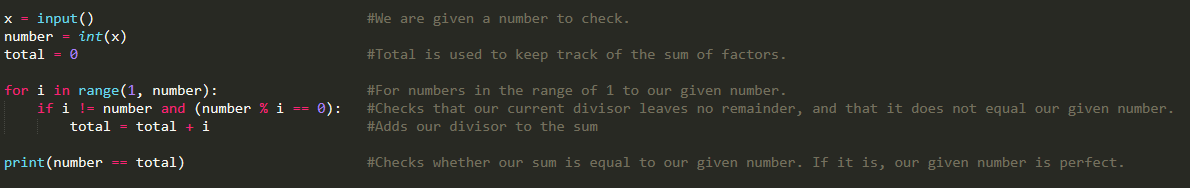
**Prolog Assignment 2020 (Resit)**

*Cameron Fitzpatrick 18493006*

*I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work.*

I began by laying out the things we know about “perfect” numbers. We know that a perfect number is equal to the sum of its factors, not including itself. We know that the first divisor we check is 1, which will always be a factor. We need to keep track of the sum of the factors also, which will start at 0.

Next, I laid out the steps that are needed in order to check if our given number is perfect. The first check in our “loop” should be making sure that our current divisor is not equal to the original number we are given, as a perfect number is equal to the sum of its factors NOT including itself. We then check that the divisor we are currently using fits into our given number without any remainder. If this is the case, we can add the value of this divisor to our sum. Finally, before we loop back around, we iterate the divisor we are using by 1 and repeat the process. After writing this process up in Python, we can see these processes laid out.

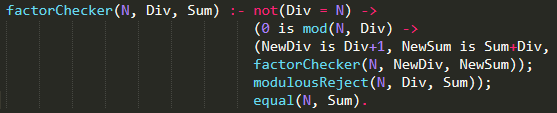
I began constructing this same process in Prolog. I think the best way to illustrate my methodology is to split the python version I coded above into sections and compare it to the Prolog equivalent.

From here on out, I’ll be using these abbreviations for the sake of decluttering:

* N = Our given number that is being tested.
* Div = Our current divisor being checked.
* Sum = The current sum of factors.

**Step 1: Establishing our components.**

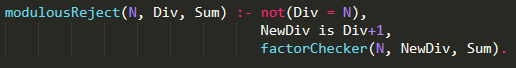
****Above we have our Python version on the left, along with the Prolog one on the right. The ****user of our Prolog program will use the query perfect(N) to check their number. When the query is used, it takes the N value provided and sends us to our factorChecker rule. Our factorChecker rule establishes that the divisor we begin on is 2, and that our beginning sum is 1.

**Step 2a: Checking if our divisors are factors.**

Our factorChecker rule takes 3 arguments: our N, Div and Sum. The not(Div = N) check will pass only when our Div does not equal our given number. The arrow that follows it acts as an “if, then, else” command. If our current Div equals our N, we must have checked all the possible divisors and so we are sent to our equal() rule. Otherwise, the “(0 is mod(N, Div)” rule checks the modulus of our N divided by our Div and will only pass if that modulus is 0. We want the modulus to be 0 because that means that the divisor we are currently checking is a factor of our given number, and so we must add it to the sum. If the modulus passes, our current div is incremented by 1, and our previous div is added to the sum. We are then sent back to the start of this loop to repeat it with the new Div value and the new Sum value.

Should our modulus check fail, we will be sent to our modulusReject() rule, which will increment our Div by 1.

**Step 2b: A divisor that is not a modulus of 0.**

****

Should our current Div not have a modulus of 0 when divided into N, we are sent to the modulusReject() rule. We start with “not(Div = N)” which we’ve seen in factorChecker(). This will only pass and allow us to continue if our Div is not equal to N, because if it did, we have no need to progress to more divisors. Should the not() pass, we create a newDiv that is our current Div incremented by 1 and get send back to factorChecker() with these new arguments.

**Step 3: Is our N perfect?**



The equal() rule is the last part of our program, and we are sent here once all the divisors of our N are checked. If our Div is equal to N, it means we have tested all the possible divisors. We will be sent to our equal rule, which checks if our Sum is equal to N. If Sum = N, then our rule will return true as that means we have a perfect number. If Sum != N, our rule will return false as we do not have a perfect number, because for a number to be perfect it must be equal to the sum of its divisors not including itself.

===========================================================================

**TESTING**

To test the program, I ran the first five perfect numbers through it (6, 28, 496, 8128, 33,550,336). They all passed as perfect numbers. Next, I chose ten random numbers between 1-1000 (429, 121, 143, 444, 45, 760, 711, 484, 120, 572) and tested them. All came back false, as none are perfect numbers. I then began testing more fringe cases, like 0 and some negative numbers. I found that this put my program in an infinite loop. Details of how I fixed this issue can be found in the bug fixes section.

===========================================================================

**BUG FIXES**

|  |  |
| --- | --- |
| **BUG** | **FIX** |
| Program loops when presented with an N less than 1. | Fairly simple fix here, just had to add a clause to our perfect rule that says N must be greater than 1. 0 is not a perfect number and no perfect number is a negative. |
| Program outputs that 1 is a perfect number | The solution to our previous bug can be used, just increasing the threshold for N to 2 instead of 1. This leads 1 to be automatically rejected as not a perfect number. |
|  |  |

**EFFICIENCY IMPROVEMENTS**

Currently, our program is checking every number in the range 1 -> N to find factors. While not noticeable for the first few perfect numbers, this can lead to extremely long execution times for our programs. Any time we can shave off this is welcomed, as no one likes waiting around for a result. Let’s revisit what we know about perfect numbers. We know that in order to be perfect, a number must be equal to the sum of its factors not including itself. With this in mind, we can say that the very last factor for any given number must be half of the given number, as any number above that cannot divide into N without a remainder. By putting this rule into our program, we can essentially half the runtime. In the picture below we can see that all we must do is modify the not(N = 0) condition in our factorChecker() and our modulusReject().



This new rule will only pass if the current Div we are checking is not more than half of our given number. This saves us checking the rest of the numbers up to N, as we already know they cannot be factors. With this change, we can already see a big improvement. When checking perfect(33,550,336) without these improvements, the approximate runtime was about 47 seconds. With our newly added parameters, we halved the runtime to approximately 27 seconds.

===========================================================================

After reading more about perfect numbers, I came across some interesting info regarding odd numbers. Of the 51 known perfect numbers, all of them are even. Its widely thought that there is no such thing as an odd perfect number. All the numbers up to 101500 have been tested, returning no odd perfect numbers, so it seems extremely unlikely that any exist. With this knowledge in mind, it seems fair to say we can assume any number given to the program that is odd can be returned as false. Seen below, if we add “(0 is mod(N, 2)” into our perfect rule, we can rule out any odd numbers and immediately return them as false.

**CITATIONS/REFERENCES**

Perfect Number Wikipedia - <https://en.wikipedia.org/wiki/Perfect_number>

Mathworld website which mentions the 101500 numbers tested <https://mathworld.wolfram.com/PerfectNumber.html>

List of perfect numbers - <https://en.wikipedia.org/wiki/List_of_perfect_numbers>