_find + '#' + line *# zhaba is a mix of the line we need to find and the line itself, separated by #*

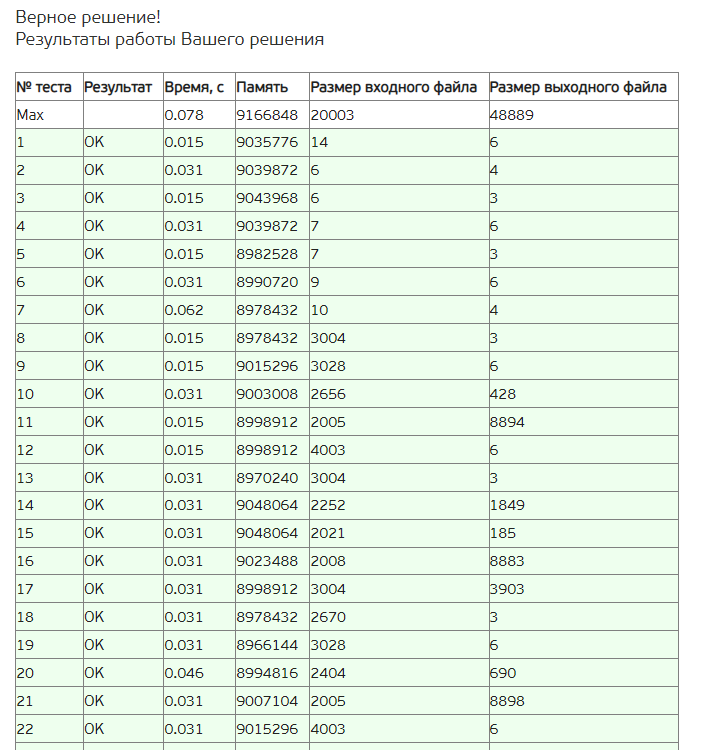
    ans = **find\_subline**(zhaba, fin\_len)

with **open**('output.txt', 'w') as d:

    d.**write**(ans.**strip**())

**Описание проведенных тестов.**

**1**

Изображение выглядит как стол

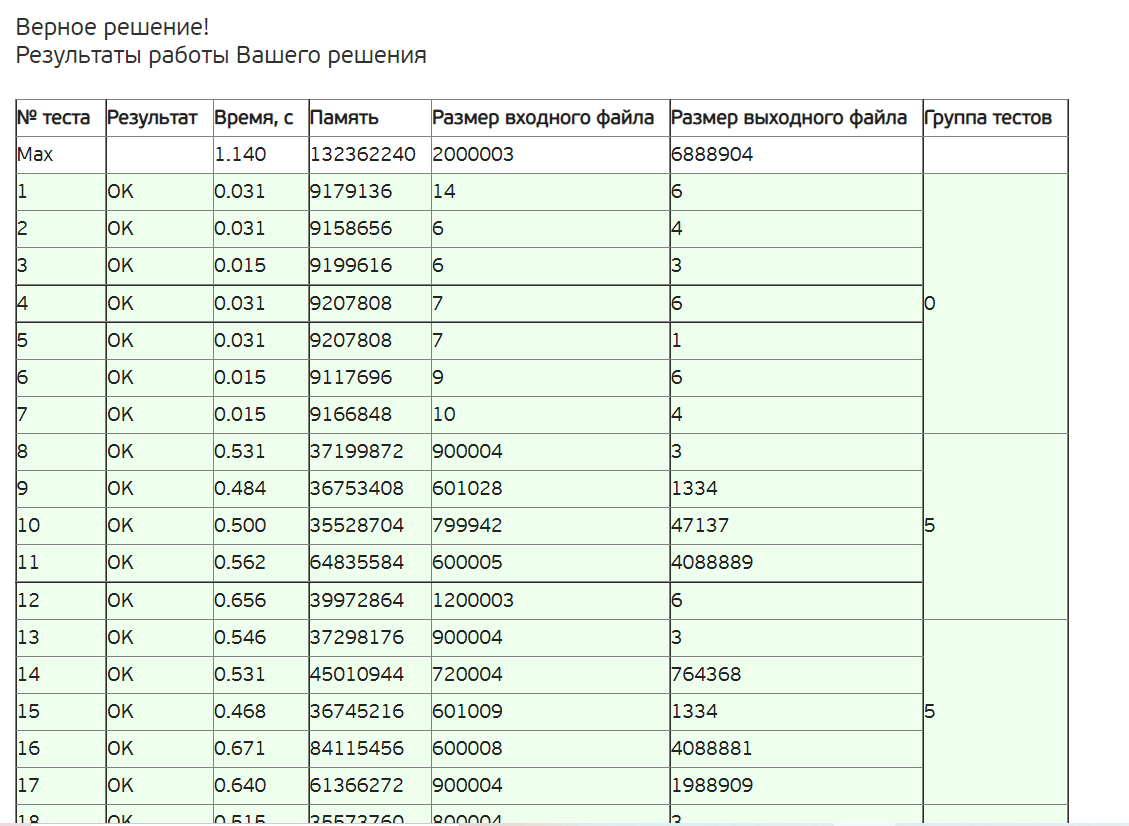
Автоматически созданное описание

**2**

Изображение выглядит как стол

Автоматически созданное описание

**3**

****

**Изображение выглядит как стол

Автоматически созданное описание**

**4**

|  |  |
| --- | --- |
| **Input** | **Output** |
| aaabaaab  5  0 4 4  0 4 3  0 4 2  0 4 1  3 4 1 | YES  YES  YES  YES  NO |
| trololo  4  0 0 7  2 4 3  3 5 1  1 3 2 | YES  YES  YES  NO |

**5**

Изображение выглядит как стол

Автоматически созданное описание

**6**

Изображение выглядит как стол

Автоматически созданное описание

**7**

|  |  |
| --- | --- |
| **Input** | **Output** |
| cool toolbox  aaa bb  aabaa babbaab  aaaaaaaaaaaaaaaaaaaaaaaaaabbbbdddaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab  fghjkytbnkiuygbnmkjhgfdfghjklkjhgfdfghhdsfysj cvbnmfjrdcvbjhgfdsfghjkjhgfdsdfvbnmbnbnbbbbbhhhfdd | [1, 1, 3]  [0, 0, 0]  [0, 4, 3]  [0, 0, 25]  [17, 22, 6] |

**Сдвиг текста**

Изображение выглядит как стол

Автоматически созданное описание

**Поиск подстроки**

Изображение выглядит как стол

Автоматически созданное описание

**Выводы по проделанной работе.**

Такс-такс, утро перед экзаменом, я пишу отчет))) Могла бы, конечно, написать ночью, но это бы уже вторая ночь без сна, я б не выдержала. Лабу я в итоге решала два дня и полную ночь, или около того.

Вот и все)) Лаба очень красивая сама по себе, над ней бы подольше посидеть))) Хочу теперь сдвиг текста написать более эффективно, хотя бы через хеши лили мою идею с z-функцией. Прям хочется посидеть подумать))

Я была в шоке с того, как долго работает сложение строк, честно. А еще нет ничего в мире сложнее упрощения подсчета хешей :DDD

Очень грустно, что на этом все… А кому же еще я буду писать свои отчеты, зная, что они понравятся… Как же ломать мозг ночами над задачками… Радоваться каждый раз когда что-то начало работать, ходить на защиты… эх, я буду скучать, больше всего, именно по алгоритмам…