Сегодняшнее занятие – вводное. Решаем задачи, дальше – разбор, дальше – перерешивание. Новых интересных тем сегодня не будет, ждите их со следующего занятия.

Разбор задач.

Задача А. Апельсины бочками.
Задача на условный оператор. Как нужно было ее решать: выписать на бумажку все возможные формы слова «бочка» для разных чисел, понять, с какими числами какая форма употребляется, написать программу, которая в зависимости от числа выводит соответсвующую форму.

Начинаем выписывать.

Одна – бочка.
Две, три, четыре – бочки.
Пять, шесть и так далее – бочек.
Но двадцать одна – бочка.
Двадцать две, двадцать три, двадцать четыре – бочки.
Двадцать пять – бочек.

Закономерность уже явно видно. Однако, стоит немного подумать и понять, что, например, для чисел, больших 100, работают абсолютно те же закономерности, что и для меньших чисел – для 101 форма та же самая, что и для 1, для 502 – та же, что и для 2, для 711 – та же, что и для 11 и так далее. Об этом достаточно легко забыть, особенно для чисел, заканчивающихся, например, на 11.
Отсюда вытекает важное соображение при решении разных задач – постарайтесь проверить, что вы точно учли *все* случаи. Их часто бывает много.

Теперь о том, как это писать. Мы уже поняли, что в числе нам важны только две последние цифры. Упростим себе работу, оставив только две последние цифры от числа, которые нам и нужны. Дальше по нашим рисункам на бумажке напишем большой красивый условный оператор, который сделает все, что нам нужно. Выглядит осмысленная часть решения задачи примерно вот так (код на Python 3):

```
x = N %100
if x >= 10 and x <= 20:
    print (N, "bochek")
elif x % 10 == 1:
    print(N, "bochka")
elif (x % 10) in (2, 3, 4):
    print(N, "bochki")
else:
    print(N, "bochek")
```

Задача В. Сложное уравнение.

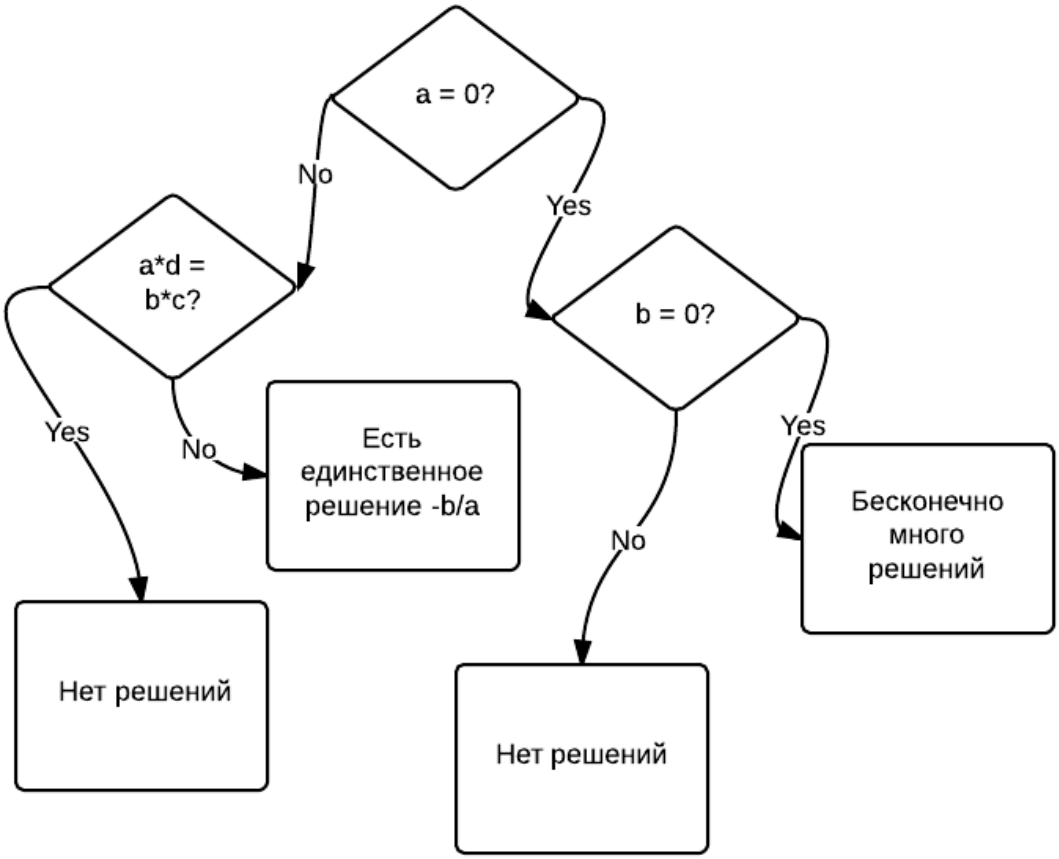
Это еще одна задача на ту же идею: выписать все случаи на бумажке и аккуратно написать условный оператор. Тут, правда, рассуждения более математические, но в целом, все то же самое, что и в задаче А. Давайте рассуждать.
Когда некоторое число может являться решением такого уравнения? Когда при его подстановке числитель дроби – 0. Тогда, казалось бы, искомое число – это $-b/a$.
А что произойдет, если $a = 0$? Ведь на 0 делить нельзя. Посмотрим, как тогда вообще выглядит наше уравнение:

$$\frac{b}{cx + d} = 0$$

Сразу становится немного понятнее. Если $b = 0$, то бесконечно много чисел являются решениями уравнения из числителя, а так как хотя бы одно из чисел не равно нулю, то знаменательно в ноль не обращается, то есть и у уравнения бесконечно много решений. Если же $b \neq 0$, то решений у уравнения нет.

Хорошо, со случаем $a = 0$ разобрались. Теперь вернемся к случаю $a \neq 0$.
В каком случае – b/a все равно не решение уравнения? В случае, если оно обращает знаменатель дроби в 0, то есть если $c * (-b/a) + d = 0$. (Забегая вперед, скажу, что в программе проверять это лучше, не используя деление, а домножив выражение на a и проверяя, не равны ли $a * d$ и $b * c$)

Случаев в задаче получилось достаточно много, чтобы запутаться, и если сразу начинать писать программу, может получится что-то совершенно невразумительное. В таких случаях полезно рисовать что-то вроде блок-схемы. Для этой задачи получится что-то такое:



Теперь несложно написать код программы, в котором, правда, нужно еще не забыть перед выводом ответа проверить, целое ли это число ☺ (код на Python 3):

```
if a == 0:
    if b == 0:
        print("INF")
    else:
        print("NO")
else:
    if a*d == b*c:
        print("NO")
    else:
        if b%a == 0:
            print(-b//a)
        else:
            print("NO")
```

Задача C. Метод бутерброда.

Такие задачи проще всего решать, нарисовав на бумажке пару примеров и посмотреть, как они изменяются после шифровки. В принципе, хватит и одного. Тогда можно заметить, что для расшифровки нужно сначала вывести все буквы на нечетных местах зашифрованного слова в правильном порядке, а потом все оставшиеся в обратном. Код задачи может выглядеть, например, так (C++) :

(считаем, что # не содержится в строке)

```
for (int i = 0; i < s.length(); i+=2)
    cout << s[i];

int last = s.length() - 2 + (s.length() - 1)%2;

for (int i = last; i > 0; i-=2)
     cout << s[i];
```

Задача D. Часы.

А вот это уже задача на честное придумывание решения. В задачах на время вообще всегда полезно переводить все в минимально возможные единицы измерения: в дни в задачах про каледари и в секунды/минуты в задачах на время. Поэтому давайте переведем все в минуты и будем думать именно в них. Важно помнить, что в сутках 24*60 = 1440 минут.

Тогда оперировать временем становится чуть-чуть больше. Становится понятно, как решать задачу – давайте узнаем, сколько времени пройдет между первым и третьим временами из условия, умножим его на два (часы же идут в два раза медленнее) и прибавим его ко времени на часах. Получим в точности то, что нужно.

В этой задаче явно будет несколько раз требоваться перевод из формата «часы и минуты» в просто минуты и обратно, поэтому имеет смысл оформлять это как отдельные функции. Вообще, есть негласное правило – если фрагмент кода встречается в программе больше одного раза, его нужно выносить в функцию или процедуру. Это не то правило, которое нужно соблюдать совсем всегда, однако в большинстве случаев так все же стоит делать.

Стоит также учесть тот факт, что конечное время может в ближайший раз наступить уже только в следующих сутках, поэтому разницу между временами нужно считать, помня об этом.
Итак, код программы (снова Python 3):

```
day = 1440
def t_to_m(hour, minute):
    return hour*60+minute

def m_to_t(minutes):
    minutes %= day
    return minutes//60, minutes%60

real_time = t_to_m(x1, y1)
clock_time = t_to_m(a1, b1)
new_time = t_to_m(x2, y2)

diff = 0
if new_time > real_time:
    diff = new_time - real_time
else:
    diff = new_time + day - real_time

clock_time = clock_time + diff*2

a2, b2 = m_to_t(clock_time)

print(a2, b2)
```