William Watkins

CS 315 Final Report

12/19/2019

Perfect Numbers, Kaprekar Numbers, and Lucky Numbers

Many lessons have been taught and many examples have been given this semester in CS 315, but the most valuable educational asset I have acquired is efficient programming. Previous classes have proclaimed its importance, but the reason was not addressed as clearly as in this class. I have written programs to find perfect numbers, Kaprekar numbers, and lucky numbers as an exercise to show how I have learned to practice efficient programming. I have used my knowledge and logic to write these programs with minimal help from outside sources. All sources used have been cited. With some of my programs demonstrating inefficient qualities, I will examine others' programs and discuss some ways in which the differences in the programs demonstrate better practice.

An aliquot sum is the sum of a number's divisors excluding the number itself. When a number's aliquot sum equals the number itself, it is a perfect number. For example, the divisors of 6 are 1, 2, 3, and 6. For its aliquot sum, we add 1 + 2 + 3 = 6. Since its aliquot sum equals the number itself, it is a perfect number. In order to write an algorithm to find perfect numbers, I looked to Euclid's discovery in which (q (q + 1)) / 2 is an even perfect number whenever q is a prime of the form $2^p - 1$ for a prime p. When a number exists in this form, it is called a Mersenne prime.

I wrote a program that would find perfect numbers. Given the above information, I decided to use an unsigned long long data type, because that is the largest integer value type possible in C++. The program was designed to check all numbers in a range 1 to 2^n, where n is a positive integer $2 < n < 32$. If the n was prime, it would store q as $2^n - 1$, then carry out the operation p = (q (q + 1)) / 2 and print p as a perfect number and n as the prime. There is error in this method.

My initial thought was to find a prime number, it would be a number such that could not be divided by 2 or 3. This idea is only partially correct because numbers whose divisors are prime do not necessarily have 2 or 3 as their divisors. For example, the divisors of 25 are only 1, 5, and 25. Since my program was including 25 as a prime number, I added 25 to my list of divisors. This is a poor tactic because as the numbers get larger, more prime numbers would have to be added to the list of divisors to check for. As I continued testing my program, it was including numbers that were not on an official list of perfect numbers. One of which was 2096128. I found its divisors to be 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2047, 4094, 8188, 16376, 32752, 65504, 131008, 262016, 524032, and 1048064, and their sum is 2096128. The prime selected here was 11. Either something was wrong, or my program had discovered a new perfect number. The error was that I was checking powers of 2 for divisors, and not primes. $2^{11} - 1 = 2047$, which is not a Mersenne prime. I had missed 23 * 89 = 2047, therefore, 23 and 89 are divisors of 2096128, so its aliquot sum is not equal to itself, therefore it is not a perfect number.

The solution here is to check if n is prime, then check if $2^n - 1$ is prime. I had to search and borrow some code to write a function to check if a number is prime, and the correction and addition of another check fixed the error, and my code would then run without error, producing correct results.

Kaprekar numbers are those whose digits of its square may be split up, then added together to equal the number itself. For example, $9^2 = 81$, and 8 + 1 = 9, so 9 is a Kaprekar number. $297^2 = 88209$, and 88 + 209 = 297, so 297 is a Kaprekar number. The position of the split is ambiguous, though. Some numbers are debatable as to whether they are Kaprekar numbers or not. $5292^2 = 28005264$. 28 + 5264 = 5292, but 2800 + 5264 ≠ 5292, so there are two lists present on The On-Line Encyclopedia of Integer Sequences, one that includes this number and others like it, and another list that does not include them. My program was not designed to find those numbers.

One number that shares a similar property is 9. The sum of the digits from the product of 9 and another integer from 1 to 9 equals 9. For example, 9 * 3 = 27, and 2 + 7 = 9. I found that all Kaprekar numbers divided either evenly by 9, or had a remainder of 1, so the first comparison in my program was if the square of the number modulo 9 is equal to 0 or 1. The program then counts the digits of the square, then determines how many digits will be included in the left number and right number. For the left number, we can chop digits from the right by using /= 10. Since the square is a unsigned long long integer, it does not include decimals, do we do this the same number of times as there are digits in the right number. To find the number on the right, I reversed the order of digits in the number, then chopped digits from the end just like the left number, then reversed the order of digits again. Initially reversing the number would disregard zeros that were initially at the end of the number, though, putting them in the front of the number, so I had to count the zeros at the end of the number before reversing and chopping, then put them back onto the number after reversing and chopping by multiplying it by 10 as many times as there were tailing zeros. The program then finally checks if the sum of the right and left numbers is equal to the initial number and prints it if it is so.

My code seems long and the program runs slower as the numbers get larger. This is an undesirable quality for a program to have. I was able to find a program online written by Nigel Galloway in 2012, and it was categorized as “Casting out Nines.” His code is considered fast, and it includes the example mentioned earlier, 5292^2, and similar cases. The code also prints out that the number is a member of the residual set 1 or 0, which the 1 or 0 is the same as the square of the number modulo 9. An issue that I see with his code, however, is that he used “unsigned long long int k = pow((double)Base, nz -),” which I had to use bitshifting because the pow() function was returning a double and not an unsigned long long integer, and it produced a warning.

My third program consisted of finding lucky numbers. Lucky numbers are those left remaining after starting with a sequence of positive integers, striking out those that are of each index interval of the first integer greater than 1. Starting with a sequence of increasing integers, the first integer greater than 1 is 2, so we strike out all second integers, leaving us with only odd numbers. Now the next integer greater than 1 is 3, so we strike out every third integer, and we continue the process until we have less numbers in our sequence than the greatest number.

Given an array, we need to strike out places on indexes determined by the values left after each iteration. Using the array a[1000], we start with a while loop. Starting with index position 1, the while loop will iterate over the array. For each index, it will check if the number has been stricken out. If not, the value of that index gets set to a variable. Then the array is parsed, striking out those values remaining in each index as determined by the first iteration loop.

I did not use references for this program, only logic and trial and error. Pencil and paper were the best tools for correcting my original process. This program took longer than the others to write, and

it is the shortest. The program produces the correct results as compared to The On-Line Encyclopedia of Integer Sequences list A000959. The list only goes up to 303, however, and the greatest integer in the results of the program is 997.

I have written these programs to be as efficient as I could make them given how I imagine the programming should be. There is always room for improvement, and I have found an example of a faster program for the Kaprekar numbers. Practicing efficient coding is now my main goal when it comes to programming now. The importance of it is now clear to me after taking CS 315.

References:

Perfect Numbers:

- https://www.britannica.com/science/perfect-number
- https://www.geeksforgeeks.org/aliquot-sum/
- https://byjus.com/maths/perfect-numbers/
- https://mathcs.clarku.edu/~djoyce/java/elements/bookIX/propIX36.html
- https://oeis.org/A000396/list

Kaprekar Numbers:

- https://onlinejudge.org/external/9/974.pdf
- https://cs.uwaterloo.ca/journals/JIS/VOL3/iann2a.html
- http://mathworld.wolfram.com/KaprekarNumber.html
- http://oeis.org/A053816
- http://oeis.org/A006886
- https://rosettacode.org/wiki/Kaprekar_numbers#Casting_Out_Nines_.28fast.29

Lucky Numbers:

- http://mathworld.wolfram.com/LuckyNumber.html
- http://oeis.org/A000959/list
-