# Задача 1. Время в школе

Для того, чтобы определить общую длительность всех уроков, нужно количество уроков $n$ умножить на длительность урока $a$.

Между $n$ уроками будет $(n-1)$ перемена, из них обычных перемен будет $(n-2)$, поэтому общая длительность перемен находится по формуле $(n-2)*b$.

Для получения общего времени, которое Вася провел в школе, остается добавить длительность большой перемены 30.

Описанные рассуждения можно записать в виде следующего кола на языке программирования Python.

```
n = int(input())
a = int(input())
b = int(input())
print(n * a + (n - 2) * b + 30)
```

# Задача 2. Разноэтажный дом

Для начала рассмотрим расположение квартир в первом подъезде. На первом этаже расположены квартиры с номерами 1, 2, 3; на втором этаже расположенные квартиры с номерами 4, 5, 6 и так далее. Получим формулы для нахождения номеров квартир первого подъезда:

- $(K-1)*3+1$;
- $(K-1)*3+2$;
- $(K-1)*3+3$.

где $K$ — номер этажа.

Для второго и третьего подъезда формулы изменятся незначительно. А именно, необходимо добавить в формулы ещё одно слагаемое — количество квартир в подъездах с меньшими номерами (в первом подъезде для формул второго подъезда, в первых двух подъездах для третьего подъезда). Рассмотрим решение, использующее полученные формулы:

```
FLATS = 3
a = int(input())
b = int(input())
c = int(input())
k = int(input())

prev = 0
print((k - 1) * FLATS + 1)
print((k - 1) * FLATS + 2)
print((k - 1) * FLATS + 3)

prev += FLATS * a
print(prev + (k - 1) * FLATS + 1)
print(prev + (k - 1) * FLATS + 2)
print(prev + (k - 1) * FLATS + 3)

prev += FLATS * b
print(prev + (k - 1) * FLATS + 1)
print(prev + (k - 1) * FLATS + 2)
print(prev + (k - 1) * FLATS + 3)
```

Переменная FLATS хранит в себе количество квартир на площадке, переменная prev — количество квартир в подъездах с меньшими номерами.

Однако данное решение наберёт не более 40 баллов, ибо не учитывает, что в некоторых подъездах может не быть этажа с номером $K$. Для получения полного балла достаточно добавить следующую модификацию в решение — перед выводом номеров квартир проверять, что в данном подъезде достаточное количество этажей.

Итоговое решение выглядит так:

```
FLATS = 3
a = int(input())
b = int(input())
c = int(input())
k = int(input())

prev = 0
if a >= k:
    print((k - 1) * FLATS + 1)
    print((k - 1) * FLATS + 2)
    print((k - 1) * FLATS + 3)

prev += FLATS * a
if b >= k:
    print(prev + (k - 1) * FLATS + 1)
    print(prev + (k - 1) * FLATS + 2)
    print(prev + (k - 1) * FLATS + 3)

prev += FLATS * b
if c >= k:
    print(prev + (k - 1) * FLATS + 1)
    print(prev + (k - 1) * FLATS + 2)
    print(prev + (k - 1) * FLATS + 3)
```

# Задача 3. Длина числа

Если немного перефразировать условие, в задаче требуется посчитать суммарное количество цифр во всех целых числах от $L$ до $R$. Можно сделать это, пробежавшись от $L$ до $R$ циклом, на каждом шаге добавляя к ответу количество цифр в очередном числе:

```
L = int(input())
R = int(input())
sum = 0
for i in range(L, R+1):
    sum += len(str(i))
print(sum)
```

Такой способ набирает 44 балла, однако при полных ограничениях не проходит по времени. Например, на третьем тесте из условия приходится перебирать циклом уже $10^9$ чисел, а в худшем случае это количество может вообще достигать $10^{17}$.

Для решения на полный балл применим следующий прием — вместо вычисления количества от $L$ до $R$ вычислим это количество во всех числах от 1 до $R$ и вычтем из него количество в числах от 1 до $L-1$. Тем самым мы свели нашу задачу к двум однотипным задачам про числа от 1 до $n$, в которых есть только правая граница, а не обе границы сразу.

Как решать задачу про суммарное количество цифр во всех целых числах от 1 до $n$? Найдем $k$ — количество цифр в числе $n$. Тогда наш диапазон попадут все однозначные, двузначные $\dots k-1$-значные числа. Пробежимся циклом по всем этим длинам (это не больше 17 итераций) и прибавим к ответу количество всех чисел такой длины (равное $9 \cdot 10^{длина-1}$), умноженное на эту длину. А чисел длины $k$ в наш диапазон попадет $n - 10^{k-1} + 1$.

```
def from_1_to_n(n):
    k = len(str(n))
    sum = (n - 10**(k-1) + 1) * k
    for i in range(1, k):
        sum += 9 * 10**(i-1) * i
    return sum

L = int(input())
R = int(input())
print(from_1_to_n(R) - from_1_to_n(L-1))
```

Отметим, что первую подгруппу на 20 баллов ($R \leqslant 50$) можно было решить без использования циклов с помощью нескольких условных операторов. Например, разобрав отдельно три случая: обе границы — однозначные числа (ответ $R-L+1$), обе границы — двузначные числа (ответ $2 \cdot (R-L+1)$) и, наконец, левая граница — однозначная, а правая — двузначная (ответ $2 \cdot (R-9) + (10-L)$).

# Задача 4. Браслет

Рассмотрев внимательно условие задачи можно заметить, что нас интересуют сдвиги исходной строки которые являются палиндромами. Более того, под позицией разреза подразумевается индекс символа, с которого начинается сдвиг (причём в нулевой индексации). Единственным исключением является сдвиг, начинающийся с индекса 0 (исходная строка). Для этого сдвига позицией разреза считается значение равное длине строки — $N$.

Для упрощения понимания, рассмотрим все сдвиги строки «arbat», которая представляет из себя второй тест:

- строка «arbat», индекс начала сдвига 0, позиция разреза $N$;
- строка «rbata», индекс начала сдвига 1, позиция разреза 1;
- строка «batar», индекс начала сдвига 2, позиция разреза 2;
- строка «atarb», индекс начала сдвига 3, позиция разреза 3;
- строка «tarba», индекс начала сдвига 4, позиция разреза 4.

Самым очевидным решением является следующее. Переберём все возможные сдвиги (индексы начала сдвигов). И для каждого сдвига проверим, является ли он палиндромом. Если это так, добавим к ответу данную позицию разреза.

```
n = int(input())
word = input()

ans = []
for left in range(n):
    shift = word[left:] + word[:left]
    if shift == shift[::-1]:
        ans.append(left if left else n)

print(len(ans))
for value in ans:
    print(value)
```

В данном решении для нахождения сдвигов и переворота строки используются срезы. Также, в 8 строке используется тернарный оператор, необходимый для того, чтобы вместо нуля добавить в ответ $N$.

Оценим производительность данного решения. Внешний цикл совершает ровно $N$ итераций. Внутри цикла мы получаем сдвиг строки и сравниваем его со своей перевёрнутой версией, что также занимает линейное время. Итого время работы решения будет пропорционально $N^2$, что позволит ему набрать не более 36 баллов.

Для построения более производительного решения необходимо заметить, что в слове от 1 до 8 букв «a». Если зафиксировать одну букву и рассмотреть, как может выглядеть палиндром, содержащий данную букву, то можно заметить следующее. Либо данная буква будет в центре палиндрома, либо центр палиндрома будет точно посередине между двумя последовательными вхождениями данной буквы в строку.

На самом деле можно заметить ещё несколько закономерностей, в том числе связанных с чётностью длины строки. Однако это приведёт к усложнению решения и не требуется в данном случае.

Исходя из этого, легко видеть, что нам достаточно проверить максимум 16 возможных позиций потенциального центра палиндрома. Рассмотрим решение, использующее данную логику:

```
n = int(input())
word = input()

pos = []
for i in range(n):
    if word[i] == 'a':
        if pos and (pos[-1] + i) // 2 not in pos:
            pos.append((pos[-1] + i) // 2)
        pos.append(i)
if len(pos) > 1 and (pos[-1] + pos[0] + n) // 2 % n not in pos:
    pos.append((pos[-1] + pos[0] + n) // 2 % n)

ans = []
for center in pos:
    left = (center + n // 2 + 1) % n
    shift = word[left:] + word[:left]
    if shift == shift[::-1]:
        ans.append(left if left else n)

print(len(ans))
for value in ans:
    print(value)
```

В данном решении сначала определяются позиции возможных центров палиндрома. Очевидно, что все позиции буквы «a» в слове являются потенциальными центрами.

Кроме этого необходимо добавить позиции между вхождениями «a». При этом, когда между последовательными вхождениями буквы «a» чётное количество символов, и центром палиндрома являются сразу 2 символа, то из них выбирается «левый». Также может возникнуть ситуация, когда две буквы «a» идут подряд — тогда центр палиндрома будет совпадать с позицией буквы «a» (и нужно избежать повторного добавления данного центра).

В конце необходимо добавить позицию между последним и первым вхождением «a» (напоминаем, что буквы слова записаны по кругу).

После этого перебираются выбранные центральные позиции, для каждой центральной позиции находится соответствующий ей сдвиг и проверяется его палиндромность.

Производительность данного решения много лучше, чем у предыдущего. Ибо мы сделаем не более 16 проверок с линейным временем работы. Итого время работы решения будет пропорционально $N$, что позволит ему набрать полный балл.

# Задача 5. Задача о числах

В этой задаче достаточно написать эффективную эмуляцию действий Васи.

Для начала попробуем смоделировать процесс ровно так, как он описан в условиях, что вполне реально при дополнительных ограничениях $m \leqslant 2n$. Для этого считаем $a_1, a_2, \ldots, a_n$ и запишем эти числа в массив $a$. Для эмулирования надо понять, как написать «стирание» и «добавление», описанные в условиях.

Сначала нужно сделать «добавление». Во многих языках программирования в массив можно вставлять новые элементы справа (например, в Python это делается при помощи метода `.append()`). Так, эмулировать действия Васи можно достаточно просто: пусть самое левое не стертое число — $a_i$. Тогда при помощи цикла $a_i$ раз вставим в конец массива элемент $a_i$.

Отметим, что при отсутствии такой команды в выбранном языке программирования, можно завести массив достаточно большой длины, так же считать в него $n$ чисел, а оставшиеся не трогать. Теперь можно завести переменную — индекс первого еще не использованного индекса массива, изначально равный $n + 1$. Теперь можно «вставлять в конец» следующим образом: запишем по индексу, лежащему в этой переменной, необходимое число, после чего увеличим значение переменной на 1. Минус лишь в том, что длину массива нужно при таком подходе заранее угадать, чтобы при прохождении тестов не произошел выход за границы массива.

Чтобы реализовать «стирание» можно удалить первый элемент массива, но это долгая операция. Поэтому лучше «стиранию» сопоставить в эмуляции не явное удаление элемента из массива, а увеличение на единицу «начала» массива. Для этого заведем переменную $k$ — индекс числа, которое Вася еще не стер, и, с одной стороны, первое не стертое число теперь не всегда $a_1$ в массиве, а $a_k$ (что, исходя из описания выше, не существенно для реализации «добавления»), но с другой стороны без удаления эмуляция будет работать намного быстрее. Впрочем, если использовать вместо массива очередь, список или дек, этой проблемы можно избежать.

Отметим, что совершаемые действия: зафиксировать индекс $k$, сделать с ним некоторые действия («добавление»), а потом увеличить его на единицу — это в точности то, что делает цикл `for`. Однако циклу `for` необходим диапазон, в котором он будет перебирать $k$. Это может быть, например, диапазон от 1 до $m - n$ (если $m > n$). Тогда полученная эмуляция будет состоять из двух вложенных циклов, а после ее завершения ответом на задачу будет число $a_m$. Такое решение набирает не менее 20 баллов.

Проблема такого подхода заключается в том, что одна операция «добавления» может увеличить размер массива на $10^9$, что не уложится ни в ограничения по времени, ни в ограничения по памяти. Но для того, чтобы получить ответ, то есть число $a_m$, не требуется так много добавлять в массив — достаточно завершить эмуляцию в тот момент, когда длина массива станет не меньше $m$. Это можно сделать при помощи команды `break` во вложенном массиве. Такое решение набирает не менее 40 баллов.

Однако для того, чтобы решить задачу на полный балл, такого подхода недостаточно, и проблему с неэффективностью «добавления» нужно решить по-другому.

Заметим, что массив тратит слишком много памяти для хранения простой информации: в некоторых $a_i$ позициях, идущих подряд, записано одно и то же число $a_i$. Оптимизация заключается ровно в этом: будем хранить вместо некоторого числа пару (само число; сколько раз это число записано друг за другом). Пару можно реализовать встроенными средствами языка (например в Python это делается при помощи кортежей), а можно разбить пару на два массива: в первом хранить сами числа, а во втором — сколько раз они идут подряд.

Так, например, изначально массив можно задать парами $(a_1, 1), (a_2, 1), \ldots, (a_n, 1)$.

Однако чтобы эта оптимизация была эффективна не только по затрачиваемой памяти, но и по времени, нужно еще и обрабатывать не каждое число, вытаскивая его из пары, а всю пару, то есть сразу несколько одинаковых чисел, за раз.

Если Вася подходит к числам, равных $a_i$, в паре $(a_i, cnt_i)$ и мы сразу эмулируем все $cnt_i$ чисел, то «стирание» будет реализовано абсолютно аналогично: мы просто переходим к следующей паре. Значит, обрабатывать пары можно циклом.

«Добавление» теперь требует всего одного действия: по паре $(a_i, cnt_i)$ в конец массива добавляется пара $(a_i; cnt_i \cdot a_i)$. Самостоятельно подумайте, почему количество чисел, которое будет идти подряд, именно такое.

Но теперь появляется проблема в том, что ответ не лежит в $m$-й паре, более того эта оптимизация

сделана ровно для того, чтобы пара, в которой содержалось $m$-е стертое число, появлялась раньше.

Чтобы найти $m$-е стертое число будем считать, сколько чисел осталось «добавить», чтобы мы смогли узнать ответ. Пусть это число равно $d$. Изначально, если $m > n$, то $d = m - n$ (самостоятельно подумайте, что делать, если $m \leqslant n$). Теперь при очередной операции «добавления» может произойти так, что добавилась пара $(a_i, cnt_i)$, где $cnt_i \leqslant d$. Тогда мы точно понимаем, что $m$-е число лежит точно в последней паре, и ответ $a_i$ — можно тут же выводить это число и выходить из цикла (или завершать работу программы). Иначе, если $cnt_i > d$, давайте вычтем из $d$ число $cnt_i$ — теперь именно столько чисел надо еще «добавить», чтобы узнать $m$-е.

Осталось оценить время работы программы, и для этого нужно подобрать худший случай. На самом деле, эта оптимизация ужасно работает в случае, когда в массиве всего одна единица. Действительно, всегда будет добавляться пара $(1; 1)$, число $d$ будет идти от $m - 1$ до 0, уменьшаясь с каждым шагом цикла всего на 1, что совсем не лучше, чем предыдущая реализация.

Тем не менее, мы понимаем, что основную проблему мы решили, поэтому давайте назовем этот случай «особым» и разберем отдельно. К нему добавим в целом случаи, когда массив изначально состоит только из единиц, равно случаи, когда $d$ уменьшается всегда ровно на 1. Разбор таких случаев тривиален: ответ в них всегда равен 1.

Теперь в худшем случае в массиве есть хотя бы одно число, большее 1, то есть не менее 2. Пусть это число равно $x$. Теперь, когда алгоритм дойдет до пары $(x; 1)$ запишет пару $(x; x)$. Потом он дойдет до пары $(x; x)$ и запишет пару $(x; x^2)$. И так далее. То есть за 1-е, 2-е, 3-е, 4-е, ... $n$ шагов $d$ уменьшится не менее, чем на 1, $x$, $x^2$, $x^3$, и так далее. Значит, если $x^t > m$, то за $(t + 1) \cdot n$ шагов эмуляция завершится, откуда сложность алгоритма: $\mathcal{O}(nlog_x(m))$.

К сожалению, для некоторых языков программирования этой асимптотики недостаточно, чтобы пройти самые сложные тесты. Но как мы видим, основная проблема связана с тем, что массив может почти целиком состоять из единиц, которые в сумме вычтут из $m$ сильно меньше, чем одна единственная $x$ на поздних шагах цикла.

Тогда можно просто сделать «сжатие» входных данных: если во входных парах после $(a_i; 1)$ хочется добавить пару $(a_{i+1}; 1)$, где $a_i = a_{i+1}$, то вместо этого просто увеличим счетчик последней пары на 1. Можно доказать, что сложность такого алгоритма: $\mathcal{O}(nlog_x(m/n))$.