Assignment 8
Perfection, Deficiency and Abundance

Prof. Darrell Long

Winter 2023

1 Introduction

IN NUMBER THEORY, a perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself (called its *proper divisors*). For instance, 6 has divisors 1, 2 and 3 (excluding itself), and $1+2+3=6$, so 6 is a perfect number.

The sum of divisors of a number, excluding the number itself, is called its *aliquot sum*, $s(n)$ of a positive integer $n>0$ is the sum of all *proper divisors* of n, that is, all divisors of n other than n itself. When we write $d \mid n$ we mean that $n \div d$ is an integer with no remainder, and when we write $d \nmid n$ then $n \div d$ has a remainder. So we define $s(n)$ as,

$$s(n) = \sum_{d|n,\ 0<d<n} d.$$

A perfect number is one that is equal to its aliquot sum. Equivalently, a perfect number is a number that is half the sum of all of its positive divisors including itself; in symbols, $\sigma_1(n) = 2n$ where σ_1 is the sum-of-divisors function. That is,

$$\sigma_1(n) = \sum_{d|n,\ 0<d\leq n} d.$$

For instance, 28 is perfect as $\sigma_1(28) = 1+2+4+7+14+28 = 56 = 2\times 28$. When you see the Σ symbol (or the Π symbol), you should think of a `for` loop.

In about 300 BC, Euclid showed that if 2^p-1 is prime then $2^{p-1}(2^p-1)$ is perfect. The first four perfect numbers were the only ones known to the early Greek mathematicians, and the mathematician *Nicomachus* noted that 8 128 was perfect as early as around AD 100.

In modern language, Nicomachus states without proof that *every* perfect number is of the form $2^{n-1}(2^n-1)$ where 2^n-1 is prime. He says of perfect numbers, "There is a method of producing them, neat and unfailing, which neither passes by any of the perfect numbers nor fails to differentiate any of those that are not such, which is carried out in the following way." He then goes on to explain a procedure which is equivalent to finding a *triangular number* based on a Mersenne prime. He seems to be unaware that n itself has to be prime. He also says (wrongly) that the perfect numbers end in 6 or 8 alternately. (The first 5 perfect numbers end with digits 6, 8, 6, 8 and 6; but the sixth also ends in 6.) It is important that you understand that

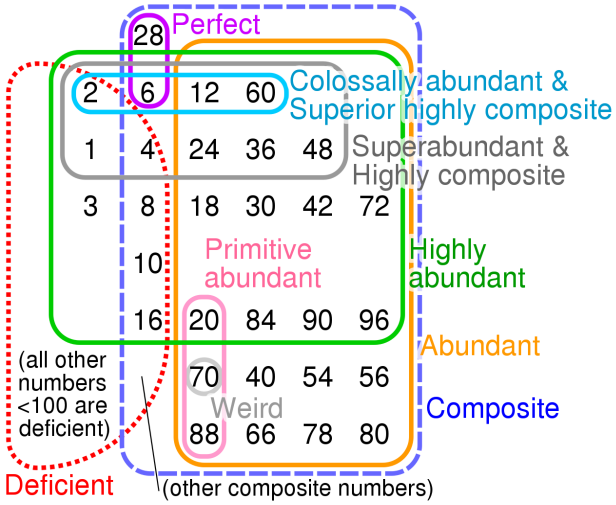

Figure 1: Euler Diagram of $1, \ldots, 100$

not every number of the form $2^{n-1}(2^n - 1)$ is perfect, but every perfect number is of the form $2^{n-1}(2^n - 1)$. Are there any odd perfect numbers? *We do not know.*

Philo of Alexandria in his first-century book *On the Creation* mentions perfect numbers, claiming that the world was created in 6 days and the moon orbits in 28 days because 6 and 28 are perfect. Philo is followed by *Origen* and by *Didymus the Blind*, who adds the observation that there are only four perfect numbers that are less than 10 000. Saint Augustine defines perfect numbers in *City of God* (Book XI, Chapter 30) in the early 5$^{\text{th}}$ century AD, repeating the claim that God created the world in 6 days because 6 is the smallest perfect number. The Egyptian mathematician Ismail ibn Fallūs (1194–1252) mentioned the next three perfect numbers (33 550 336; 8 589 869 056 and 137 438 691 328) and listed a few more which are now known to be incorrect.

The first known European mention of the fifth perfect number is a manuscript written between 1456 and 1461 by an unknown mathematician. In 1588, the Italian mathematician *Pietro Cataldi* identified the sixth (8 589 869 056) and the seventh (137 438 691 328) perfect numbers, and also proved that every perfect number obtained from Euclid's rule ends with a 6 or an 8.

You should know that a *prime number* is one that is only divisible by itself and 1, or for $n > 2$ (n is the first prime), for $2 \leq x < n, x \nmid n$. Finding primes can be tedious, but in this assignment we will be looking at all proper divisors so using *trial division* is appropriate.

In summary, a *perfect number* n is one where $s(n) = n$, and a *deficient number* is one where $s(n) < n$ and an *abundant number* is one where $s(n) > n$.

> If you look carefully at Figure **??**, you might understand why measurement systems other than *metric* are as they are. Why does the day have 24 hours? The hour 60 minutes? Why are there 360° in a circle? Why does the foot have 12 inches? The yard 36 inches? Humans find fractions easier than repeating decimals.

2 Specifics

You will be writing a **C** program, and the first thing is to include the appropriate header files.

Header Files

```c
#include <stdbool.h> // Gives us the bool data type
#include <stdio.h>   // ... standard I/O
#include <stdlib.h>  // ... strtoul() and other useful things
#include <string.h>  // ... string utilities
#include <unistd.h>  // ... getopt() and other useful things
```

You will need to process the command line arguments. The `-p` option determines whether you will print just *perfect numbers* or all numbers (which is the default).

Command line arguments

```c
#define DEFAULT_MAXIMUM 1000
#define OPTIONS "pn:"

int main(int argc, char **argv) {
    int n = DEFAULT_MAXIMUM;
    bool imperfect = true;
    int opt;

    while ((opt = getopt(argc, argv, OPTIONS)) != -1) {
        switch (opt) {
        case 'p':
            imperfect = !imperfect;
            break;
        case 'n':
            n = strtoul(optarg, NULL, 10);
            break;
        }
    }
    // Your other code goes here ...
}
```

For each positive integer from the ordered set $n \in \{2, \ldots, n\}$ you will classify it as *prime*, *perfect*, *deficient* or *abundant*.

You will print the number, followed by a list of all of its *proper divisors*, followed by its classification. Classifying a number requires you to determine *all* of its proper divisors.

- How do you know if a number n is *prime*? It has only 1 as a *proper divisor*.
- How do you know if a number n is *perfect*? The *sum* of the proper divisors $s(n) = n$.
- How do you know if a number n is *deficient*? The *sum* of the proper divisors $s(n) < n$.
- How do you know if a number n is *abundant*? The *sum* of the proper divisors $s(n) > n$.

You should see a pattern here that you can (and should) exploit.

You will want to keep an array of the proper divisors, alternatively, you can keep a string of their decimal representation, but that seems inelegant, especially if you are trying for *extra credit*. You will need to do this because the `-p` option will prevent printing of all *imperfect* numbers. You will not know which class a number falls into until you have examined all divisors.

Example

```
eco :: ./classify -n 20
2 : [1] is prime
3 : [1] is prime
4 : [1, 2] is deficient
5 : [1] is prime
6 : [1, 2, 3] is perfect
7 : [1] is prime
8 : [1, 2, 4] is deficient
9 : [1, 3] is deficient
10 : [1, 2, 5] is deficient
11 : [1] is prime
12 : [1, 2, 3, 4, 6] is abundant
13 : [1] is prime
14 : [1, 2, 7] is deficient
15 : [1, 3, 5] is deficient
16 : [1, 2, 4, 8] is deficient
17 : [1] is prime
18 : [1, 2, 3, 6, 9] is abundant
19 : [1] is prime
20 : [1, 2, 4, 5, 10] is abundant
```

Example

```
eco :: ./classify -n 10000 -p
6 : [1, 2, 3] is perfect
28 : [1, 2, 4, 7, 14] is perfect
496 : [1, 2, 4, 8, 16, 31, 62, 124, 248] is perfect
8128 : [1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064] is
   perfect
```

2.1 Executables

An executable on UNIX is just another name for a program. The terms *program*, *executable*, *executable binary*, and sometimes just *binary*, all refer to a program that can be run, or *executed*. The distinction between an plain old executable and an executable binary is in its representation. An executable binary is platform specific and is simply a file full of machine code: pure binary. An executable could be a shell script, which contains readable text. The only thing these two share is that they must have the executable bit set in order to be run. Refer to the discussion of UNIX file permissions in assignment 0 if you have forgotten what the executable bit refers to.

3 Your Task

Use the `Makefile` that is provided for you.

Create a program called `classify.c` that takes two arguments:

`-p` that determines whether to print only perfect numbers

`-n #` that determines the largest number to classify

It will classify each number into one of *four* categories: prime, perfect, deficient or abundant.

You will do this using the 'getopt()' library

3.1 *bonus pro opere superior*

Some of you may be thinking that this assignment is too simple, it does not test your abilities. Consequently, we are offering you the opportunity to earn an extra 20% by finding the so-called *weird numbers*.

A *semiperfect number* is a natural number that is equal to the sum of *some or all* of its proper divisors. A semiperfect number that is equal to the sum of all its proper divisors is a perfect number. Most abundant numbers are also semiperfect; abundant numbers that are not semiperfect are called *weird numbers*.

Determining whether a number is semiperfect requires that the sum of *every subset* of its proper divisors be checked. The set of all subsets of a set S is called its *power set*. Let $|S|$ be the number of elements in S, then the power set, written $\mathscr{P}(S)$ and has cardinality 2^k, $k = |S|$.

An algorithm for cycling through all subsets naturally comes from looking at the binary representation of a number. Assume that we have an array $d_0, \ldots, d_{k-1}$ that hold the k proper divisors of our number n. Let $i = 1, \ldots, 2^{k-1}$. For each such k select the divisors that correspond to 1 bits in the binary representation of i.

Let's see how we might accomplish this in Python:

Semiperfection

```
1 def combination(d):
2     for c in range(1 << len(d)):
3         yield [ _ for _ in range(len(d)) if c & (1 << _) ]
4
5 def semiperfect(d, n):
6     for c in combination(d):
7         if sum([ d[_] for _ in c ]) == n:
8             return True
9     return False
```

How to we accomplish a similar operation in **C**? Consider this simple program:

Checking for set bits

```c
#include <stdio.h>

int main(void) {
    int x = 1962;
    for (int i = 0; i < 32; i += 1) {
        if (x & 1 << i) { printf("bit %d is 1\n", i); }
    }
    return 0;
}
```

If you decide to try for extra credit, what should your output look like? Here's a simple example. Be warned, do not expect finding weird numbers to be fast—they are few in number, and you have to evaluate the power set for each number.

Example

```
eco $ ./classify -w -n 100 | grep weird
70: [1, 2, 5, 7, 10, 14, 35], abundant and weird
836 {11}: [1, 2, 4, 11, 19, 22, 38, 44, 76, 209, 418], abundant and
   weird
```

4 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.
—Leopold Kronecker