1. Binary – 'import numpy as np'

In this challenge, we are given a python script. However, I highly recommend you run this challenge in Linux environment. (As you may notice, we have a line: '*#!/usr/bin/env python3*', this stands for this will be run by Python3. For further information, you may want to search about shebang)

```
#!/usr/bin/env python3

import numpy as np
a = np.array([0], dtype=int)
val = int(input("Oh Oh Oh I got a 0hverflow in my integer!\n"))
if val == -1:
    exit()
a[0] = val
a[0] = (a[0] * 7) + 1
print(a[0])
if a[0] == -1:
    print("VHC2022{" + str(val) + "}\n")\
```

In the hint, we gave that we will have an error of **Integer overflow** in the code, due to no validation on input you parse in. The *NumPy* library is implemented on C programming language, and from the *numpy* documentation, for '*dtype = int*', we will receive a 64-bit integer variable. A 64-bit variable is in range of -(2^63) to (2^63)-1 (from -9223372036854775808 to 9223372036854775807).

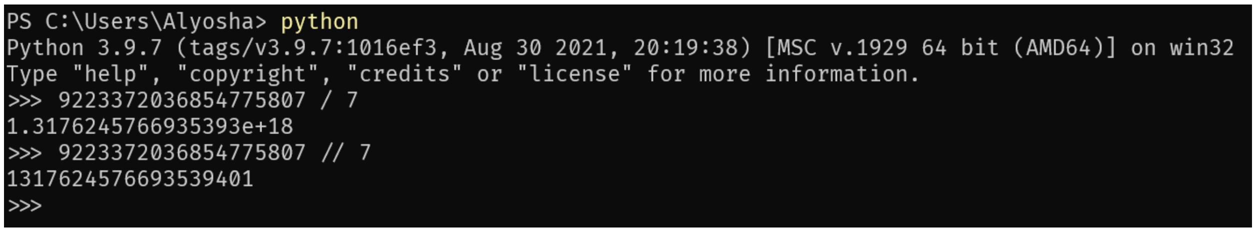
The idea of integer overflow is to parse in a number that goes over the upper/lower bound of data type, and then manipulate it by adding more numbers to make the variable completely 'overflow'.

In this challenge, our target value is -1 (sadly, we can't directly put -1 in). But notice that, our value can multiply by 7 and then add 1. Here is the point they will become an overflow.

In this writeup, I will push the value to 64-int upperbound. The upperbound for a 64-bit int is 9223372036854775807. Let's try this number first.

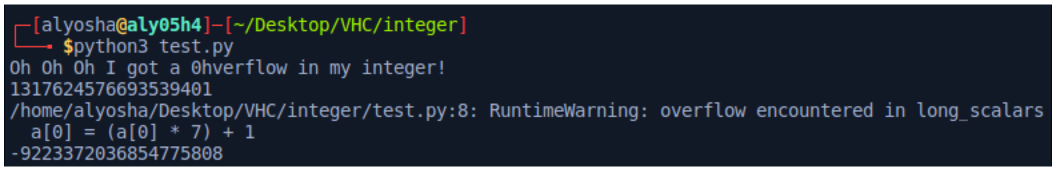
We know that, the input '*val*' will be multiplied by 7, then plus 1. I will try with the number 9223372036854775807 first, to see how far can we go.

You can open a session of python in your terminal and calculate the division.



```
PS C:\Users\Alyosha> python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 9223372036854775807 / 7
1.3176245766935393e+18
>>> 9223372036854775807 // 7
1317624576693539401
>>>
```

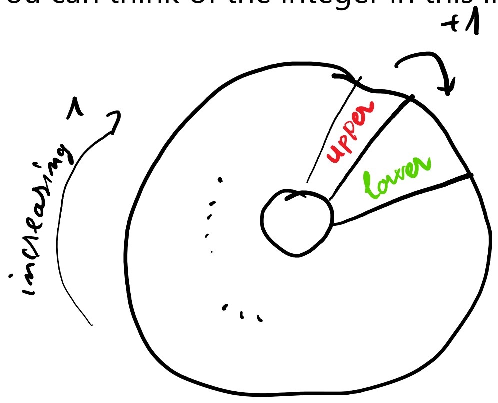
We will try to parse 1317624576693539401 into the program. And see what the current value of a[0] is.



```
┌─[alyosha@aly05h4]─[~/Desktop/VHC/integer]
└──╼ $python3 test.py
Oh Oh Oh I got a 0hverflow in my integer!
1317624576693539401
/home/alyosha/Desktop/VHC/integer/test.py:8: RuntimeWarning: overflow encountered in long_scalars
  a[0] = (a[0] * 7) + 1
-9223372036854775808
```

See something similar? YES! We reached the lower bound of 64-bit integer. But why?

In the easy way, you can think of the integer in this like this roll:



The step $a[0] * 7$ (With a[0] = 1317624576693539401), actually you make the variable go to the upper bound, and then plus 1, so it reached the lower bound. This concepts is not raising errors, just some warnings and the compiler lets you go. This concept is different by language but luckily for us, C used this and most others too.

Back to our problem, we reached the lower bound, -9223372036854775808, and if we can plus 9223372036854775807, we got -1. Because we can't add directly the upper bound value, we need some tricks.

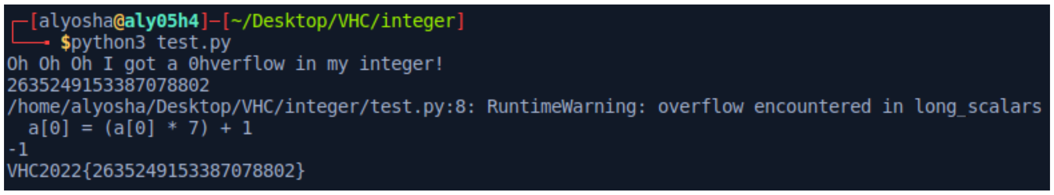
In the earlier, we tried with 1317624576693539401, which can multiply with 7 to reach the upper bound and +1 to reach the lower bound. Let us call this value v.

$$v * 7 + 1 = lowerbound$$

So what about v + v?

$$(v + v) * 7 + 1 = (v * 7 + 1) + (v * 7) = lowerbound + upperbound = -1$$

YES! You got the answer. The number we need to parse in is 2635249153387078802!



```
┌─[alyosha@aly05h4]─[~/Desktop/VHC/integer]
└──╼ $python3 test.py
Oh Oh Oh I got a 0hverflow in my integer!
2635249153387078802
/home/alyosha/Desktop/VHC/integer/test.py:8: RuntimeWarning: overflow encountered in long_scalars
  a[0] = (a[0] * 7) + 1
-1
VHC2022{2635249153387078802}
```