# Hailstone Numbers

*This problem is simple to explain and investigate, but gives some interesting and perplexing results.*

**1. A sequence is defined using the following rules:**

- ***If the previous term is even, divide it by 2.***
- ***If the previous term is odd, triple it and add 1.***

**Follow these rules to complete the next few terms of this sequence:**

6 3 10 _____ _____ _____ _____ _____ _____ _____ …

Write down anything you notice about this sequence.

What makes it different from other sequences you have investigated?

**2. The same rules can be applied to sequences with different starting terms. Try these:**

21 _____ _____ _____ _____ _____ _____ _____ …

17 _____ _____ _____ _____ _____ _____ _____ …

13 _____ _____ _____ _____ _____ _____ _____ …

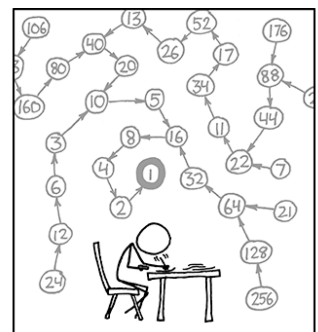
85 _____ _____ _____ _____ _____ _____ _____ …

**3. Mathematician Lothar Collatz reckons that no matter what positive integer you start with, eventually the sequence will reach the number 1.**

How far do you have to continue the sequence when the first term is 7?

7 …

Investigate with your own starting terms, and write down anything you notice.




THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

xkcd.com

"Mathematics may not be ready for such problems" - Paul Erdős

# Hailstone Numbers SOLUTIONS

*This problem is simple to explain and investigate, but gives some interesting and perplexing results.*

**1. A sequence is defined using the following rules:**

- ***If the previous term is even, divide it by 2.***
- ***If the previous term is odd, triple it and add 1.***

**Follow these rules to complete the next few terms of this sequence:**

6 3 10 **5 16 8 4 2 1 4** 2 1 4 2 1 …

Write down anything you notice about this sequence.

What makes it different from other sequences you have investigated?

Terms go up and down (like hailstones in clouds), but finally settle down to repeat 4, 2, 1

**2. The same rules can be applied to sequences with different starting terms. Try these:**

21 **64 32 16 8 4 2 1** 4 2 1 …

17 **52 26 13 40 20 10 5** 16 8 4 2 1 4 2 1 …

13 **40 20 10 5 16 8 4** 2 1 4 2 1 …

85 **256 128 64 32 16 8 4** 2 1 4 2 1 …

**3. Mathematician Lothar Collatz reckons that no matter what positive integer you start with, eventually the sequence will reach the number 1.**

How far do you have to continue the sequence when the first term is 7?

7 **22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1**

1 is the $17^{th}$ term in the sequence, achieved after making 16 steps.

Investigate with your own starting terms, and write down anything you notice.

**Challenge:** *Write a python program to investigate the Collatz conjecture.* For instance, can you write a function which takes a starting number as its input, and returns the number of steps taken to reach 1 as its output? Can you write a loop which uses your function to test a whole range of input values, and print out the results?

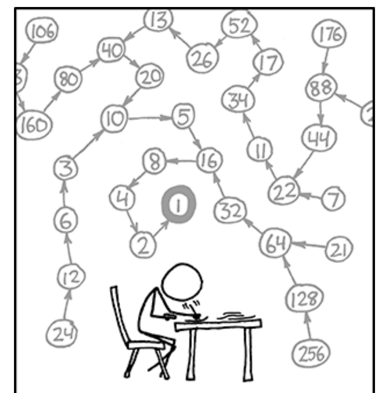
Use your program to answer the following questions:

a) What is the smallest starting number that takes more than 100 steps?

b) What is the largest value you can find that takes fewer than 10 steps?

c) If I start with a 2-digit number, what is the greatest possible value the sequence of numbers could contain, and when does this occur?

d) What if the rules change (eg $\div 3$ if divisible by 3, or $\times 2 + 1$ if not)




THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

xkcd.com

"Mathematics may not be ready for such problems" - Paul Erdős

# Hailstone Numbers *with Python*

**Step 0:** Play around (don't worry about the final structure: just try to get something going).
*Doing some messing about with the problem will let you get a feel for the problem, and try out your ideas.*

| Code | Notes |
|---|---|
| `1: n = 7`<br>`2:`<br>`3: if n % 2 == 0:`<br>`4:    n = n // 2`<br>`5: else:`<br>`6:    n = 3 * n + 1`<br>`7:`<br>`8: print(n)` | Assign a value to n, which we will try to 'Collatz'.<br><br>Set up an **if** function to determine whether our value is odd or even. We can use **% 2** for this.<br>If it's even, divide it by 2, and if odd, × 3 and +1.<br><br>To see the results of your work, use **print**. |

*To consider:*

- What happens when you replace **// 2** (integer division) with **/ 2** (normal division)? Why?
- Add some more **print** statements in other places to see when different bits of the code are run.
- Change the value of **n** (eg make it 22) and see what effect that has on the code.
- For what values of **n** will your code not work? How might we deal with this?

**Step 1:** Plan the overall structure of the program (don't worry about details or syntax yet).
*Answer the questions "What do I want?" and "How should it look?" before answering "How will I do it?"*

- I would like to be able to automate the process, but still be able to get information about the values produced along the way.
- I want to be able to run the same thing repeatedly for many different starting values, and display the results in a meaningful way.
- I want to be able to ask the program to repeat the procedure until the end (reaching a value of 1), for the first thousand positive integers, and print out information about any that, say, took more than 10 steps, or print out the numbers which resulted in the largest value along the way.
- For all the numbers from 1 to 1000, try the Collatz process until you get down to 1, and print out:
```
4: 2 steps (largest value 4)
5: 5 steps (largest value 16)
6: 8 steps (largest value 16)
```

**Step 2:** Break down the problem into separate pieces, and work out their basic details.
*If this is a function, what are the inputs and outputs? If it's a loop, what kind? If it's a list, what goes in it? If it's an accumulator, what is it measuring, how is it initialised, and how is it incremented?*

Firstly, the Collatz process, which needs repeating until the result becomes 1, needs to be enclosed in a loop. I can choose either a **for** loop (fixed number of iterations) or a **while** loop (iterates until a particular condition is met). In this case, since I expect the value to eventually reach 1, and I don't know how many steps it will take, I will choose a **while** loop.

Next, if I am interested in both the number of steps and the largest value achieved, I want both of those to be kept track of during the loop. I can use an **accumulator**, which I could call **steps**, which would be initialised to 0 at the start, then each time the loop has to run, will increase by 1. I also want to keep track of the largest value reached, which I could call **largest**, and initialise it with a value of **n**. Each time the loop creates a new value of **n**, it will need to check to see if it is larger than **largest**, and if so, update **largest** to make it equal to this value of **n**.

All of this needs to be carried out for any starting value I give it. If I want to run the program again for every value, this would be fine, but if I want to be able to test lots of values automatically (eg all the

numbers from 1 to 1000), I should enclose everything so far in a function. Call it **collatz** with an input value of **n**, and the output will need to be *two* values: **steps** and **largest**. These will form a **tuple** (which is a single object which can contain multiple other objects, a bit like a list: it is a convenient way to get a function to return more than a single result).

Now the Collatz process has been encapsulated in the **collatz** function, I just need to make a loop to cycle through some starting values, and print the results. I can use an **f-string** to print something that incorporates both fixed text and values from the function (by enclosing the names of variables or calculations in **{**curly braces**}** I can incorporate their values into the output string that **print** produces).

**Step 3:** Write the code. Focus on each bit separately, to make sure it does its job properly.
*If each bit works well, and the structure is in place to let them work together, the whole thing will work.*

```
 1: def collatz(n):
 2:   steps = 0
 3:   largest = n
 4:   while n > 1:
 5:     if n % 2 == 0:
 6:       n = n // 2
 7:     else:
 8:       n = 3 * n + 1
 9:     steps += 1
10:     if n > largest:
11:       largest = n
12:   return steps, largest
13:
14: for i in range(1, 100):
15:   steps, largest = collatz(i)
16:   print(f"{i}: {steps} steps
(largest value: {largest})")
```

Define the function at the beginning, so when you refer to it later, Python already knows it exists.
Initialise the values of **steps** and **largest** before the loop begins (otherwise they'll be reset each time).
Loop for as long as it takes for **n** to drop to 1.
Run the procedure (this snippet is from step 0).

Each time the **while** loop runs, increment the number of steps by 1 (note: `steps += 1` is equivalent to `steps = steps + 1`.
Whenever **n** is recalculated, compare to **largest** and replace **largest** with the value of **n** if needed.
Both **steps** and **largest** are returned as outputs.
The main loop runs from 1 to 99, finding the value of **steps** and **largest** from the **collatz** function.
An **f-string** incorporates these values for **print**.

*To consider:*

- Can you add a condition in the **collatz** function to deal with a bad input like 0 or a non-integer?
- Instead of just keeping track of the largest value, use a list, initialised with `values = []`, and add values to it each time through, using `values.append(n)`. At the end, return **values** instead of **largest** and you can print out all the numbers in the sequence, which can give more insight into what's going on.
- Why not analyse a larger range, but add a condition before the **print** statement so that you only display the most interesting values, such as those that take more than 50 steps, or those where the largest value achieved is greater than 1000?

**Step 4:** Make improvements. Do you have more questions to answer? Can we display the results better?
*Once you've got code that works, then you can start tweaking, adding features, and improving robustness.*

One nice improvement you can make involves importing the **matplotlib** module to produce a graph:

```
 1: import matplotlib
 2: import matplotlib.pyplot as plt
 3:
 4: values = [5, 16, 8, 4, 2, 1]
 5: plt.plot(values)
 6: plt.savefig("collatz5.png")
```

Importing this module lets us use its built-in graphing functions. Specifically, the **pyplot** bits.
You can use a **values** list generated by the code.
Plotting values loads them into a graph object, and then using **savefig** lets you view the image.

*To consider:*

- You can embed the **plt.plot** line within a loop, and lots of **values** lists will be added to the same plot. You could view the first 100 all super-imposed on the same set of axes.