Collatz Conjecture Prover

Most people don't know what the **Collatz Conjecture** is (they should, it's pretty cool), so here's a basic introduction:

1. Pick any positive integer.
2. If it's positive, multiply it by 3 and ad 1. If it's even, divide it by 2.
3. Repeat steps 2-3 endlessly—you'll always eventually end up with 1, no matter the number.

There's actually a very interesting story behind this conjecture. In short, the conjecture was introduced in 1937 by **Lothar Collatz**, a German mathematician. It's a very interesting theory, and it seems very intuitive, but surprisingly our current system of mathematics is incapable of proving it. In fact, considering it gained popularity during the height of WWII, many believed that it was proposed by Germany to distract western scientists and mathematicians from contributing to the war—even leading into the modern era, countless mathematicians have devoted their lives towards the Collatz Conjecture, yet still it remains unproven. If you're interested in learning more about it, watch this video by Veritasium.

In short, Mathematicians have shifted their efforts towards checking as many numbers as possible—at first through manual calculations, but later through the use of computers. Today, we will be writing a simplified version of the programs used to crunch out the numbers of this theorem, showing how revolutionary a very elementary understanding of programming can be.

First, let's begin by defining the number we're testing.

```
1 void main() {
2    int n = 2;
3 }
```

As you can see, I defined our variable to be tested as "n". By doing this, we have successfully completed step 1 of the Collatz Conjecture. Before progressing to step 2, we need to first learn about a new operator: the **modulus operator**.

The modulus operator is written using a "%" symbol and simply returns the remainder of one number after being divided by another. For example:

```
1 void main() {
2   print(3 % 2);
3 }
```

Output: 1

In the program above, the value “1” was returned, which tracks considering that the remainder of 3 / 2 is 1. As a result, the modulus operator is most often referred to as the **remainder operator**. Feel free to check the modulus of other values on your own, but just know that the modulus will be relevant in checking if a number is even for step 2.

Think about it: the remainder of any number divided by two will be 0 if the number is even, and 1 if the number is odd. As a result, we can check is a number is even using the following code:

```
void main() {
  int n = 2;
  if(n % 2 == 0){
    print("Number is even!");
  }
}
```

Output: Number is even!

Additionally:

```
1 void main() {
2   int n = 3;
3   if(n % 2 == 0){
4     print("Number is even!")
5   } else {
6     print("Number is odd!");
7   }
8 }
```

Output: Number is odd!

Now that we have an idea on how to check is a *n* is even or odd, we can complete step 2 of the Collatz Conjecture:

```
1 void main() {
2   int n = 3;
3   if(n % 2 == 0){
4     n = n / 2 as int;
5   } else {
6     n = 3 * n + 1;
7   }
8
9   print(n);
10 }
```

Output: 10

A couple of reminders:

- The quotient of two numbers will return a double, so it needs to be cast into an integer.
- The Order of Operations, PEMDAS, applies to mathematics within coding.

Time to move on to step 3: repeating step 2 until we eventually reach n = 1. This could be accomplished by manually inputting the result of each run, *or* we could have the program do it automatically. The best way to accomplish this would be to implement a **loop**.

Loops in programming do as they say: they loop code over and over again. There are many different kinds of loops in programming, but the one we will be using here is called a **iteration statement** (AKA a **while-loop**). While loops simply continue to loop code until a certain, specified condition is met. They are declared much like a conditional / if statement, except use the word “while” instead of “if”. Knowing this, we can use one of these while loops to do step 3 manually for us, like so:

```
void main() {
  int n = 3;

  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2 as int;
    } else {
      n = 3 * n + 1;
    }
  }

  print(n);
}
```

Output: 1

And now, we've completed the Collatz Conjecture and have the ability to test any positive integer we can think of. Try it yourself, it'll always end up back at 1! This is a great example of how quick programs can run calculations, even numbers in the billions will be done in under a second!

Now, let's take this even further. We have the ability to test one number, but how about testing multiple numbers? For that, we'll use another while-loop in addition to a new comparison operator: the **less-than operator**. Referenced using "<", the less than operator is much like an equivalent operator "==", except it checks if one number is less than another.

By implementing this with a new counting variable, we get this:

```
void main() {
  int i = 1;
  while (i < 11) {
    int n = i;
    while (n != 1) {
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }

    i++;
  }

  print("All are proven.");
}
```

Output: All are proven.

To highlight the key elements here:

1. *i* is defined as an integer initially at i = 1.
2. An additional while-loop was added that runs while *i* < 11.
3. *i* is incremented by 1 each time the loop runs—that's what "++", the **increment operator**, means.

This is an extremely common setup you'll see a lot in computer science. In fact, it's used so often that it was even turned into its own type of loop: a **for-loop**.

A for-loop is an easier way to write what we've just programmed. By combining those three key elements (definition of value, condition for loop, and loop change), we can define a for-loop in one statement:

```
void main() {
  for (int i = 1; i < 11; i++) {
    int n = i;
    while (n != 1) {
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }
  }

  print("All are proven.");
}
```

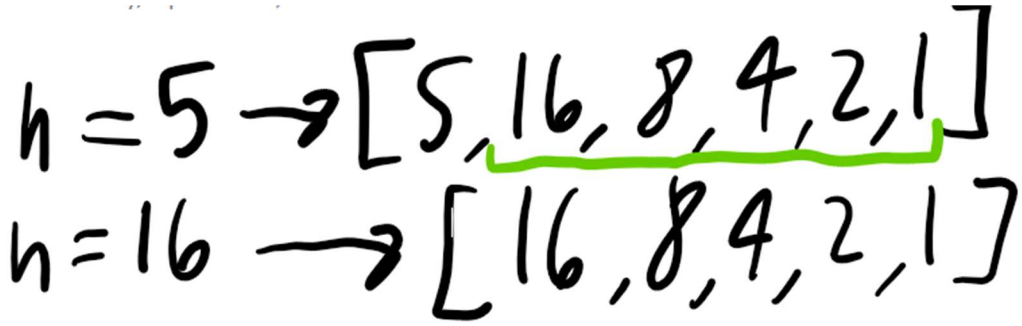
Output: All are proven.

Notice how it worked exactly the same way, it just looks simpler on our end.

Now, we have a simple program which proves the Collatz Conjecture to a given number. To change the starting and ending number, just change the initial definition of *i* and the number *i* is less than respectively.

Now that we have this basic program, it's time for optimization!

Let's think about what the Collatz Conjecture is, really: a long, branching list of positive integers, with the last entry being 1. Thus, by working through the conjecture, you're really just defining one really long chain of numbers, for example:



By proving the Collatz Conjecture for 16 in this example, you're not only proving it for each

additional number which contains an number from 16's list within its own (i.e. 5). Thus, if we were to keep track of all the numbers, we've proven the conjecture for, including the ones we stumble across in the process, we could prove the conjecture once we come across any number from that list.

But how do we keep track of all of the numbers we've proven? We can't just manually define a variable for each one!

That's where **data sets** come in. Data sets are collections of values defined under 1 variable. There are many different types of data sets, but the basic one we'll be using is a **list**. A list is just that: a list of values, with nothing fancy or complicated about it. Let's go ahead and define a list in our program:

```
1  void main() {
2    List proven = [1, 2, 3];
3
4    for (int i = 1; i < 11; i++) {
5      int n = i;
6      while (n != 1) {
7        if (n % 2 == 0) {
8          n = n / 2 as int;
9        } else {
10         n = 3 * n + 1;
11       }
12     }
13   }
14
15   print("All are proven.");
16 }
```

Output: All are proven.

As you can see, a list is defined using brackets, with entries specified within them separated by commas. 1 is proven by default, and we've already proved 2 and 3 ourselves, so we're safe to include them in there.

Now, let's go ahead and use our list. We can check a value in a list using brackets an the **index** of the value within, so we can check if *n* equals the first value of *proven* like this:

```
void main() {
  List proven = [1, 2, 3];

  for (int i = 1; i < 11; i++) {
    int n = i;
    while (n != proven[0]) {
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }
  }

  print("All are proven.");
}
```

Output: All are proven.

Considering the first value of *proven* is 1, nothing changes here. Notice how the first value of our list has an index of 0. That's because, to computers, the first number is 0, so

0 = first value

1 = second value

2 = third value

...and so on.

Now, in this example, we're only checking the first value of *proven*, when we really need to check all of them. We could manually check all values using another for-loop, or we could use a function to make it simpler.

A List is an example of an **Object**. An object is a type of variable in programming which has the ability to store multiple values and attributes. A great example of an Object would be a String: a String itself is not just one value, but really a string of individual character values—which is how it gets its name. We'll learn more about objects later, but the reason we're discussing them now is because of **object functions**. Object functions are functions stored as part of an object and frequently reference or alter their own values. The

different values, attributes, and functions of an object can be referenced by placing a "." after an object (i.e. *proven*.length giving the length of our list).

The object function we're concerned with is *proven.contains()*, a function which checks if an inputted value is within the list. Knowing this, we can modify our code to:

```
void main() {
  List proven = [1, 2, 3];

  for (int i = 1; i < 11; i++) {
    int n = i;
    while (!proven.contains(n)) {
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }
  }

  print("All are proven.");
}
```

Output: All are proven.

Now, our while-loop checks if *proven* does not contain n before looping, referencing all values within *proven* and providing a great time reduction. To make it even more efficient, let's now add each value *n* reaches using the object function *proven.add()*, all to form that chain of values we discussed earlier:

```
void main() {
  List proven = [1, 2, 3];

  for (int i = 1; i < 11; i++) {
    int n = i;
    while (!proven.contains(n)) {
      proven.add(n);
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }
  }

  print("All are proven.");
}
```

Output: All are proven.

It now works even faster. However, we're lying to ourselves here. When we add *n* to *proven*, we haven't really proven *n* yet. Instead, what we need to do is record all values *n* reaches in a separate list before it's proven, and then add them to our main list once proven. We can add all values from one list to another using the object function *proven.addAll()*. Knowing this, we can alter our code to become:

```
void main() {
  List proven = [1, 2, 3];

  for (int i = 1; i < 11; i++) {
    List steps = [];
    int n = i;

    while (!proven.contains(n)) {
      steps.add(n);
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }

    proven.addAll(steps);
  }

  print("All are proven.");
}
```

Output: All are proven.

Now we're cooking! Except, how do we know that our list works properly? We can't see what's going on behind the scenes, so we have no idea if it's working as intended. To fix this, let's run the program again, but print *proven* to check:

```
void main() {
  List proven = [1, 2, 3];

  for (int i = 1; i < 11; i++) {
    List steps = [];
    int n = i;

    while (!proven.contains(n)) {
      steps.add(n);
      if (n % 2 == 0) {
        n = n / 2 as int;
      } else {
        n = 3 * n + 1;
      }
    }

    proven.addAll(steps);
  }

  print(proven);
}
```

Output: [1, 2, 3, 4, 5, 16, 8, 6, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 9, 28, 14]

Works like a charm! Now, we're able to prove the Collatz Conjecture up to a given number in the most efficient way possible.

However, the idea is for this program to run endlessly—that way, we can keep proving it to larger and larger numbers. Currently, we set the maximum number manually, so it'll always stop after a certain time. We need to find a way for it to run endlessly...

Let's look back at the wording for the steps for ideas:

1. Pick any positive integer.
2. If it's positive, multiply it by 3 and ad 1. If it's even, divide it by 2.
3. Repeat steps 2-3 endlessly—you'll always eventually end up with 1, no matter the number.

Notice how in step 3, it says to repeat steps 2-**3.** Think about it: what happens when you repeat step 3 while in step 3? You get an infinite loop: exactly what we're after! Replace "step" with "function", and now we know how to create an endless loop: repeating a function within a function. Now, let's get rid of the for-loop (we won't be needing that anymore) and try the repeating function trick:

```
void main() {
  List proven = [1, 2, 3];
  int n = 1;

  List steps = [];
  while (!proven.contains(n)) {
    steps.add(n);
    if (n % 2 == 0) {
      n = n / 2 as int;
    } else {
      n = 3 * n + 1;
    }
  }

  proven.addAll(steps);
  print(proven);

  main();
}
```

Output: Uncaught RangeError: Failed to execute 'postMessage' on 'Window': Maximum call stack size exceeded

What a scary error message! Without getting into the complex details, just know that this error message is basically saying "The computer is doing too much!"

Somewhere out there in the world, a computer just almost exploded, as it was doing nothing but straight calculations for too long of a time. We need to give it a break! We can do that like this:

```dart
void main() async {
  List proven = [1, 2, 3];
  int n = 1;

  List steps = [];
  while (!proven.contains(n)) {
    steps.add(n);
    if (n % 2 == 0) {
      n = n / 2 as int;
    } else {
      n = 3 * n + 1;
    }
  }

  proven.addAll(steps);
  print(proven);

  await Future.delayed(Duration(milliseconds: 1));

  main();
}
```

What we've done here is made the *main* function **asynchronous** (meaning not instantaneous) and added a brief **delay** at the end before running again. This is very complex conceptually and notationally, so we won't get into that later.

The important thing is: it works! What we've done here with looping functions is called **recursion**. Recursion is the act of calling a function while defining that same function, creating a loop. Recursion is very dangerous—you saw that with the scary error message. However, when done effectively, it can be really useful.

One flaw with our current program: it keeps outputting the same thing. That's because we keep on redefining our variables when we start the main function again, because they're all local variables. To fix this, we need to make them **global variables**. Recall that variables "cannot escape the curly braces they're defined in." The solution to that is to move their definition outside of any curly braces / function, making them global variables. To alter out code:

```dart
List proven = [1, 2, 3];
int n = 1;

void main() async {
  List steps = [];
  while (!proven.contains(n)) {
    steps.add(n);
    if (n % 2 == 0) {
      n = n / 2 as int;
    } else {
      n = 3 * n + 1;
    }
  }

  proven.addAll(steps);
  print(proven);

  await Future.delayed(Duration(milliseconds: 1));

  main();
}

```

We're almost there! We've converted those variables to global variables, but we're not changing them. As a result, we're just proving it for 1 over and over again. Let's increment n after each loop and reintroduce *i* as a temporary variable:

```dart
List proven = [1, 2, 3];
int n = 1;

void main() async {
  List steps = [];
  int i = n;
  while (!proven.contains(i)) {
    steps.add(i);
    if (i % 2 == 0) {
      i = i / 2 as int;
    } else {
      i = 3 * i + 1;
    }
  }

  proven.addAll(steps);
  n++;

  print(proven);

  await Future.delayed(Duration(milliseconds: 1));

  main();
}

```

The code works fine now, except your code probably just crashed the website. That's not because of the calculations the computer's doing—that works just fine. It's because it keeps printing everything.

This isn't obvious at first, but printing requires a ton of resources. It requires communication with the OS, writing instead of calculating, and, in our case, communication over the internet. Thus, we shouldn't print too often, but with this code, we're telling the computer to print every millisecond until infinity.

Now, go ahead and restart the page, and let's fix this.

To fix this, we'll need to go full circle and use the modulus again. Instead of printing every time we loop, how about we only print every 1,000 times we loop:

```dart
List proven = [1, 2, 3];
int n = 1;

void main() async {
  List steps = [];
  int i = n;
  while (!proven.contains(i)) {
    steps.add(i);
    if (i % 2 == 0) {
      i = i / 2 as int;
    } else {
      i = 3 * i + 1;
    }
  }

  proven.addAll(steps);
  n++;

  if (n % 1000 == 0) {
    print(proven);
  }

  await Future.delayed(Duration(milliseconds: 1));

  main();
}
```

Now, the code runs without the computer having a panic attack.

Except, you might be having a panic attack when you see that long list of numbers printed! Instead, how about we just print the number we've proven the Collatc Conjecture to: *n*:

```dart
List proven = [1, 2, 3];
int n = 1;

void main() async {
  List steps = [];
  int i = n;
  while (!proven.contains(i)) {
    steps.add(i);
    if (i % 2 == 0) {
      i = i / 2 as int;
    } else {
      i = 3 * i + 1;
    }
  }

  proven.addAll(steps);
  n++;

  if (n % 1000 == 0) {
    print(n);
  }

  await Future.delayed(Duration(milliseconds: 1));

  main();
}

```

That's better! Now, we're so close to being done! All we need to do is add a bit of polish to the output, comment our code so we know how it works, and we're golden!:

```dart
// List of numbers proven
final List proven = [1];
// Current number being proven
int n = 1;

// Main function - asynchronous for delay
void main() async {
  // Temporary list of numbers gained from steps
  final List steps = [];
  // Temporary value of number
  int i = n;
  // Loops until tested value within proven list
  while (!proven.contains(i)) {
    // Adds temp value to temp list
    steps.add(i);
    // If even, divide by 2; if else, multiply by 3 and add 1
    if (i % 2 == 0) {
      i = i / 2 as int;
    } else {
      i = 3 * i + 1;
    }
  }

  // Add all temp values to proven list
  proven.addAll(steps);
  // Increment n
  n++;

  // If 1000th proven, print notification
  if (n % 1000 == 0) {
    print("Proven to $n");
  }

  // Delay for 1 millisecond
  await Future.delayed(Duration(milliseconds: 1));

  // Loop function = recursion
  main();
}
```

And we’re done! Good job!