Project Euler Problems

Michael E. Conlen

July 7, 2013

Preface

Project Euler, http://projecteuler.net/, is a list of programming problems with a mathematical and algorithmic bent. These problems have solutions that vary from the naïve to the sophisticated. While the easiest problems can be effectively solved naïvely the advanced problems require sophisticated solutions to run effectively. Here we compile a set of solutions in various programming languages along with a mathematical treatment of the sophisticated solutions. Where possible the solutions are generalized for various parameters given in the statement of the problem.

While Project Euler requests that the solutions not be shared outside of the forums it's clear the solutions are available on the internet. If you have not solved a problem it is up to you to be honest. It is up to you to realize that understanding the solution is not the point; the point is to have been able to develop the solution yourself.

Contents

Listings

Chapter 1

Sum of Natural Numbers Divisible by 3 and 5

> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.
>
> Find the sum of all the multiples of 3 or 5 below 1000.

1.1 Introduction

The naïve solutions is to iterate k over the range of integers and if $k \equiv 0 \pmod 3$ or $k \equiv 0 \pmod 5$ then add the integer to the sum. This solutions is given in Listing 1.1; however this solution runs in $O(n)$ time. A direct computation can be found.

Listing 1.1: Problem 1: Naïve Solution

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int j,k;
    j=0;
    for(k=0; k<1000; k++) {
        if(k%3 == 0 || k%5 == 0) j+=k;
    }
    printf("%d\n", j);
    return(0);
}
```

1.2 Direct Computation

Let n be the integer we iterate up through, in this case, 999*. Let $m_q = \left\lfloor \frac{n}{q} \right\rfloor$, the number of natural numbers less than n which are multiples of the natural number q; then notice that the sum of natural numbers less than n and divisible by q is

$$\begin{aligned} q + 2q + 3q + \cdots + m_q q &= q \sum_{k=1}^{m_q} k \\ &= q \frac{(m_q)(m_q+1)}{2} \end{aligned} \tag{1.1}$$

If we are summing over the integers which are multiples of q and r then each natural number which is a multiple of both p and r is counted twice;

* The problem asks for numbers up to 1000, thus does not include 1000 where it is a multiple of 5.

thus we subtract multiples of qr; and the solution is

$$q\frac{(m_q)(m_q+1)}{2}+r\frac{(m_r)(m_r+1)}{2}-qr\frac{(m_{qr})(m_{qr}+1)}{2} \tag{1.2}$$

A generalized version of this program is given in Listing 1.2. It's runtime is $O(1)$.

Listing 1.2: Problem 1: C Solution

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    unsigned long long  q=0, r=0, n=0;
    unsigned long long  mq, mr, mqr, sum;
    char                copt;

    while((copt = getopt(argc, argv, "n:q:r:")) != -1) {
        switch(copt) {
            case    'n':
                n = atoll(optarg)-1;
                break;
            case    'q':
                q = atoll(optarg);
                break;
            case    'r':
                r = atoll(optarg);
                break;
            default:
                goto usage;
        }
    }
    if(n == 0 || q == 0 || r == 0) goto usage;

    mq = n/q;
    mr = n/r;
    mqr = n/(q*r);
    sum = q*(mq*(mq+1))/2 + r*(mr*(mr+1))/2 - (q*r)*(mqr*(mqr+1))/2;
    printf("%lld\n", sum);
    exit(0);
usage:
    fprintf(stderr, "%s -n N -q Q -r R\n", argv[0]);
    exit(-1);
}
```

Chapter 2

Sum of Even Fibonacci Numbers

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

2.1 Introduction

The naïve solution is to iterate k over the range of natural numbers computing F_k, the k^{th} Fibonacci number, until $F_k > 4000000$ and if F_k is even add it to the sum. This solution is given in Listing 2.1. While this solution is generally quick on modern computers it is not efficient.

Listing 2.1: Problem 2: Naïve Solution

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  unsigned long long int fib(unsigned long long n)
5  {
6       if(n==0) return(0);
7       if(n==1) return(1);
8       return(fib(n-1) + fib(n-2));
9  }
10
11 int main(int argc, char *argv[])
12 {
13      unsigned long long j,k,Fk;
14
15      j=0; k=0;
16      while(1) {
17           Fk = fib(k++);
18           if(Fk > 4000000) break;
19           if(Fk%2 == 0) j += Fk;
20      }
21      printf("%llu\n", j);
22      return(0);
23 }
```

2.2 Order of F_n

We claim that the computation of `fib(n)` is at least exponential. Let $T(n)$ be the time to compute the n^{th} Fibonacci number and we can see that if we let $T(0) = 1$ in units of the time to make the comparison in Line 6, then $T(1) = 2 > 1$ and $T(2) = 2 + T(1) + T(0) > T(1) + T(0)$. In general

$T(n) > T(n-1) + T(n-2)$. That is, the estimate is directly related to the Fibonacci numbers themselves.

We may use generating functions* to compute F_k. Assume there is a function $f(x) = \sum_{k=0}^{\infty} F_k x^k$. Recall the defining equation of the Fibonacci numbers;

$$F_{k+2} = F_{k+1} + F_k \tag{2.1}$$

with the boundary conditions $F_0 = 0$ and $F_1 = 1$†. Multiply equation 2.1 by x^k,

$$F_{k+2}x^k = F_{k+1}x^k + F_k x^k \tag{2.2}$$

* See [1, 2].

† This is not precisely the same boundary as $T(0) = 1$ and $T(1) = 1$; however we can see that it is simply a shift if we consider the negative extension $T(-1) = 0$ noting that it preserves the recurrence relation. This preserves the standard numbering of the Fibonacci sequence.

then sum both sides of equation 2.2 over all k and compute

$$
\begin{aligned}
& & \sum_{k=0}^{\infty} F_{k+2}x^k &= \sum_{k=0}^{\infty} F_{k+1}x^k + \sum_{k=0}^{\infty} F_k x^k \\
&\implies & \frac{1}{x^2}\sum_{k=0}^{\infty} F_{k+2}x^{k+2} &= \frac{1}{x}\sum_{k=0}^{\infty} F_{k+1}x^{k+1} + f(x) \\
&\implies & \frac{1}{x^2}\left(f(x) - F_1 x - F_0\right) &= \frac{1}{x}\left(f(x) - F_0\right) + f(x) \\
&\implies & f(x) - F_1 x - F_0 &= x\left(f(x) - F_0\right) + x^2 f(x) \\
&\implies & f(x) - xf(x) - x^2 f(x) &= F_1 x + F_0 - F_0 x \\
&\implies & f(x)\left(1 - x - x^2\right) &= x \\
&\implies & f(x) &= \frac{x}{1 - x - x^2} \qquad (2.3)
\end{aligned}
$$

If we define $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$ we can see that

$$
1 - x - x^2 = (1 - x\varphi)(1 - x\psi) \qquad (2.4)
$$

Thus we can simplify equation 2.3 using equation 2.4 and partial fraction decomposition

$$
\begin{aligned}
\frac{x}{1 - x - x^2} &= \frac{x}{(1 - x\varphi)(1 - x\psi)} \\
&= \frac{A}{1 - x\varphi} + \frac{B}{1 + x\psi} \\
&= \frac{(1 - x\psi)A + (1 - x\varphi)B}{1 - x - x^2} \\
&= \frac{(A + B) - x(\psi A + \varphi B)}{1 - x - x^2} \qquad (2.5)
\end{aligned}
$$

and from equation 2.5 we can deduce that $A + B = 0$ and $\psi A + \varphi B = 1$ and compute

$$\begin{aligned}
\psi A + \varphi B &= -1 \\
\implies \psi A + \varphi(-A) &= -1 \\
\implies \qquad (\psi - \varphi)A &= -1 \\
\implies \qquad\qquad A &= \frac{1}{\varphi - \psi}
\end{aligned}$$

thus

$$B = -\frac{1}{\varphi - \psi}$$

We rewrite equation 2.3

$$\begin{aligned}
\frac{1}{1 - x - x^2} &= \frac{1}{\varphi - \psi}\left(\frac{1}{1 - x\varphi} - \frac{1}{1 - x\psi}\right) \\
&= \frac{1}{\sqrt{5}}\left(\sum_{k=0}^{\infty} \varphi^k x^k - \sum_{k=0}^{\infty} \psi^k x^k\right) \\
&= \sum_{k=0}^{\infty} \frac{\varphi^k - \psi^k}{\sqrt{5}} x^k \qquad (2.6)
\end{aligned}$$

thus, recalling that $f(x) = \sum_{k=0}^{\infty} F_k x^k$ we conclude that

$$F_k = \frac{\varphi^k - \psi^k}{\sqrt{5}} \qquad (2.7)$$

Since $|\psi| < 1$ we have that $\psi^k \to 0$ as $k \to \infty$; thus we can see that F_k is exponential in k; hence $T(n)$ is at least $O(\varphi^n)$*.

2.3 Computing `fib(n)` Directly

Equation 2.7 gives us a closed form for F_n which can be computed for the cost of two calls to `pow()`. Recall, however, that $|\psi| < 1$ and, in fact, $\left|\frac{\psi}{\sqrt{5}}\right| < \frac{1}{2}$; thus we can avoid computation of ψ^n and estimate that $F_k = \frac{\varphi^k}{\sqrt{5}} + \epsilon$ and that $|\epsilon| < 1$; thus we can compute

$$F_k = \left\lfloor \frac{\varphi^k}{\sqrt{5}} + \frac{1}{2} \right\rfloor \tag{2.8}$$

This solution is given in Listing 2.2. Note that we make use of the fact that for positive numbers typecasting a `double` to an `int` type is equivalent to the floor function. In this implementation the runtime of `fib(n)` is generally $O(1)$ as `pow()` is implemented in constant time on most processors†. This gives the overall program a runtime of $O(\log n)$; but we can do better.

2.4 Sum of Even Fibonacci Numbers

Notice that the first two Fibonacci numbers are odd, followed by an even. We can see from the defining relation that the even Fibonacci numbers have an index $k \equiv 0 \pmod 3$. Moreover, the sum of the even Fibonacci numbers

* If we work with the estimate that $T_0 = 1$, $T_1 = 2$ and $T_{k+2} = T_{k+1} + T_k + m$ where m is the number of operations for the non-recursive steps in the general case, then a similar analysis will show that the runtime is exponential.

† If this is not available exponentiation by squaring is $O(\log n)$.

Listing 2.2: Problem 2: `fib(n)` Direct

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define PHI 1.6180339887498949895
#define OORF 0.4472135954999579

unsigned long long int fib(unsigned long long n)
{
    double Fn;

    Fn = pow(PHI, n)*OORF + 0.5;
    return((unsigned long long)Fn);
}

int main(int argc, char *argv[])
{
    unsigned long long j,k,Fk;

    j=0; k=0;
    while(1) {
        Fk = fib(k++);
        if(Fk > 4000000) break;
        if(Fk%2 == 0) j += Fk;
    }
    printf("%llu\n", j);
    return(0);
}
```

is equal to the sum of the odd Fibonacci numbers before it. We can make use of this fact and the following observation

Theorem 2.1. *Let F_k be the k^{th} Fibonacci number with $F_1 = 1$ and $F_2 = 1$; then $\sum_{k=1}^{n} F_k = F_{n+2} - 1$.*

Solution. Let $n = 1$ then $\sum_{k=1}^{1} F_k = F_1 = 1 = 2 - 1 = F_3 - 1$; this proves the base case. We compute for arbitrary n

$$\begin{aligned} \sum_{k=1}^{n} F_k &= F_n + \sum_{k=1}^{n-1} F_k \\ &= F_n + F_{n+1} - 1 \\ &= F_{n+2} - 1 \end{aligned}$$

□

Thus, if we know the index of the largest Fibonacci number less than or equal to some value, we can compute the desired sum.

2.5 Finding the Index

Notice that if we have a Fibonacci number F, then we can compute the index into the sequence, k, by

$$k = \log_{\varphi}\left(F\sqrt{5} + \psi^k\right)$$

but without k we must estimate, but $\left|\psi^k\right| < \frac{1}{2}$ for $k > 1$; thus we have that

$$k < \log_\varphi\left(F\sqrt{5} + \frac{1}{2}\right)$$

moreover, the difference is less than 1 for sufficiently large F* so let

$$k = \left\lfloor \log_\varphi\left(F\sqrt{5} + \frac{1}{2}\right)\right\rfloor$$

Now, suppose that F is not a Fibonacci number, but is some natural number; then for some $j \in \mathbb{N}$, $F_j < F < F_{j+1}$ so

$$j \leqslant \left\lfloor \log_\varphi\left(F\sqrt{5} + \frac{1}{2}\right)\right\rfloor \leqslant j + 1 \tag{2.9}$$

thus

$$\left\lfloor \log_\varphi\left(F\sqrt{5} + \frac{1}{2}\right)\right\rfloor \in \{j, j+1\} \tag{2.10}$$

We wish to determine j, the index of the largest Fibonacci number less than or equal to F†; so compute k then F_k and if $k = j + 1$ $F_k > F$ and if $k = j$ then $F_k \leqslant F$. In either case we can determine j.

Finally, to get the largest *even* Fibonacci number recall that a Fibonacci number F_k is even if and only if $k \equiv 0 \pmod 3$.

---

* Consider $\frac{d}{dx}\log(x) = \frac{1}{x}$.

† Recall that the problem states ... *terms in the Fibonacci sequence whose values do not exceed four million...*

2.6 Direct Computation

We combine the results of the last two sections in Listing 2.3.

Listing 2.3: Problem 2: Direct Solution

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#define PHI 1.618033988749895
#define OORF 0.4472135954999579
#define RF 2.23606797749979
#define LPHI 0.48121182505960347

unsigned long long int fib(unsigned long long  n)
{
    double  Fn;

    Fn = pow(PHI, n)*OORF + 0.5;
    return((unsigned long long)Fn);
}

int main(int argc, char *argv[])
{
    unsigned long long  j,k,Fk, Fk2;

    j=0; k=0;
    k = (unsigned long long)(log(4000000*RF+0.5)/LPHI);
    Fk = fib(k);
    if(Fk > 4000000) k--;
    Fk2=fib(k+2);
    j = (Fk2 - 1)/2;
    printf("%llu\n", j);
    return(0);
}
```

2.7 Generalization

We generalize the above solution to arbitrary n within the limits of `unsigned long long` in Listing 2.4.

Listing 2.4: Problem 2: General Solution

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define PHI 1.618033988749895
#define OORF 0.4472135954999579
#define RF 2.23606797749979
#define LPHI 0.48121182505960347

unsigned long long int fib(unsigned long long n)
{
    double  Fn;

    Fn = pow(PHI, n)*OORF + 0.5;
    return((unsigned long long)Fn);
}

int main(int argc, char *argv[])
{
    unsigned long long  n=0, j,k,Fk, Fk2;
    char                copt;

    while((copt = getopt(argc, argv, "n:")) != -1) {
        switch(copt) {
            case    'n':
                n = atoll(optarg);
                break;
            default:
                goto usage;
        }
    }
    if(n==0) goto usage;
    j=0; k=0;
    k = (unsigned long long)(log(n*RF+0.5)/LPHI);
    Fk = fib(k);
    if(Fk > n) k--;
    k -= k%3;
    Fk2=fib(k+2);
    j = (Fk2 - 1)/2;
    printf("%llu\n", j);
    exit(0);
usage:
    fprintf(stderr, "%s -n N\n", argv[0]);
    exit(-1);
}
```

Chapter 3

Largest Prime Factor of n

> The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the mots important and useful in arithmetic
>
> C. F. Gauss

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143 ?

3.1 Introduction

This algorithm works simply by factoring the integer in to prime factors then searching for the largest of the list. We implement this algorithm in Listing 3.1.

Listing 3.1: Problem 3: Naïve Solution

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "factor.h"

int main(int argc, char *argv[])
{
    unsigned long long  n = 600851475143, i, d=0;
    unsigned long long  len, *list;

    factorN(n, &list, &len);
    for(i=0; i<len; i++) {
        if(d<list[i]) d = list[i];
    }
    printf("%llu\n", d);
    return(0);
}
```

3.2 Prime Factorization

Factoring the integer into primes is the key component of this algorithm. A simple algorithm is devised which should work well for all but exceedingly large numbers. The header, `factor.h`, is given in listing 3.2. The code in `factor.c` is given in listing 3.3. This will be used in several problems.

3.3 `factorN()`

The function `factorN()` works by finding small factors, factoring them out and continuing to look for successively larger factors.

More precisely, let $n \in \mathbb{N}$ be given. Let $n_1 = n$. Let $m_2 \in \mathbb{N}$ be the largest number such that 2^{m_2} divides n_1; then set $n_2 = \frac{n_1}{2^{m_2}}$. Inductively continue until we have a number n_k such that $k > \sqrt{n_k}$. Since $\{n_i\}$ is a non-decreasing sequence this number k exists. n_k will then be the largest prime factor of n.

Clearly $n_k \mid n$. We can see that n_k is not composite, since if it was the factors of n_k would be smaller than n_k and thus would have been divided out of n_k at the appropriate step in the construction. We are left to show that there is no prime p such that $p > n_k$ and $p \mid n$. This can be seen by looking at the sequence $\{n_k\}$. At each step where n_k is a composite then at least one of it's prime factors is less than $\sqrt{n_k}$; thus, if there are no such prime factors then n_k is prime.

Listing 3.2: `factor.h`

```
#ifndef FACTOR_H
#define FACTOR_H

#include <config.h>
#include <stdint.h>

#if HAVE_CUNIT_CUNIT_H
int init_factor(void);
int clean_factor(void);
void      unit_factorN(void);
void      unit_twoFactor(void);
#endif

int factorN(unsigned long long n, unsigned long long **list ,
    unsigned long long *len);
int twoFactor(uint64_t *list , uint64_t len, uint64_t *x,
    uint64_t *y, uint64_t n);
#endif
```

Listing 3.3: factor.c:factorN()

```
int factorN(unsigned long long n, unsigned long long **list,
    unsigned long long *len)
{
     unsigned long long  sn, i, *f;
     queue               *q;

     if((q = queueCreate(NULL)) == NULL) goto error0;
     sn = sqrt(n);
     while((n & 1) == 0) {
          f = malloc(sizeof(unsigned long long));
          *f = 2;
          queueEnqueue(q, f);
          n = n >> 1;
     }
     sn = sqrt(n);
     for(i=3; i<=sn; i+=2) {
          while(n%i == 0) {
               f = malloc(sizeof(unsigned long long));
               *f = i;
               queueEnqueue(q, f);
               n = n/i;
               sn = sqrt(n);
          }
     }
     if(n != 1) {
          f = malloc(sizeof(unsigned long long));
          *f = n;
          queueEnqueue(q, f);
     }
     *len = queueLength(q);
     *list = malloc(sizeof(unsigned long long) * (*len));
     for(i=0; i< *len; i++) {
          f = queueDequeue(q);
          (*list)[i] = *f;
          free(f);
     }
     queueDestroy(q);
     return(0);
error0:
     return(-1);
}
```

Chapter 4

Largest Palindrome Product

> A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.
>
> Find the largest palindrome made from the product of two 3-digit numbers.

4.1 Introduction

This problem presents a few programming issues, the first is to factor an integer and find products of the factors of a given decimal size. The other problem is to be able to represent the sequence of palindrome numbers in a way that we can easily find the predecessor of an element in the sequence.

4.2 Palindromic Numbers

4.2.1 Introduction

Palindromic numbers are natural numbers which are the same when written (in a given base) forwards and reverse. These numbers can be thought of as having two varieties; those with an odd number of digits and those with an even number of digits. We can represent a palindromic number by an integer, representing the leading sequence of the integer and a value to indicate whether the number of digits is odd or even; that is, whether the last digit in the integer should be included once or twice respectively. By way of example the palindromic number 10101 can be represented by $(101, \texttt{ODD})$ and the number 457754 can be represented by $(457, \texttt{EVEN})$.

4.2.2 Predecessor and Successor

To see how to compute predecessor or successor palindromic numbers we consider what the sequence looks like in these terms. The first nine terms are the natural numbers 0 through 9. These are represented by the values $(0, \texttt{ODD})$ through $(9, \texttt{ODD})$. These numbers are followed by $11, 22, 33, \ldots, 99$ which are represented by $(1, \texttt{EVEN})$ through $(9, \texttt{EVEN})$. The next portion of the sequence is

$$101, 111, 121, 131, \ldots, 191, 202, 212, 222, \ldots, 292, 303, \ldots, 999$$

These are represented by $(10, \texttt{ODD})$ through $(99, \texttt{ODD})$. It can be seen that the following portion of the sequence is represented by the values $(10, \texttt{EVEN})$ through $(99, \texttt{EVEN})$.

We can then consider how to compute the successor of a given palindromic number (a, b), written $(a, b) + +$. If $a + 1 \neq 10^k$ for $k \in \mathbb{N}$, $k > 0$ then $(a + b) + + = (a + 1, b)$. That is, $(7, \texttt{ODD}) + + = (8, \texttt{ODD})$ for example. If $a + 1 = 10^k$ for $k \in \mathbb{N}$ and $k > 0$ then we have two situations depending on b. The successor of $(9, \texttt{ODD})$ is $(1, \texttt{EVEN})$, likewise $(99, \texttt{ODD}) + + = (10, \texttt{EVEN})$ and so forth; thus we see that in this case

$$(a, b) + + = \left(\frac{a+1}{10}, \texttt{EVEN} \right)$$

In the case where $b = \texttt{EVEN}$ we have $(a, b) + + = (a + 1, \texttt{ODD})$.

We can write the successor function as

$$(a, b) + + = \begin{cases} (a + 1, b), & a + 1 \neq 10^k,\ k \in \mathbb{N},\ k > 0 \\ \left(\frac{a+1}{10}, \texttt{EVEN} \right), & a + 1 = 10^k,\ k \in \mathbb{N},\ k > 0 \wedge b = \texttt{ODD} \\ (a + 1, \texttt{ODD}), & a + 1 = 10^k,\ k \in \mathbb{N}, k > 0 \wedge b = \texttt{EVEN} \end{cases} \tag{4.1}$$

This function is implemented in `palindromeSuccessor()` given in Listing 4.1.

From the successor we can compute the predecessor function $(a, b) - -$, Listing 4.1: Problem 4: `palindromeSuccessor()`

```
int palindromeSuccessor(palindromeN *n)
{
    double      k;

    k = log10(n->a + 1);
    if(k - floor(k) < 1e-15 && (int)k >= 1) {    // k in N
        if(n->b == PALINDROME_ODD) {
            n->a = (n->a + 1)/10;
            n->b = PALINDROME_EVEN;
        } else {
            n->a++;
            n->b = PALINDROME_ODD;
        }
    } else { // k not in N
        n->a++;
    }
    return(0);
error0:
    return(-1);
}
```

remembering to take care of a few extra special cases.

$$(a,b) - - = \begin{cases} \texttt{UNDEFINED}, & a = 0 \wedge b = \texttt{ODD} \\ (a-1, \texttt{ODD}), & a = 1 \wedge b = \texttt{ODD} \\ (a-1, b), & a \neq 10^k, k \in \mathbb{N}, k > 0 \\ (a-1, \texttt{EVEN}), & a = 10^k, k \in \mathbb{N}, k > 0 \wedge b = \texttt{ODD} \\ ((a-1) \cdot 10 + 9, \texttt{ODD}), & a = 10^k, k \in \mathbb{N} \wedge b = \texttt{EVEN} \end{cases} \tag{4.2}$$

4.2.3 Computing Integer from Representation

Given the representation of a palindromic number used above we need to compute the actual integer. The value a represents the leading digits which must be shifted some places to the left. The number of places depends on the value of b. If $b = \texttt{ODD}$ then the shift is $\lfloor \log_{10}(a) \rfloor$. If $b = \texttt{EVEN}$ then the shift is $\lfloor \log_{10}(a) \rfloor + 1$.

We might attempt to figure out the value of each place in decimal; but this has been done already by `sprintf()`. We simply determine the length of a, $\texttt{digits}(a) = \lfloor \log_{10}(a) \rfloor + 1$ to determine the length of the string necessary, $\texttt{length}(a) + 1$, then use `sprintf()` to place the integer into the string, then treating each character as an integer subtract the value of the string "0" from each value.

The function `palindromeInteger()` in Listing 4.3 implements this.

Listing 4.2: Problem 4: `palindromePredecessor()`

```
int palindromePredecessor(palindromeN *n)
{
    double  k;

    if(n->a == 0 && n->b == PALINDROME_ODD) {
        n->b = PALINDROME_UNDEF;
        return(0);
    }
    if(n->a == 1 && n->b == PALINDROME_ODD) {
        n->a--;
        return(0);
    }
    k = log10(n->a);
    if(k - floor(k) < 1e-15) {
        if(n->b == PALINDROME_ODD) {
            n->a--;
            n->b = PALINDROME_EVEN;
        } else {
            n->a = (n->a - 1)*10 + 9;
            n->b = PALINDROME_ODD;
        }
    } else {
        n->a--;
    }

    return(-1);
}
```

Listing 4.3: Problem 4: `palindromeInteger()`

```
uint64_t palindromeInteger(palindromeN n) {
    uint64_t    d, shift, len, i;
    char        *str;

    len = floor(log10(n.a)) + 1;
    shift = n.b == 1 ? len : len - 1;
    d = n.a * pow(10, shift);
    if((str = malloc(sizeof(char)*(len + 1))) == NULL) goto
        error0;
    sprintf(str, "%llu\n", n.a);
    for(i=0; i< shift; i++) {
        d += ((int)str[i] - '0')*pow(10, i);
    }
    free(str);
    return(d);
error0:
    return(0);
}
```

4.2.4 Finding an Answer

Given that we can find the largest palindromic integer less than a given number* we must determine if it satisfies the condition that it is the product of two integers of a given length. To do this we may start by factoring the integer into a list of m prime factors. Then we can determine if any grouping of the prime factors into to integers will have products of the required length. There are $\sum_{k=0}^{m} \binom{m}{k} = 2^m$ ways to group m prime factors into two groups where we choose k of them for one product and the remaining $m-k$ are in the other product. By enumerating over these 2^m groupings we can determine

* Using the free digits at the start of the integer find the palindromic integer associated with it. Either it satisfies the inequality, at which point any palindromic integer larger than it will not satisfy the inequality, or it's predecessor will satisfy the inequality.

if any grouping satisfies by computation.

Listing 4.4: Problem 4: `twoFactor()`

```
int twoFactor(uint64_t *list, uint64_t len, uint64_t *x,
    uint64_t *y, uint64_t n)
{
    uint64_t     i;

    if(n >= (uint64_t)1 << len) goto error0; // n is out of
        bounds
    *x = 1; *y = 1;
    for(i=0; i<len; i++) if((n & ((uint64_t)1 << i)) == 0) *y
         *= list[i]; else *x *= list[i];
    return(0);

error0:
    return(-1);
}
```

This algorithm appears inefficient, possibly even exponential, at first glance; however it should not be too bad. Let n be given. We may write $n = \prod_p p^{k_p}$ where p ranges over all prime numbers and $k_p = 0$ if $p \nmid n$. The number of prime factors $m(n)$ is given by $m(n) = \sum k_p$. We can understand the growth of m by inverting it. The smallest integer which has x prime factors is 2^x since any integer which has as many prime factors must have some factor(s) which are not 2 and all prime numbers not equal to 2 are greater than 2. $m\left(2^x\right) = x$; thus the growth of m is $\log_2(n)$; thus an upper bound on the number of groupings we must check for each n is $2^{\log_2(n)} = n$. Summing over all n we have that the algorithm is $O\left(n^2\right)$.

The function `twoFactor()` takes a list of (presumably prime) factors and an integer n such that 2^n is less than the length of the list and groups the prime factors. The code is given in Listing 4.4.

In short, the algorithm is as follows

1. Find largest palindromic number less than 999×999

2. For each palindromic number less than 999×999 down to 100×100 do:

   (a) Factor the palindromic integer

   (b) For each grouping of the factors determine if the grouping results in a pair of 3-digit numbers; if so terminate

The code for this is given in Listing 4.5

Listing 4.5: Problem 4: `main()`

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "palindrome.h"
#include "factor.h"

int main(int argc, char *argv[])
{
    char                    copt, *str;
    uint64_t                strlen = 0, halfLen = 0, len = 0, n =
        0, x = 0, y = 0, i=0;
    uint64_t                min, curN, *factorList, factorLen;
    palindromeN             p;

    while((copt = getopt(argc, argv, "n:")) != -1) {
        switch(copt) {
            case    'n':
                len = atoll(optarg);
                break;
            default:
                goto usage;
        }
    }
    if(len==0) goto usage;
```

```

    for(i=0; i<len; i++) {
        x += 9*pow(10, i);
    }
    n = x*x;
    strlen = (uint64_t)floor(log10(n)) + 1;
    p.b = ((strlen%2) == 0) ? PALINDROME_EVEN : PALINDROME_ODD;
    if((str = malloc(sizeof(char)*(strlen + 1))) == NULL) goto error0;
    sprintf(str, "%llu\n", n);
    halfLen = (strlen/2.0);
    if(p.b == PALINDROME_ODD) halfLen++;
    p.a = 0;
    for(i=0; i< halfLen; i++) {
        p.a += (str[i] - '0')*pow(10, halfLen - 1 - i);
    }
    if(n < palindromeInteger(p)) palindromePredecessor(&p);
    x = pow(10, len - 1);
    min = x*x;
    while((curN = palindromeInteger(p)) >= min) {
        factorN(curN, &factorList, &factorLen);
        for(i=0; i< ((uint64_t)1 << factorLen); i++) {
            twoFactor(factorList, factorLen, &x, &y, i);
                if(len == floor(log10(x)) + 1 && len == floor(log10(y)) + 1) {
                    printf("Integer %llu factors into %llu and %llu\n", curN, x, y);
                    goto done;
                }
        }
        palindromePredecessor(&p);
    }
    printf("found no solutions\n");
done:
    exit(0);
error0:
    exit(-1);

usage:
    fprintf(stderr, "%s -n N\n", argv[0]);
    exit(-1);
}
```

Chapter 5

Smallest Multiple

> 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.
>
> What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

5.1 Introduction

The solution to this problem is the least common multiple of all integers from 1 to 20.

Definition 5.1 (Least Common Multiple)**.** The *least common multiple* of two integers a and b is the smallest $n \in \mathbb{N}$ such that $a \mid n$ and $b \mid n$. We write $\operatorname{lcm}(a, b) = n$.

We can define the lcm on more than two integers inductively; that is

$\operatorname{lcm}(a_0, a_1, \ldots, a_n) = \operatorname{lcm}(\operatorname{lcm}(a_0, a_1, \ldots, a_{n-1}), a_n)$. We can compute the lcm of two integers a and b using the gcd.

Definition 5.2 (Greatest Common Divisor)**.** For two integers a and b with both not equal to zero the *greatest common divisor* is the $n \in \mathbb{N}$ such that $n \mid a$ and $n \mid b$ and any other number $k \in \mathbb{N}$ which divides both also divides n*. We write $\gcd(a, b) = n$.

The connection between the lcm and the gcd is

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)} \tag{5.1}$$

To prove this we need some lemmas†

Lemma 5.3 (Euclid's Lemma)**.** *Let $a, b \in \mathbb{Z} \setminus \{0\}$ and $c \in \mathbb{Z}$ such that $c \mid ab$ with $\gcd(b, c) = 1$ then $c \mid a$.*

Proof. Notice that $\gcd(ab, ac) = |a| \gcd(b, c) = |a|$. By hypothesis $c \mid ab$ and clearly $c \mid ac$ so $c \mid \gcd(ab, ac) = |a|$ that is, $c \mid a$. $\square$

Lemma 5.4. *Let $a, b \in \mathbb{N} \setminus \{0\}$, let $\ell = \operatorname{lcm}(a, b)$ and $g = \gcd(a, b) = 1$; then $\ell = ab$.*

Proof. Notice that $b \mid \ell$; thus $b = \ell n$ for some $n \in \mathbb{Z}$. First we show that $a \mid n$ by Lemma 5.3. By hypothesis $\gcd(a, b) = 1$ and by definition $a \mid \ell = bn$ but $a \nmid b$ so $a \mid n$.

* This definition could be written more succinctly as the greatest integer which divides both however the definition can be extended to commutative rings where *greatest* fails to have meaning.

† See [3]

Since $\ell \mid ab$ it follow that $\ell \leqslant ab$; thus $ab \leqslant \ell = ab\left(\frac{n}{a}\right) = bn \geqslant ba$, where we use the fact that $a \mid n \implies n \geqslant a$. Since $ab \leqslant \ell \leqslant ab$ it follows that $\ell = ab$. $\square$

Lemma 5.5. *Let $a, b \in \mathbb{N} \setminus \{0\}$ and let $g = \gcd(a, b)$; then $\gcd\left(\frac{a}{g}, \frac{b}{g}\right) = 1$.*

Proof. Let $c \in \mathbb{N}$ such that $c \mid \frac{a}{g}$ and $c \mid \frac{b}{g}$; then $gc \mid a$ and $gc \mid b$. By maximality of g it follows that $c = 1$. $\square$

Theorem 5.6. *Let $a,\ b \in \mathbb{Z} \setminus \{0\}$. Let $\ell = \operatorname{lcm}(a, b)$ and $g = \gcd(a, b)$ then $\ell g = ab$*

Proof. By Lemma 5.5, $\gcd\left(\frac{a}{g}, \frac{b}{g}\right) = 1$. By Lemma 5.4 then $\operatorname{lcm}\left(\frac{a}{g}, \frac{b}{g}\right) = \frac{ab}{g^2}$; so $ab = g^2 \cdot \operatorname{lcm}\left(\frac{a}{b}, \frac{b}{g}\right) = g \cdot \operatorname{lcm}(a, b) = g\ell$ as required. $\square$

Thus Equation 5.1 is confirmed. This reduces the problem to one of finding the gcd of two numbers.

5.2 Computing the Greatest Common Divisor

Euclidean Division states that for $a, b \in \mathbb{Z}$ and $b \neq 0$ there exists unique integers $q,\ r$ such that $a = bq + r$ and $0 \leqslant r < |b|$. We can use this in an

Listing 5.1: Problem 5: `gcd()`

```
uint64_t gcd(uint64_t a, uint64_t b)
{
    uint64_t    t;
    while(b != 0) {
        t = b;
        b = a % t;
        a = t;
    }
    return(a);
}
```

algorithm to find $\gcd(a, b)$. Let a and b be given; then write

$$a = bq_0 + r_0$$

$$b = r_0 q_1 + r_1$$

$$r_0 = r_1 q_2 + r_2$$

$$\dots$$

$$r_{n-2} = r_{n-1} q_n + r_n$$

By Euclidean Division we can see that the sequence $\{r_k\}$ is decreasing and this procedure terminates when $r_n = 0$. We can see then from the equations that r_{n-1} divides a and b by noting that since $r_n = 0$ then $r_{n-1} \mid r_{n-1} q_n = r_{n-2}$ and work up the ladder. Let c be any number that divides a and b; then by the first equation $c \mid r_0$, and so forth we see that $c \mid r_k$ for all $k < n$; in particular $c \mid r_{n-1}$; so $c \leqslant r_{n-1}$ and $r_{n-1} = \gcd(a, b)$. This algorithm is implemented in Listing 5.1.

Listing 5.2: Problem 5: `lcm()`

```
uint64_t lcm(uint64_t a, uint64_t b)
{
    return((a*b)/gcd(a, b));
}
```

5.3 Solution

The solution depends on the `gcd()` function in Listing 5.1. Additionally we implement the lcm in Listing 5.2. We iterate over the integers as shown in `main()` in Listing

»»»> p5

Listing 5.3: Problem 5: `main()`

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include "algebra.h"

int main(int argc, char *argv[])
{
    uint64_t    n=0, l=0, i;
    char        copt;

    while((copt = getopt(argc, argv, "n:")) != -1) {
        switch(copt) {
            case    'n':
                n = atoll(optarg);
                break;
            default:
                goto usage;
        }
    }
    if(n<3) goto usage;
    l = lcm(2, 3);
    for(i=4; i<=n; i++) {
        l = lcm(l, i);
    }
    printf("lcm = %llu\n", l);
    exit(0);
usage:
    fprintf(stderr, "%s -n N\n", argv[0]);
    exit(-1);
}
```

Chapter 6

Sum Square Difference

The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \cdots + 10^2 = 385$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \cdots + 10)^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

6.1 Introduction

There is an obvious naïve solution which is $O(n)$; however this is unnecessary. We may rewrite the general problem as

$$\left(\sum_{k=0}^{n}\right)^2 - \sum_{k=0}^{n} k^2 \tag{6.1}$$

Both sums in Equation 6.1 have closed form solutions. If we happen to have these solutions we may check them by induction; however, we may also derive them using exponential generating functions.

6.2 Exponential Generating Functions

Recall the generating functions introduced in Section 2.2. Wilf introduces another form of generating functions in [1] called the exponential generating functions.

Definition 6.1 (Exponential Generating Function)**.** Let $\{a_n\}_{n=0}^{\infty}$ be a sequence; then the *exponential generating function* of $\{a_n\}$ is

$$A(x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!} \tag{6.2}$$

The exponential generating function is derived from the sequence $\{1, 1, 1, \dots\}$; that is, $a_n = 1$ for all n. We can see that in this case $A(x) = e^x$. This leads us to the following theorem

Theorem 6.2. *Let $A(x)$ be the exponential generating function for some sequence $\{a_n\}_{n=0}^{\infty}$ then the exponential power series for $\{a_{n+1}\}_{n=0}^{\infty}$ is $A'(x)$.*

Proof. Let $\{a_n\}_{n=0}^{\infty}$ be a sequence and define $A(x) = \sum_{k=0}^{\infty} a_k \frac{x^k}{k!}$. We compute

$$\begin{aligned} A'(x) &= \frac{d}{dx} \sum_{k=0}^{\infty} a_k \frac{x^k}{k!} \\ &= \frac{d}{dx} \left[a_0 + \sum_{k=1}^{\infty} a_k \frac{x^k}{k!} \right] \\ &= \sum_{k=1}^{\infty} a_k k \frac{x^{k-1}}{k!} \\ &= \sum_{k=1}^{\infty} a_k \frac{x^{k-1}}{(k-1)!} \\ &= \sum_{k=0}^{\infty} a_{k+1} \frac{x^k}{k!} \end{aligned}$$

□

With this we can describe the recurrence relationships we work with in terms of differential equations. Recall that for the Fibonacci numbers we had $F_{n+2} = F_{n+1} + F_n$; we can see easily from this that if $f(x) = \sum_{k=0}^{\infty} F_k \frac{x^k}{k!}$ that f satisfies $f'' = f' + f$. Applying the initial conditions that $f(0) = 0$ and $f'(0) = 1$ we will be rewarded as desired.

6.3 Sum of Integers

We wish to compute $\sum_{k=0}^{\infty} k$; thus we write $s_n = \sum_{k=0}^{n} k$ and note that $s_0 = 0$ and $s_{n+1} = \sum_{k=0}^{n+1} k = (n+1) + \sum_{k=0}^{n} k = (n+1) + s_n$. Defining

$S(x) = \sum_{n=0}^{\infty} s_n \frac{x^n}{n!}$ and using Theorem 6.2 we have

$$S'(x) = S(x) + \sum_{n=0}^{\infty} (n+1)\frac{x^n}{n!} \tag{6.3}$$

We rewrite the last term in Equation 6.3 as

$$\sum_{n=0}^{\infty} n\frac{x^n}{n!} + \sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x + \sum_{n=0}^{\infty} n\frac{x^n}{n!} \tag{6.4}$$

and we can compute the last term in the right hand side of Equation 6.4 as follows*

$$\begin{aligned}
\sum_{n=0}^{\infty} n\frac{x^n}{n!} &= \sum_{n=1}^{\infty} n\frac{x^n}{n!} \\
&= x\sum_{n=1}^{\infty} n\frac{x^{n-1}}{n!} \\
&= x\sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!} \\
&= x\sum_{n=0}^{\infty} \frac{x^n}{n!} \\
&= xe^x
\end{aligned} \tag{6.5}$$

Thus replacing the last term in Equation 6.3 and arranging we have

$$S'(x) - S(x) = (x+1)e^x \tag{6.6}$$

* We could also use the $x\frac{d}{dx}$ operator here and have $\frac{d}{dx}e^x - 1 = e^x$.

Using the initial condition that $S'(0) = 1$ we solve the differential equation*. Using D^{-1} as the inverse of the differential operator $D = \frac{d}{dx}$ we have

$$
\begin{aligned}
S(x) &= \frac{1}{2}\left(x^2 + 2x\right) e^x \\
&= \frac{1}{2}\left(x^2 + 2x\right) \sum_{n=0}^{\infty} \frac{x^n}{n!} \\
&= \frac{1}{2}\left[\sum_{n=0}^{\infty} \frac{x^{n+2}}{n!} + 2\sum_{n=0}^{\infty} \frac{x^{n+1}}{n!}\right] \\
&= \frac{1}{2}\left[D^{-2} \sum_{n=0}^{\infty} \frac{d^2}{dx^2}\frac{x^{n+2}}{n!} + 2D^{-1} \sum_{n=0}^{\infty} \frac{d}{dx}\frac{x^{n+1}}{n!}\right] \\
&= \frac{1}{2}\left[D^{-2} \sum_{n=0}^{\infty} (n+2)(n+1)\frac{x^n}{n!} + 2D^{-1} \sum_{n=0}^{\infty} (n+1)\frac{x^n}{n!}\right]
\end{aligned}
$$

Now we apply Theorem 6.2 in reverse; we can shift the coefficients of the exponential power series to resolve the inverse differential operators and we have

$$
\begin{aligned}
S(x) &= \frac{1}{2}\left[\sum_{n=0}^{\infty} n(n-1)\frac{x^n}{n!} + \sum_{n=0}^{\infty} 2n\frac{x^n}{n!}\right] \\
&= \sum_{n=0}^{\infty} \frac{n(n-1) + 2n}{2}\frac{x^n}{n!} \\
&= \sum_{n=0}^{\infty} \frac{n(n+1)}{2}\frac{x^n}{n!}
\end{aligned}
$$

thus $s_n = \frac{n(n+1)}{2}$ as we expected†.

* Everyone always wonders when they will use Calculus if they just want to program.

† If we track the constant of integration it becomes part of the $n = 0$ term and we can resolve it using the fact that $s_0 = 0$.

6.4 Sum of Integers Squared

To compute $\sum_{k=0}^{n} k^2$ we proceed as we did in the previous section. The differential equation becomes

$$S' - S = \sum_{n=0}^{\infty} (n+1)^2 \frac{x^n}{n!} \tag{6.7}$$

and the last term of Equation 6.7 becomes $\left(x^2 + 3x + 1\right) e^x$. The solution to the differential equation is

$$S(x) = \frac{1}{6}\left(2x^3 + 9x^2 + 6x\right) e^x$$

and the coefficients of the exponential power series are computed to

$$\frac{n(n+1)(2n+1)}{6} \tag{6.8}$$

as expected.

6.5 Solution

Recalling that we were intending to write software we now have all the information for the solution. The difference between the sum of the squares of the first n natural numbers and the square of the sum of the first n natural numbers is

$$\left(\frac{n(n+1)}{2}\right)^2 - \frac{n(n+1)(2n+1)}{6} = \frac{3n^4 + 2n^3 - 3n^2 - 2n}{12} \tag{6.9}$$

The solution, rather anticlimactically, is given in Listing 6.1.

Listing 6.1: Problem 6: `SSD()`

```
unsigned long long int SSD(unsigned long long  n)
{
    return((3*n*n*n*n+2*n*n*n-3*n*n-2*n)/(12));
}
```

Chapter 7

The n^{th} Prime Number

By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13.

What is the 10 001st prime number?

7.1 Introduction

The distribution of primes is known to be irregular. Yitang Zhang has shown that there are infinitely many pairs of consecutive primes such that the difference between them is less than 7×10^7 [4]; that is, if p_k is the k^{th} prime number that

$$\liminf_{n\to\infty}(p_{n+1} - p_n) < 7 \times 10^7 \tag{7.1}$$

Guass conjectured and Hadamard and Poussin later proved a result concerning the prime counting function $\pi(x)$ where $\pi(x)$ is the number of primes less than x. The theorem is known as the Prime Number Theorem [5].

Theorem 7.1 (Prime Number Theorem)**.**

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\log(x)}} = 1 \tag{7.2}$$

If follows from this that

$$\limsup_{x \to \infty} (p_{n+1} - p_n) = \infty \tag{7.3}$$

since $\frac{\pi(x)}{x} \to 0$ as $x \to \infty$. These results show that finding the n^{th} prime number is difficult. The naïve solution would be to check each natural number k in succession testing it for primality by dividing it by every prime found less than $\sqrt{k}$. For small natural numbers, say $n = 10001$ this isn't too bad; however for very large natural numbers the divisions becomes expensive*.

7.2 Sieve of Eratosthenes

Suppose we wish to determine all the prime numbers less than a given natural number, say x. We can list all the natural numbers unto x. We know that 2 is the first prime number so we can remove all multiples of 2 from the list, that is, $\{4, 6, 8, \dots\}$. The next number remaining is 3, which we determine to be prime and we remove all multiples of three from the list, $\{6, 9, 12, \dots\}$. We

* Multiplication is practically $O(n \log(n) \log(\log(n))$ while addition is $O(\log(n))$.

then find 5 and continue as before. If we have the memory this method works well and only involves addition rather than the more expensive multiplication. The time complexity of the algorithm is $O(n \log(\log(n)))$ at register size and $O(n \log(n) \log(\log(n)))$ in bit complexity for large numbers*. The Sieve of Erathothenes is implemented in Listing 7.1

Listing 7.1: Problem 7: `sieveE()`

```
int sieveE(uint64_t x, uint64_t **list, uint64_t *len)
{
    uint64_t    i, j, count=0;
    uint64_t    *s;

    if(x < 2) goto error0;
    if((s = calloc(x+1, sizeof(uint64_t))) == NULL) goto error0;
    s[0] = 1; s[1] = 1;
    for(i=2; i<=x; i++) {
        for(j=2; j*i<=x; j++) {
            s[i*j] = 1;
        }
    }
    *len = 0;
    for(i=2; i<=x; i++) if(s[i] == 0) (*len)++;
    if((*list = malloc(sizeof(uint64_t)* *len)) == NULL) goto error1;
    for(i=2; i<=x; i++) if(s[i] == 0) (*list)[count++] = i;

    free(s);
    return(0);
error1:
    free(s);
error0:
    return(-1);
}
```

* Compare to the bit complexity of multiplication.

7.3 Sizing the Sieve

With the Sieve of Eratosthenes we need an estimate of where the n^{th} prime number may be found. While the Prime Number Theorem estimates the density we must be wary to undershooting x*. Before the Prime Number Theorem was proven Chebychev proved a weaker result which provides upper and lower bounds on $\pi(x)$.

Theorem 7.2 (Chebychev's Theorem)**.** *For* $x \geqslant 8$

$$\frac{\log(2)}{4} \cdot \frac{x}{\log(x)} \leqslant \pi(x) \leqslant 30(\log(2))\frac{x}{\log(x)} \tag{7.4}$$

However, for $n = 10001$ these bounds tell us $x \in (3987, 783242)$. More recently Pierre Duscart proves stricter bounds[6]. The bounds are

$$\left(1 + \frac{1}{\log(x)}\right)\frac{x}{\log(x)} \leqslant \pi(x) \leqslant \left(1 + \frac{1.2762}{\log(x)}\right)\frac{x}{\log(x)} \tag{7.5}$$

where the first bound is valid for $x \geqslant 599$†. These bounds give an estimate of $x \in (104044, 106571)$. This provides a nice upper bound to work with.

To generalize this let the number of primes be given, say n; then we must solve (simplifying the lower bound in Equation 7.5)

$$\frac{x(1 + \log(x))}{\log(x)^2} - n = 0 \tag{7.6}$$

* We can imagine techniques which would allow us to extend and continue if we did undershoot these turn out to be unnecessary.

† Duscart provides more precise bounds for larger x.

however this does not succumb to inversion easily. We can however solve this numerically using Newton's method. We compute the derivative

$$\frac{\log{(x)}^2 - 2}{\log{(x)}^3} \tag{7.7}$$

which gives us the recurrence relationship

$$x_{n+1} = x_n + \frac{\log{(x_n)}\left(-x_n - x_n \log{(x_n)} + n \log{(x_n)}^2\right)}{-2 + \log{(x_n)}^2} \tag{7.8}$$

For an initial x_n we may make use of the Prime Number Theorem and set

$$x_0 = n \log{(n)} \tag{7.9}$$

We implement this function in Listing 7.2. Convergence is extremely fast.

Listing 7.2: Problem 7: `findBound()`

```
uint64_t findBound(uint64_t n)
{
    long double x, y, lx;

    if(n<599) n = 599;
    x = n*logl(n);
    while(1) {
        lx = logl(x);
        y = x + ( lx * (-x -x * lx + n * lx * lx)) / ( -2 +
            lx * lx);
        if(fabsl(y-x)< 1e-1) return((uint64_t)round(y));
        x = y;
    }
}
```

Chapter 8

Largest Product

Find the greatest product of five consecutive digits in the 1000-digit number.

73167176531330624919225119674426574742355349194934

96983520312774506326239578318016984801869478851843

85861560789112949495459501737958331952853208805511

12540698747158523863050715693290963295227443043557

66896648950445244523161731856403098711121722383113

62229893423380308135336276614282806444486645238749

30358907296290491560440772390713810515859307960866

70172427121883998797908792274921901699720888093776

65727333001053367881220235421809751254540594752243

52584907711670556013604839586446706324415722155397

53697817977846174064955149290862569321978468622482

83972241375657056057490261407972968652414535100474

82166370484403199890008895243450658541227588666881

16427171479924442928230863465674813919123162824586

17866458359124566529476545682848912883142607690042

24219022671055626321111109370544217506941658960408

07198403850962455444362981230987879927244284909188

84580156166097919133875499200524063689912560717606

05886116467109405077541002256983155200055935729725

71636269561882670428252483600823257530420752963450

8.1 Introduction

There is an obvious naïve solution, shown in Listing 8.1. The character string `string` contains the number string.

8.2 Other Considerations

There are two improvements I thought to make to this code. Using a much longer sequence of digits I tested these idea.

The first idea involves the fact that each digit in the string gets converted from the character to the integer five times. We can convert these once in a single pass before computing products; however this appears to take approximately 26% longer. This solution is given in Listing 8.2.

Listing 8.1: Problem 8: `main()`

```
int main(int argc, char *argv[])
{
    uint64_t    maxProd = 0, curProd = 1, cur, len;
    char        copt;

    while((copt = getopt(argc, argv, "n:")) != -1) {
        switch(copt) {
            default:
                goto usage;
        }
    }
    len = strlen(string);
    for(cur = 4; cur <len; cur++) {
        curProd = (string[cur - 4] - '0') * (string[cur - 3]
            - '0') * (string[cur - 2] - '0') * (string[cur -1]
            - '0') * (string[cur] - '0');
        if(curProd > maxProd) maxProd = curProd;
    }
    printf("maxProd = %llu\n", maxProd);
    exit(0);
//error0:
    exit(-1);
usage:
    fprintf(stderr, "Usage: %s\n", argv[0]);
    exit(-1);
}
```

Listing 8.2: Problem 8: Shifting

```c
int main(int argc, char *argv[])
{
    uint64_t    i=0, n=0, maxProd = 0, curProd = 1, cur, len;
    char        copt;

    while((copt = getopt(argc, argv, "n:")) != -1) {
        switch(copt) {
            default:
                goto usage;
        }
    }
    len = strlen(string);
    for(i=0; i<len; i++) string[i] -= '0';
    for(cur = 4; cur <len; cur++) {
        curProd = string[cur - 4] * string[cur - 3] * string[cur - 2] * string[cur -1] * string[cur];
        if(curProd > maxProd) maxProd = curProd;
    }
    printf("maxProd = %llu\n", maxProd);
    exit(0);
error0:
    exit(-1);
usage:
    fprintf(stderr, "Usage: %s\n", argv[0]);
    exit(-1);
}
```

The second change is that instead of taking 995×5 products I could compute the first product of five digits. For each subsequent product I could divide out the first digit in the current list and multiply the new digit being added to the list. This requires some particular handling of the case where there is a zero in the list, but reduces the number of multiplications by approximately 60%. This solution takes approximately twice as long to run. This solution is given in Listing 8.3.

Listing 8.3: Problem 8: Shifting

```
int main(int argc, char *argv[])
{
    uint64_t    n=0, maxProd = 0, curProd = 1, cur, len, zeroCount = 5;
    char        copt;

    while((copt = getopt(argc, argv, "n:")) != -1) {
        switch(copt) {
            default:
                goto usage;
        }
    }
    len = strlen(string);
    curProd = (string[0] - '0') * (string[1] - '0') * (string[2] - '0') * (string[3] - '0') * (string[4] - '0');
    for(cur = 5; cur <len; cur++, zeroCount++) {
        if(string[cur] == '0') { string[cur] = '1'; zeroCount = 0; }
        curProd = (curProd * (string[cur] - '0'))/(string[cur-5] - '0');
        if(curProd > maxProd & zeroCount > 4) maxProd = curProd;
    }
    printf("maxProd = %llu\n", maxProd);
    exit(0);
error0:
    exit(-1);
usage:
    fprintf(stderr, "Usage: %s\n", argv[0]);
    exit(-1);
}
```

Chapter 9

Special Pythagorean triplet

A Pythagorean triplet is a set of three natural numbers, a b c, for which,

$$a^2 + b^2 = c^2$$

For example, $3^2 + 4^2 = 9 + 16 = 2^5 = 52$.

There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find the product abc.

9.1 Introduction

The naïve solution is to try each combination of (a, b, c) such that $a + b + c = 1000$ of which there are 10^6, determine if they constitute a Pythagorean triplet and compute the product. This solution is given in Listing 9.1.

Listing 9.1: Problem 9: `findTriple()`

```
int findTriple(uint64_t n, uint64_t *a, uint64_t *b, uint64_t
    *c)
{
    for(*a=1; *a<n; (*a)++) {
        for(*b=0; *b<*a; (*b)++) {
            *c=n - (*a) - (*b);
            if((*a) * (*a) + (*b) * (*b) == (*c) * (*c))
                return(0);
        }
    }
    return(-1);
}
```

9.2 Primitive Pythagorean Triplets

First a definition is in order to be clear,

Definition 9.1 (Pythagorean Triple)**.** Let $a,\ b,\ c \in \mathbb{N}$, then the triple (a, b, c) is a Pythagorean triple if $a^2 + b^2 = c^2$.

Thus we could enumerate the pairs of integers (a, b) and see if the resulting c is an integer; however this is slow. There is a special subset of triples that can be used to generate the full set, the primitive triples

Definition 9.2 (Primitive Pythagorean Triple)**.** A Pythagorean triple (a, b, c) is said to be primitive iff a, b and c are coprime.

If (a, b, c) is a primitive Pythagorean triple we can generate any other Pythagorean triple by multiplying by a factor k; that is, suppose that (a, b, c) is some non-primitive pythagorean triple, then let $k = \gcd(a, b, c)$ and let $a = a'k$, $b = b'k$ and $c = c'k$; then (a', b', c') is a primitive Pythagorean triple since

$$\begin{aligned} a^2 + b^2 &= c^2 \\ \implies \quad (ka')^2 + (kb')^2 &= (kc')^2 \\ \implies k^2 \left(a'^2 + b'^2\right) &= k^2 c' \\ \implies \quad a'^2 + b'^2 &= c'^2 \end{aligned}$$

and $\gcd(a', b', c') = 1$. Conversely we can see that taking any primitive triple we can multiply by any $k \in \mathbb{N}$ an the result is still a Pythagorean triple.

We now show that we can consider any pair of numbers $\gcd(m, n)$ such that $m > n$ and $(m, n) = 1$ and generate a primitive Pythagorean triple.

Theorem 9.3. *(a, b, c) is a primitive Pythagorean triple if and only if there exists $m, n \in \mathbb{N}$ such that $m > n$, $\gcd(m, n) = 1$, $m \not\equiv n \pmod 2$ and $(a, b, c) = (2mn, m^2 - n^2, m^2 + n^2)$.*

Proof. Suppose that $m, n \in \mathbb{N}$ with $m > n$ and $\gcd(m, n) = 1$; then let $a = 2mn$, $b = m^2 - n^2$ and $c = m^2 + n^2$ and we compute

$$\begin{aligned} a^2 + b^2 &= (2mn)^2 + \left(m^2 - n^2\right)^2 \\ &= (2mn)^2 + m^4 - 2m^2n^2 + n^4 \\ &= m^4 + 2m^2n^2 + n^4 \\ &= \left(m^2 + n^2\right)^2 \\ &= c^2 \end{aligned}$$

So that (a, b, c) are a Pythagorean triple. To show that $\gcd(a, b, c) = 1$ notice that since $m \not\equiv n \pmod 2$ that c is odd*. Let $p \mid \gcd(a, b, c)$, then $p > 2$. $p \mid c + b \implies p \mid 2m^2$ and $p \mid c - b \implies p \mid 2n^2$. Since p is odd we have $p \mid \gcd(m, n)$ which is a contradiction of $\gcd(m, n) = 1$; therefore $\gcd(a, b, c) = 1$ and hence (a, b, c) are primitive.

Conversely suppose that (a, b, c) are a primitive Pythagorean triple so that $a^2 + b^2 = c^2$. WLOG let a be even. If b is even then so is c; but then 2 divides all three and (a, b, c) is not primitive; so b is odd and hence c is odd; so $b - c$ and $b + c$ are even; set $c - b = 2j$ and $c - b = 2k$; then

$$\begin{aligned} a^2 &= c^2 - b^2 \\ &= (c - b)(c + b) \end{aligned}$$

Since a is even we can write

$$\left(\frac{a}{2}\right)^2 = \left(\frac{(c-b)}{2}\right)\left(\frac{(c+b)}{2}\right) = s \cdot t$$

$\square$

We can see that $\gcd(s, t) = 1$ since $\gcd(b, c) = 1$†.

Now we show that there exists $m, n \in \mathbb{N}$ such that $s = n^2$ and $t = m^2$. If $s = 1$ or $t = 1$ then the claim is vacuously true so we may assume that

* WLOG let $m = 2k$ be even and $n = 2j + 1$ be odd; then $c = m^2 + n^2 = (2k)^2 + (2j + 1)^2 = 4k^2 + 4j^2 + 4j + 1 \equiv 1 \pmod 2$.

† Suppose $p \mid \gcd(b, c)$ then $p \mid b^2$ and $p \mid c^2$ hence $p \mid a^2 \implies p \mid a$ which contradicts $\gcd(a, b, c) = 1$.

$s, \ t > 1$; then consider the prime factorizations as well as that of $\frac{a}{2}$. Since $\gcd(s,t) = 1$ we can see that each prime factor of s and t must occur twice in $\left(\frac{a}{2}\right)^2$ and hence twice in s and t; thus we may set $s = n^2$ and $t = m^2$. Now we can compute

$$c = t + s = m^2 + n^2 \tag{9.1}$$

$$b = t - s = m^2 - n^2 \tag{9.2}$$

and by extension

$$x = 2mn \tag{9.3}$$

establishing the equality. Suppose that $p \mid \gcd(m,n)$ then $p \mid (b,c)$ which is a contradiction so $\gcd(m,n) = 1$, also establishing that b is odd since otherwise $2 \mid \gcd(a,b,c)$ which is a contradiction.

9.3 Algorithm

Making use of the established enumeration we may work as follows. For each $m > 1$ consider $n < 1$ such that $m \not\equiv n \pmod 2$*. Construct the triplet

* For sufficiently large m it will become more efficient to compute the prime factorization of m and enumerate the $n < m$ that are coprime.

Listing 9.2: Problem 9: `findTriple2()`

```
int findTriple2(uint64_t x, uint64_t *a, uint64_t *b,
    uint64_t *c)
{
     uint64_t     m, n, k;
     for(m=1; m<x; m++) {
          for(n = m%2 == 0 ? 1 : 2; n<m; n+=2) {
               if(x%(2*m*m + 2*m*n) == 0) {
                    k=x/(2*m*m+2*m*n);
                    *a=k*2*m*n;
                    *b=k*(m*m-n*n);
                    *c=k*(m*m+n*n);
                    return(0);
               }
          }
     }
     return(-1);
}
```

$(2mn, m^2-n^2, m^2+n^2)$. If $1000 \equiv 0 \pmod{2m^2+2mn}$ then set

$$k = \frac{1000}{2m^2+2mn} \tag{9.4}$$

and compute $k^3(2mn)(m^2-n^2)(m^2+n^2) = 2k^3\left(m^5n - mn^5\right)$. This solution is given in Listing 9.2.

Chapter 10

Summation of Primes

The sum of the primes below 10 is $2 + 3 + 5 + 7 = 17$.

Find the sum of all the primes below two million.

10.1 Introduction

This problem is trivial given the use of the Sieve of Eratosthenes described in Section 7.2.

Listing 10.1: Problem 10: `sumPrimes()`

```
uint64_t sumPrimes(uint64_t n)
{
    uint64_t     *list = NULL, len, rc, i, sum = 0;

    rc = sieveE(n, &list, &len);
    if(list == NULL) goto error0;
    for(i=0; i<len; i++) sum+= list[i];
    free(list);
    return(sum);
error0:
    return(0);
}
```

Chapter 11

Largest Product In a Grid

In the 2020 grid below, four numbers along a diagonal line have been marked in red.

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08

49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00

81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65

52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91

22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80

24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50

32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70

67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21

24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72

21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95

78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92

16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57

86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58

19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40

04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66

88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69

04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36

20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16

20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54

01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

The product of these numbers is 26 63 78 14 = 1788696.

What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 2020 grid?

11.1 Introduction

This is easily solved by searching through all the possible products as shown in Listing 11.1. Note that `grid` is defined to be a two dimensional array containing the grid of numbers.

Listing 11.1: Problem 11: `findMax()`

```
uint64_t findMax(void)
{
    uint64_t    x, y, curProd, maxProd = 1;

    for(x=0; x<20; x++) {
        for(y=0; y<17; y++) {
            curProd = grid[x][y] * grid[x][y+1] * grid[x][y+2] * grid[x][y+3];
            if(curProd > maxProd) maxProd = curProd;
        }
    }
    for(x=0; x<17; x++) {
        for(y=0; y<20; y++) {
            curProd = grid[x][y] * grid[x+1][y] * grid[x+2][y] * grid[x+3][y];
            if(curProd > maxProd) maxProd = curProd;
        }
    }
    for(x=0; x<17; x++) {
        for(y=0; y<17; y++) {
            curProd = grid[x][y] * grid[x+1][y+1] * grid[x+2][y+2] * grid[x+3][y+3];
            if(curProd > maxProd) maxProd = curProd;
        }
    }
    for(x=3; x<20; x++) {
        for(y=0; y<17; y++) {
            curProd = grid[x][y] * grid[x-1][y+1] * grid[x-2][y+2] * grid[x-3][y+3];
            if(curProd > maxProd) maxProd = curProd;
        }
    }
    return(maxProd);
}
```

Chapter 12

Highly Divisible Triangular Number

The sequence of triangle numbers is generated by adding the natural numbers. So the 7th triangle number would be 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28. The first ten terms would be:

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \ldots$$

Let us list the factors of the first seven triangle numbers:

$$\begin{aligned}
1 &: 1 \\
3 &: 1, 3 \\
6 &: 1, 2, 3, 6 \\
10 &: 1, 2, 5, 10 \\
15 &: 1, 3, 5, 15 \\
21 &: 1, 3, 7, 21 \\
28 &: 1, 2, 4, 7, 14, 28
\end{aligned}$$

We can see that 28 is the first triangle number to have over five divisors.

What is the value of the first triangle number to have over five hundred divisors?

12.1 Introduction

The triangle numbers are the natural numbers of the form $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ as noted in Section 6.3. To count the divisors of a number x we write $x = \prod_{j=1}^{r} p_j^{\alpha_j}$ then for each j tuple $(\alpha_1, \alpha_2, \ldots, \alpha_r)$ there is a unique divisor; hence the number of divisors is $\prod_{j=1}^{r}(\alpha_j + 1)$.

The solution in Listing 12.1 simply enumerates the triangle numbers factoring each integer and scans the list of prime factors for the largest, denoted `maxPrime`. The algorithm then allocates an array, `factorList` of length `maxPrime +1` which is zero filled. This allows us to scan the list of prime factors and increment the element `factorList[list[j]]` to count the exponent of each prime number.

We could scan the entire list taking the product of each element `+1`; however this is likely inefficient, instead we scan the prime factor list for indexes into `factorList` and add the appropriate value to the product; then set the value to zero so that factors of multiplicity greater than 1 are not counted multiple times.

Listing 12.1: Problem 12: `findTriangle()`

```
uint64_t findTriangle(uint64_t n)
{
    uint64_t    i, j, sum, *list, len, *factorList, maxPrime, divisors = 1;
    for(i=1; ; i++) {
        sum = (i*(i+1))/2;
        factorN(sum, &list, &len);
        maxPrime = 0;
        for(j=0; j<len; j++) if (list[j] > maxPrime) maxPrime = list[j];
        factorList = calloc(maxPrime + 1, sizeof(uint64_t));
        for(j=0; j<len; j++) factorList[list[j]]++;
        divisors=1;
        for(j=0; j<len; j++) {
            divisors = divisors * (factorList[list[j]]  + 1);
            factorList[list[j]] = 0;
        }
        free(list);
        free(factorList);
        if(divisors > n) break;
    }
    return(sum);
}
```

Chapter 13

Large Sum

> Work out the first ten digits of the sum of the following one-hundred 50-digit numbers. ...

The number is listed in Appendix A.

13.1 Introduction

We could use several `uint64_t` integers to break the 50 digit number into three 19 digit components then write our own addition function; however the GNU Multiple Precision library handles this type of problem excellently at which point the problem becomes trivial, as in Listing 13.1.

Listing 13.1: Problem 13: `findSum()`

```
int findSum(void)
{
    char        *line = NULL;
    size_t      linecap = 0;
    mpz_t       input, sum;
    uint64_t    i;
    FILE        *f;

    mpz_init(input);
    mpz_init(sum);
    mpz_set_ui(sum, 0);
    f = fopen("p13.txt", "r");
    for(i=0; i<100; i++) {
        getline(&line, &linecap, f);
        mpz_set_str(input, line, 10);
        mpz_add(sum, sum, input);
    }
    printf("sum = ");
    mpz_out_str(stdout, 10, sum);
    printf("\n");
    return(0);
}
```

Chapter 14

Largest Collatz Sequence

The following iterative sequence is defined for the set of positive integers:

$$n \to \frac{n}{2} \qquad \text{(n is even)}$$

$$n \to 3n + 1 \qquad \text{(n is odd)}$$

Using the rule above and starting with 13, we generate the following sequence:

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers finish at 1.

Which starting number, under one million, produces the longest chain?

NOTE: Once the chain starts the terms are allowed to go above one million.

14.1 Introduction

There is a simple naïve solution of iterating over each value and computing the chain. This solution is given in Listing 14.1. Running this algorithm though we can see that for the relatively small input of 10^6 that this algorithm takes 0.270 s; moreover the algorithm appears to be super-linear in time.

Listing 14.1: Problem 14: `findMaxChain()`

```
int findMaxChain(uint64_t n, uint64_t *maxCount, uint64_t *maxI)
{
    uint64_t    curCount = 0, i, collatz;

    for(i=2; i<n; i++) {
        curCount = 1;
        collatz = i;
        do curCount++; while((collatz = collatz % 2 == 0 ? collatz >> 1 : (3*collatz) + 1 ) != 1);
        if(curCount > *maxCount) { *maxCount = curCount; *maxI = i; }
    }
    return(0);
}
```

14.2 Memoization

Suppose we compute the chain

$$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

We could save these results noting that 2 generates a chain of length 2, 4 generates a chain of length 3 and s forth. Then, next time a chain hits one of these values we no longer need to compute the rest of the chain. The primary issue with using memoization in this algorithm is that it's impossible to tell the maximum value one might encounter. In the above sequence the initial value of 30 reaches a maximum of 40. With this in mind we identify a maximum size of our memoization table and allocate memory for that table alone. While chains which have large numbers may not make complete use of the table the performance gains are still substantial.

The algorithm consists of two parts. The first is to compute the chain, storing the chain in the array `z[]`, until we reach a point in the table `memo[]`. Notice that the array `z[]` is used as a queue and is dynamically extended as necessary by factors of 2. The second part then stores values in the chain which fall within the limits of `memo[]` in the table for later use. While the algorithm, given in Listing 14.2, is considerably more complex it is an order of magnitude faster for inputs which generate chains with values about to the limits of `uint64_t`.

Listing 14.2: Problem 14 Memoized: `findMaxChain()`

```
#define MEMO_SIZE    2147483648
int findMaxChain(uint64_t n, uint64_t *maxCount, uint64_t *maxI)
{
    uint64_t    curCount = 0, i, j, collatz, *z, *memo, length, base;
    size_t      zlen = 2048, zCursor = 0;
    queue       *q;

    uint64_t    maxVal = 0;

    if((memo = (uint64_t *)mmap(NULL, (MEMO_SIZE)*sizeof(uint64_t), PROT_READ | PROT_WRITE, MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED) goto error0;
    if((q = queueCreate(NULL)) == NULL) goto error1;
    if((z = malloc(sizeof(uint64_t)*zlen)) == NULL) goto error2;
    memo[0] = 0; memo[1] = 1;
    *maxCount = 1; *maxI = 1;
    for(i=2; i<=n; i++) {
        collatz = i;
        curCount = 1;
        while(collatz > MEMO_SIZE || memo[collatz] == 0) {
            z[zCursor++] = collatz;
            if(zCursor == zlen) {
                if((z = realloc(z, sizeof(uint64_t)*zlen*2)) == NULL) goto error2;
                zlen *=2;
            }
            collatz = collatz % 2 == 0 ? collatz >> 1 : (3*collatz) + 1;
            if(collatz > maxVal) maxVal = collatz;
        }
        base = memo[collatz];
        length = zCursor;
        if((base + length) > *maxCount) { *maxCount = base + length; *maxI = i; }
        for(j=0; j<length; j++) {
            if(z[j] < MEMO_SIZE) memo[z[j]] = base + length - j;
        }
        zCursor = 0;
    }
    free(z);
```

```
    queueDestroy(q);
    munmap(memo, (n+1)*sizeof(uint64_t));
    return(0);
    free(z);
error2:
    queueDestroy(q);
    goto error1b;
error1:
    perror("mmap");
error1b:
    munmap(memo, (n+1)*sizeof(uint64_t));
error0:
    return(-1);
}
```

Chapter 15

Lattice Paths

> Starting in the top left corner of a 22 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner as in Figure 15.1[*]. How many such routes are there through a 20×20 grid?

15.1 Introduction

We can consider the possible paths through the grid as a sequence of movements right and movements down[†]. If we identify these with zero and one respectively we can write a path as a binary number. The length of the binary number is the taxicab distance[‡] from the upper left corner to the lower right

---

* From the project Euler website http://projecteuler.net/problem=15.

† We assume that we are never moving away from the destination, otherwise the number of paths are infinite.

‡ The taxicab distance, also known as the Manhattan distance or the L_1 norm in two dimensions for two points $x = (x_1, x_2)$ and $y = (y_1, y_2)$ is $d(x, y) = \sum_{k=1}^{2} |x_k - y_k|$.

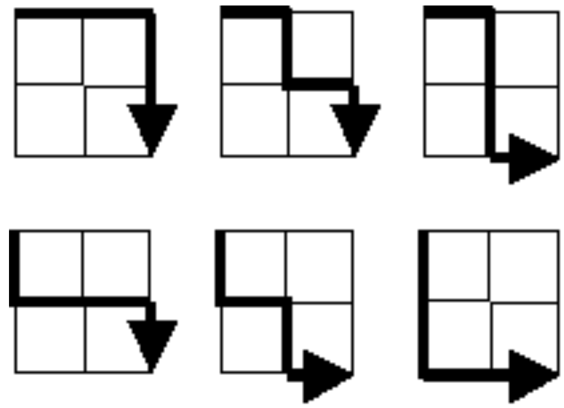

Figure 15.1: Paths through 2×2 Grid

corner. For grid of size $m \times n$ the distance is $m + n$.

Considering that we must move exactly m to the left and exactly n down we can see that the representation must have exactly m zeros and n ones. We can solve this problem by counting. If we write down where each zero is located there are $(m + n)$ options for the first zero, $(m + n - 1)$ for the second and so forth down to (n). For example in a $3x3grid$ we have six movements, we might places the zeroes in the first, second and fifth location, say $(1, 2, 5)$, but we might also identify $(1, 5, 2)$, $(2, 5, 1)$ and so forth. We must divide by the permutations of m objects, that is $m!$; thus the solution is

$$\frac{(m + n)!}{m!n!} \tag{15.1}$$

Listing 15.1: Problem 15: `findPaths()`

```
int findPaths(uint64_t m, uint64_t n)
{
    mpz_t       a, b, c;

    mpz_init(a);
    mpz_init(b);
    mpz_init(c);
    mpz_fac_ui(a, m);
    mpz_fac_ui(b, n);
    mpz_fac_ui(c, m+n);
    mpz_divexact(c, c, a);
    mpz_divexact(c, c, b);
    printf("paths = ");
    mpz_out_str(stdout, 10, c);
    printf("\n");
    return(0);
}
```

15.2 Programming the Solution

While we can write down the solution as

$$\frac{40!}{20!20!} \tag{15.2}$$

we are asked to encode the solution. Notice that $\log_2(40!) = 159.159$. Clearly until we have 256 bit numbers the computation is not straight forward; thus we are forced to use GMP (or similar). The solution is given in Listing 15.1.

Chapter 16

Power Digit Sum

$2^{15} = 32768$ and the sum of it's digits is $3 + 2 + 7 + 6 + 8 = 26$.

What is the sum of the digits of the number 6^{1000}?

16.1 Introduction

This is another problem which becomes trivial once GMP is employed. We need only use GMP to write the number as a string using `mpz_get_str()`. The length of the string can be computed using `mpz_sizeinbase()`. The solution is give in Listing 16.1

Listing 16.1: Problem 16: `findSum()`

```
int findSum(uint64_t n)
{
    char        *line = NULL, *cursor;
    mpz_t       exp;
    uint64_t    x = 0, digits;

    mpz_init(exp);
    mpz_ui_pow_ui(exp, 2, n);
    digits = mpz_sizeinbase(exp, 10);
    line = calloc(digits+2, sizeof(char));
    mpz_get_str(line, 10, exp);
    cursor = line;
    while(*cursor != 0) x += (*(cursor++) - '0');
    free(line);
    return(x);
}
```

Chapter 17

Number Letter Count

If the numbers 1 to 5 are written out in words: one, two, three, four, five, then there are $3 + 3 + 5 + 4 + 4 = 19$ letters used in total.

If all the numbers from 1 to 1000 (one thousand) inclusive were written out in words, how many letters would be used?

NOTE: Do not count spaces or hyphens. For example, 342 (three hundred and forty-two) contains 23 letters and 115 (one hundred and fifteen) contains 20 letters. The use of "and" when writing out numbers is in compliance with British usage.

17.1 Introduction

This is a rather uninspiring problem of adding up the letters in each word and multiplying them by the appropriate number of times. In order to figure out how to write the program you have all the information to compute it by hand.

Chapter 18

Maximum Path Sum I

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

$$
\begin{array}{ccccccc}
 & & & 3 & & & \\
 & & 7 & & 4 & & \\
 & 2 & & 4 & & 6 & \\
8 & & 5 & & 9 & & 3
\end{array}
\tag{18.1}
$$

That is, $3 + 7 + 4 + 9 = 23$.

Find the maximum total from top to bottom of the triangle in Appendix B

NOTE: As there are only 16384 routes, it is possible to solve this problem by trying every route. However, Problem 67, is the same challenge with a triangle containing one-hundred rows; it cannot be solved by brute force, and requires a clever method!

18.1 Introduction

One technique to easily compute the longest path is to consider the triangle as a Directed Acyclic Group (or DAG). Each node sits at under a number in the triangle and the cost to move from one node to a subsequent node is the value at that point. We connect each element of the bottom row to a single sink point. The shortest path for a DAG algorithm is time linear (in the number of edges*) rather than exponential in vertices. Since the algorithm does not depend on non-negative path weights we may compute the longest path by negating all the weights.

18.2 directed Acyclic Graphs

We first must understand what we mean by a directed acyclic graph, and we start with the graph†.

Definition 18.1 (Graph)**.** A graph is an ordered pair $G = \{V, E\}$ consisting of a finite‡ set $V \neq \varnothing$ and a set E of two element subsets of V. The elements

* The maximum number of edges is $|V|(|V|-1) = O\left(|V|^2\right)$.

† The following definitions are taken from [7].

‡ In a more general setting a graph may have an infinite set of vertices, but these are not considered here.

of V are called *vertices*. An element $e = \{a, b\} \in E$ is called an *edge* with end vertices a and b in V.

A directed graph (or digraph) is simply a graph where the edges only move in one direction

Definition 18.2 (Digraph)**.** A digraph is an ordered pair $G = \{V, E\}$ consisting of a finite set $V \neq \varnothing$ of vertices and a set E of *ordered pairs* (a, b) where $a \neq b$ are both elements of V. The elements of V are called vertices and the elements of E are called edges or sometimes arcs to distinguish them from the non-directed case.

In order to discuss acyclic graph we formalize the notion of moving through a graph.

Definition 18.3 (Trail)**.** A *trail* is a sequence $\{e_1, e_2, \ldots, e_n\}$ such that for $k \leqslant 1 < n$, e_k and e_{k+1} share a vertex. A *closed trail* is one in which e_1 and e_n share a vertex.

Definition 18.4 (Directed Acyclic Graph)**.** A directed acyclic graph (DAG) is a directed graph with no closed trails.

Definition 18.5 (Topological Sort)**.** A topological sort on a DAG G is an ordering of the vertices produced as follows:

1. Create an empty list L.

2. Find a vertex $v \in V$ with no inbound edge (for a DAG this is guaranteed to exist).

3. Add v to the end of the list L
4. Remove v and associated edges from the graph
5. If V is non-empty return to step 2

With these definitions we are prepared to compute the shortest path of a DAG. Given a DAG, a start vertex and an end vertex we execute a topological sort of the vertices. Using this we find the start vertex and label the vertex with length 0. Then for each vertex v we consider each inbound edge. The vertex on the other side of that edge will be labeled (because we are working through the vertices in the topological sort) with a length. We consider the length to that vertex plus the edge weight and find the lowest value for all edges which point to v and label v accordingly. Once we label the ending vertex we will have computed the minimum distance. The algorithm runs in $O(|E|)$ since $|E|$ will dominate $|V|$ and we are doing one computation for each edge.

18.3 Solving the Triangle

The triangle may be placed into a DAG by placing a vertex at each point in the triangle and labeling each edge away from the vertex with the value of the point. The vertices on the last row would all have edges to a single sink point. We can make use of the structure of the problem to avoid constructing the DAG and compute directly. Recall the triangle in 18.1. If we consider the values from top to bottom, from left to right as a single array we can see that the index of the first entry of each row is $0, 1, 3, 6$ and that in general these will be the triangle numbers $\frac{j(j+1)}{2}$ for row j (zero indexed). We can create a second array to hold the labelings. The first distance will be 0. For row j the k^{th} entry will (again, zero indexed) will consider distances from the $k-1$ and k entry in the previous row, with special cases for the end points $k = 0$ and $k = j$. We can simply find the labeling and edge weight for these two points.

We demonstrate using the example. For $j = 0$, $k = 0$ we set the distance to 0.

$$\begin{array}{ccccccc} & & & 3 & & & \\ & & 7 & & 4 & & \\ & 2 & & 4 & & 6 & \\ 8 & & 5 & & 9 & & 3 \end{array} \qquad \begin{array}{c} 0 \\ \\ \\ \\ \end{array} \tag{18.2}$$

For $j = 1$ and $k = 0$ we compute the sum of the previous distance (0) and the edge at $j = 0$, $k = 0$ (3). Likewise for $j = 1$, $k = 1$.

$$\begin{array}{ccccccc} & & & 3 & & & \\ & & 7 & & 4 & & \\ & 2 & & 4 & & 6 & \\ 8 & & 5 & & 9 & & 3 \end{array} \qquad \begin{array}{ccc} & 0 & \\ 3 & & 3 \\ & & \\ & & \end{array} \tag{18.3}$$

On row $j = 2$ the first entry $k = 0$ must be $3 + 7 = 10$, the last entry,

$j = 2$ and $k = 2$, must be $3 + 4 = 7$.

$$\begin{array}{ccccccc} &&&3&&& \\ &&7&&4&& \\ &2&&4&&6& \\ 8&&5&&9&&3 \end{array} \qquad \begin{array}{ccccc} &&0&& \\ &3&&3& \\ 10&&&&7 \end{array} \tag{18.4}$$

For the $j = 2$, $k = 1$ entry we have a choice. We may pass through $j = 1$, $k = 0$ with edge distance 7 or $j = 1$, $k = 1$ with edge distance 4. Since each node above it has a distance of 3 we choose $3 + 7 = 10$ over $3 + 4 = 7$ (recall, we are looking for the largest/longest path).

$$\begin{array}{ccccccc} &&&3&&& \\ &&7&&4&& \\ &2&&4&&6& \\ 8&&5&&9&&3 \end{array} \qquad \begin{array}{ccccc} &&0&& \\ &3&&3& \\ 10&&10&&7 \end{array} \tag{18.5}$$

On row $j = 3$ we fix the end points, $k = 0$ and $k = 3$ as before

$$\begin{array}{ccccccc} &&&3&&& \\ &&7&&4&& \\ &2&&4&&6& \\ 8&&5&&9&&3 \end{array} \qquad \begin{array}{ccccccc} &&&0&&& \\ &&3&&3&& \\ &10&&10&&7& \\ 12&&&&&&13 \end{array} \tag{18.6}$$

For $k = 1$ we choose between $10 + 2 = 12$ and $10 + 4 = 14$ and for $k = 2$

we choose between $10 + 4 = 14$ and $7 + 6 = 13$.

$$
\begin{array}{ccccccc}
 & & & 3 & & & \\
 & & 7 & & 4 & & \\
 & 2 & & 4 & & 6 & \\
8 & & 5 & & 9 & & 3
\end{array}
\qquad
\begin{array}{ccccccc}
 & & & 0 & & & \\
 & & 3 & & 3 & & \\
 & 10 & & 10 & & 7 & \\
12 & & 14 & & 14 & & 10
\end{array}
\tag{18.7}
$$

For the final distance we sum the distance to each point and the value at that point and choose the largest, in this case $14 + 9 = 23$.

18.4 Implementation

The solution described above is implemented in Listing 18.1.

Listing 18.1: Problem 18: `triangleMax()`

```
int64_t triangleMax(void)
{
    uint64_t    max, j, k, l, r, *distance;

    if((distance = calloc((ROWS*(ROWS+1))/2, sizeof(uint64_t))) == NULL) goto error0;
    distance[0] = 0;
    for(j=1; j<ROWS; j++) {
        distance[(j*(j+1))/2] = distance[(j*(j-1))/2] + triangle[(j*(j-1))/2];
        distance[(j*(j+1))/2 + j] = distance[(j*(j-1))/2 + (j-1)] + triangle[(j*(j-1))/2 + (j-1)];
        for(k=1; k<j; k++) {
            l = distance[(j*(j-1))/2 + k-1] + triangle[(j*(j-1))/2 + k - 1];
            r = distance[(j*(j-1))/2 + k] + triangle[(j*(j-1))/2 + k];
            distance[(j*(j+1))/2 + k] = l > r ? l : r;
        }
    }
    max = 0;
    for(j=0; j<ROWS; j++) {
        if(triangle[(ROWS*(ROWS-1))/2 + j] + distance[(ROWS*(ROWS-1))/2 + j] > max) max = triangle[(ROWS*(ROWS-1))/2 + j] + distance[(ROWS*(ROWS-1))/2 + j];
    }
    free(distance);
    return(max);
error0:
    return(-1);
}
```

Chapter 19

Counting Sundays

You are given the following information, but you may prefer to do some research for yourself.

- 1 Jan 1900 was a Monday.
- Thirty days has September,
  April, June and November.
  All the rest have thirty-one,
  Saving February alone,
  Which has twenty-eight, rain or shine.
  And on leap years, twenty-nine.
- A leap year occurs on any year evenly divisible by 4, but not on a century unless it is divisible by 400.

How many Sundays fell on the first of the month during the twentieth century (1 Jan 1901 to 31 Dec 2000)?

19.1 Introduction

Notice that $365 \equiv 1 \pmod 7$ and $366 \equiv 2 \pmod 7$ thus on non-leap years the first Sunday of each month changes by one day down to 1 then back to 7. On leap years the first Sunday of each month changes by two; that is, if the first Sunday of March 1901 is the 3^{rd} then the first Sunday of March 1902 is the 2^{nd}. Since the leap day is the last day of February is make sense to consider years starting in March (handling the first and last year accordingly).

We can write down the day of the month for the first Sunday of each Month from March 1901 through February 1902 as

$$D^{1901} = \{3, 7, 5, 2, 7, 4, 1, 6, 3, 1, 5, 2\}$$

then for the following year, as it's not a leap year we have

$$D^{1902} = \{2, 6, 4, 1, 6, 3, 7, 5, 2, 7, 4, 1\}$$

We can compute, using the ternary operator

$$D_k^{1902} = D_k^{1901} - 1 + \Big(\big(D_k^{1901} - 1\big) < 1 \ ? \ 7 \ : \ 0\Big)$$

Thus we can compute each year and sum the number of 1 elements.

19.2 Implementation

We implement this algorithm in the function `countFirstSunday()` in Listing 19.1.

Listing 19.1: Problem 19: `countFirstSunday()`

```
uint64_t countFirstSunday(uint64_t n)
{
    uint64_t    i, count = 0, year;
    int64_t     months[12] = { 3, 7, 5, 2, 7, 4, 1, 6, 3, 1, 5, 2 };

    for(year = 1901; year < n; year++) {
        for(i=0; i<12; i++) if(months[i] == 1) count++;
        if((year+1) % 4 != 0 || (year+1) % 400 == 0) for(i=0; i<12; i++) months[i] -= 1;
            else for(i=0; i<12; i++) months[i] -= 2;
        for(i=0; i<12; i++) months[i] += (months[i] < 1) ? 7 : 0;
    }
    for(i=0; i<10; i++) if(months[i] == 1) count++;
    return(count);
}
```

Chapter 20

Factorial Digit Sum

$n!$ means $n \times (n-1) \times \cdots \times 3 \times 2 \times 1$.

For example, $10! = 10 \times 9 \times \cdots \times 3 \times 2 \times 1 = 3628800$,

and the sum of the digits in the number 10! is $3+6+2+8+8+0+0 = 27$.

Find the sum of the digits in the number 100!

20.1 Introduction

For this problem I modified the solution to problem 16 in Section 16.1.

20.2 Implementation

The solution is given in Listing 20.1.

Listing 20.1: Problem 20: `factorialDigitSum()`

```
uint64_t factorialDigitSum(uint64_t n)
{
    char        *line = NULL, *cursor;
    mpz_t       f;
    uint64_t    x = 0, digits;

    mpz_init(f);
    mpz_fac_ui(f, n);
    digits = mpz_sizeinbase(f, 10);
    line = calloc(digits+2,sizeof(char));
    mpz_get_str(line, 10, f);
    cursor = line;
    while(*cursor != 0) x += (*(cursor++) - '0');
    free(line);
    return(x);
}
```

Chapter 21

Amicable Numbers

Let $d(n)$ be defined as the sum of proper divisors of n (numbers less than n which divide evenly into n). If $d(a) = b$ and $d(b)$ =a, where $a \neq b$, then a and b are an amicable pair and each of a and b are called amicable numbers.

For example, the proper divisors of 220 are $1, 2, 4, 5, 10, 11, 20, 22, 44, 55$ and 110; therefore $d(220) = 284$. The proper divisors of 284 are $1, 2, 4, 71$ and 142; so $d(284) = 220$.

Evaluate the sum of all the amicable numbers under 10000.

21.1 Introduction

First we note the problem's unfortunate choice of notation as $d(n)$ is sometimes used for the *number* of divisors and $\sigma(n)$ is used for the sum of divisors. Here

$d(n)$ will be defined to be $d(n) = \sigma(n) - n$.

Computing the proper divisors of a natural number, n, is an exercise in factoring n. The naïve solution would be to compute the prime factors with multiplicity and compute the product of each subset of prime factors. If n has j prime factors (with multiplicity) there are 2^j products to compute. For example the number 210 has prime factors $\{7, 5, 3, 2\}$ each with multiplicity 1; thus the 16 divisors are 1, 2, 3, $3 \cdot 2 = 6$, 5, $5 \cdot 2 = 10$, $5 \cdot 3 = 15$, $5 \cdot 3 \cdot 2 = 30$, 7, $7 \cdot 2 = 14$, $7 \cdot 3 = 21$, $7 \cdot 3 \cdot 2 = 42$, $7 \cdot 5 = 35$, $7 \cdot 5 \cdot 2 = 70$, $7 \cdot 5 \cdot 3 = 105$, $7 \cdot 5 \cdot 3 \cdot 2 = 210$.

For composites with a large number of factors, particularly of high multiplicity, such as $1024 = 2^1 0$ this becomes cumbersome. We will show that we can compute the sum of the divisors, which we will denote $\sigma(n)$ by a more efficient formula as shown in [8]. Let

$$n = \prod_{i \leqslant i \leqslant k} p_i^{e_i} \tag{21.1}$$

be the prime factorization of n; then

$$\sigma(n) = \prod_{1 \leqslant i \leqslant k} \frac{p_i^{(e_i+1)} - 1}{p_i - 1} \tag{21.2}$$

hence we can compute $\sigma(1024)$ as

$$\begin{aligned}\sigma(1024) &= \frac{2^{11} - 1}{1} \\ &= 2047 \qquad (21.3)\end{aligned}$$

and the sum of the proper divisors would be 1023 which is clearly a more efficient computation.

21.2 Sum of Divisor Function

Definition 21.1 (Sum of Divisors Function)**.** The *sum of divisors* function, denoted $\sigma(n)$ is a function $\sigma : \mathbb{N} \to \mathbb{N}$ defined by*

$$\sigma(n) = \sum_{d|n} d \tag{21.4}$$

More generally, let

$$\sigma_k(n) = \sum_{d|n} d^k \tag{21.5}$$

Now we show that σ is multiplicative and give a formula for computing it[5].

Theorem 21.2. *Let $n \in \mathbb{N}$, and let*

$$n = \prod_{j=1}^{s} p_j^{\alpha_j}$$

be the prime factorization of n; then

$$\sigma(n) = \prod_{j=1}^{s} \frac{{p_j}^{\alpha_j+1} - 1}{p_j - 1} \tag{21.6}$$

* The notation $\sum_{d|n} f(d)$ denotes a sum over all positive divisors of n.

implying that if $\gcd(m,n) = 1$ *then* $\sigma(mn) = \sigma(m)\sigma(n)$*; that is,* σ *is multiplicative**.

Proof. We prove by induction on s. Let $s = 1$ then

$$\begin{aligned}\sigma\left(p_1^{\alpha_1}\right) &= 1 + p_1 + p_1^2 + \dots p_1^{\alpha_1} \\ &= \frac{p_1^{\alpha_1+1} - 1}{p_1 - 1}\end{aligned} \tag{21.7}$$

Now, suppose that the theorem is true for $s = k - 1$; we show that the result holds for $s = k$. Let $n = n' p_k^{\alpha_k}$ where $p_k \nmid n'$; then

$$\begin{aligned}\sigma(n) &= \sigma\left(n'\right) + p_k\sigma\left(n'\right) + p_k^2\sigma\left(n'\right) + \dots + p_k^{\alpha_k}\sigma\left(n'\right) \\ &= \sigma\left(n'\right)\left(1 + p_k + p_k^2 + \dots p_k^{\alpha_k}\right) \\ &= \sigma\left(n'\right)\frac{{p_k}^{\alpha_k} - 1}{p_k - 1} \\ &= \prod_{j=1}^{k}\frac{{p_j}^{\alpha_j+1} - 1}{p_j - 1}\end{aligned}$$

□

21.3 Solution

We determine $d(n)$ for $1 \leqslant n \leqslant 10000$ and identify amicable pairs. To compute $d(n)$ we use a scheme similar to that in Chapter 12 where we compute the

* A multiplicative function is a function $f(n)$ such that for $a, b \in \mathbb{N}$ with $\gcd(a,b) = 1$ that $f(ab) = f(a)f(b)$.

prime factors and multiplicity. Notice that to compute the multiplicity of each prime we create an array of length `maxPrime` which is, in the worst case, length `n`; then scan this list. This is likely the most efficient method for small values of `n` but at larger values will dominate the runtime as this makes the algorithm $O\left(n^2\right)$.

Listing 21.1: Problem 21: `sumOfAmicable()`

```c
int64_t sumOfAmicable(uint64_t n)
{
    uint64_t    i, j, *list, len, *factorList, maxPrime;
    uint64_t    *d, x = 0;

    if((d = malloc(sizeof(uint64_t)*(n+1))) == NULL) goto error0;
    for(i=1; i<=n; i++) {
        factorN(i, &list, &len);
        if(list == NULL) goto error1;
        maxPrime = 1;
        for(j=0; j<len; j++) if(list[j] > maxPrime) maxPrime = list[j];
        if((factorList = calloc(maxPrime + 1, sizeof(uint64_t))) == NULL) goto error2;
        for(j=0; j<len; j++) factorList[list[j]]++;
        d[i] = 1;
        for(j=0; j<=maxPrime; j++) {
            if(factorList[j] != 0) {
                d[i] *= (pow(j, factorList[j]+1) - 1) / (j - 1);
            }
        }
        d[i] -=i;
        free(list);
    }
    for(i=1; i<=n; i++) {
        if(i == d[d[i]] && i != d[i]) x+=i;
    }
    free(factorList);
    free(d);
    return(x);
// error3:
    free(factorList);
error2:
    free(list);
error1:
    free(d);
error0:
    return(-1);
}
```

Chapter 22

Maximum Path Sum II

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

$$\begin{array}{ccccccc} & & & 3 & & & \\ & & 7 & & 4 & & \\ & 2 & & 4 & & 6 & \\ 8 & & 5 & & 9 & & 3 \end{array} \tag{22.1}$$

That is, $3 + 7 + 4 + 9 = 23$.

Find the maximum total from top to bottom of the triangle in `http://projecteuler.net/project/triangle.txt`

NOTE: This is a much more difficult version of Problem 18. It is not possible to try every route to solve this problem, as there are

$2^9$9 altogether! If you could check one trillion ($10^1$2) routes every second it would take over twenty billion years to check them all. There is an efficient algorithm to solve it. ;o)

22.1 Introduction

This problem yields to the same solution as in Section 18.4.

Chapter 23

Largest Exponential

Comparing two numbers written in index form like 2^{11} and 3^7 is not difficult, as any calculator would confirm that $2^{11} = 2048 < 3^7 = 2187$.

However, confirming that $632382^{518061} > 519432^{525806}$ would be much more difficult, as both numbers contain over three million digits.

Using the file `base_exp.txt`*, a 22K text file containing one thousand lines with a base/exponent pair on each line, determine which line number has the greatest numerical value.

NOTE: The first two lines in the file represent the numbers in the example given above.

---

* http://projecteuler.net/project/base_exp.txt

23.1 Introduction

The size of these values makes direct computation using machine size variables impossible and computation using arbitrary precision integers costly, however, notice that we can compute

$$\log\left(x^y\right) = y \cdot \log\left(x\right)$$

moreover, log is order preserving on $(0, \infty)$. Our only issue then is to ensure that the floating point type contains enough precision to resolve the differences on the log scale. The largest base is 999665 and the largest exponent is 1190800; so we must consider these ranges. If we consider a small change in y we see that it results in a difference of $\log\left(x^{y+1}\right) - \log\left(x^y\right)$. Compared to the value $\log\left(x^y\right)$ we have

$$\frac{\log\left(x^{y+1}\right) - \log\left(x^y\right)}{\log\left(x^y\right)} = \frac{1}{y}$$

Likewise considering the change in x we have

$$\begin{aligned} \frac{\log\left((x+1)^y\right) - \log\left(x^y\right)}{\log\left(x^y\right)} &= \frac{\log\left(x+1\right) - \log\left(x\right)}{\log\left(x\right)} \\ &= \frac{\log\left(x+1\right)}{\log\left(x\right)} - 1 \end{aligned}$$

Both of these are minimized when the variable is maximized and we evaluate at the limits

$$\frac{1}{1190800} = 8.39772 \times 10^{-7}$$

and

$$\frac{\log(999665+1)}{\log(999665)} - 1 = 7.24084 \times 10^{-8}$$

These values indicate that we may exceed the resolution of single precision floating point. The 80-bit `long double` will certainly resolve these differences. Were machine precision be insufficient to resolve the differences we may find the largest values which are identical up to the resolution and use an arbitrary precision library to handle differentiating these results.

23.2 Solution

We implement the solution in Listing 23.1.

Listing 23.1: Problem 99: `findMaxExponent()`

```c
uint64_t findMaxExponent(void)
{
    FILE        *f;
    uint64_t    i = 1, b, e, maxI = 0;
    long double val, maxVal = 0;
    char        c;

    if((f = fopen("p99.txt", "r")) == NULL) goto error0;
    while(fscanf(f, "%llu%c%llu", &b, &c, &e) != EOF) {
        val = e*logl(b);
        if(val > maxVal) { maxVal = val; maxI = i; }
        i++;
    }
    fclose(f);
    return(maxI);
error0:
    return(0);
}
```

Appendix A

Problem 13

37107287533902102798797998220837590246510135740250

46376937677490009712648124896970078050417018260538

74324986199524741059474233309513058123726617309629

91942213363574161572522430563301811072406154908250

23067588207539346171171980310421047513778063246676

89261670696623633820136378418383684178734361726757

28112879812849979408065481931592621691275889832738

44274228917432520321923589422876796487670272189318

47451445736001306439091167216856844588711603153276

70386486105843025439939619828917593665686757934951

62176457141856560629502157223196586755079324193331

64906352462741904929101432445813822663347944758178

92575867718337217661963751590579239728245598838407

58203565325359399008402633568948830189458628227828

80181199384826282014278194139940567587151170094390
35398664372827112653829987240784473053190104293586
86515506006295864861532075273371959191420517255829
71693888707715466499115593487603532921714970056938
54370070576826684624621495650076471787294438377604
53282654108756828443191190634694037855217779295145
36123272525000296071075082563815656710885258350721
45876576172410976447339110607218265236877223636045
17423706905851860660448207621209813287860733969412
81142660418086830619328460811191061556940512689692
51934325451728388641918047049293215058642563049483
62467221648435076201727918039944693004732956340691
15732444386908125794514089057706229429197107928209
55037687525678773091862540744969844508330393682126
18336384825330154686196124348767681297534375946515
80386287592878490201521685554828717201219257766954
78182833757993103614740356856449095527097864797581
16726320100436897842553539920931837441497806860984
48403098129077791799088218795327364475675590848030
87086987551392711854517078544161852424320693150332
59959406895756536782107074926966537676326235447210
69793950679652694742597709739166693763042633987085
41052684708299085211399427365734116182760315001271

65378607361501080857009149939512557028198746004375
35829035317434717326932123578154982629742552737307
94953759765105305946966067683156574377167401875275
88902802571733229619176668713819931811048770190271
25267680276078003013678680992525463401061632866526
36270218540497705585629946580636237993140746255962
24074486908231174977792365466257246923322810917141
91430288197103288597806669760892938638285025333403
34413065578016127815921815005561868836468420090470
23053081172816430487623791969842487255036638784583
11487696932154902810424020138335124462181441773470
63783299490636259666498587618221225225512486764533
67720186971698544312419572409913959008952310058822
95548255300263520781532296796249481641953868218774
76085327132285723110424803456124867697064507995236
37774242535411291684276865538926205024910326572967
23701913275725675285653248258265463092207058596522
29798860272258331913126375147341994889534765745501
18495701454879288984856827726077713721403798879715
38298203783031473527721580348144513491373226651381
34829543829199918180278916522431027392251122869539
40957953066405232632538044100059654939159879593635
29746152185502371307642255121183693803580388584903

41698116222072977186158236678424689157993532961922
62467957194401269043877107275048102390895523597457
23189706772547915061505504953922979530901129967519
86188088225875314529584099251203829009407770775672
11306739708304724483816533873502340845647058077308
82959174767140363198008187129011875491310547126581
97623331044818386269515456334926366572897563400500
42846280183517070527831839425882145521227251250327
55121603546981200581762165212827652751691296897789
32238195734329339946437501907836945765883352399886
75506164965184775180738168837861091527357929701337
62177842752192623401942399639168044983993173312731
32924185707147349566916674687634660915035914677504
99518671430235219628894890102423325116913619626622
73267460800591547471830798392868535206946944540724
76841822524674417161514036427982273348055556214818
97142617910342598647204516893989422179826088076852
87783646182799346313767754307809363333018982642090
10848802521674670883215120185883543223812876952786
71329612474782464538636993009049310363619763878039
62184073572399794223406235393808339651327408011116
66627891981488087797941876876144230030984490851411
60661826293682836764744779239180335110989069790714

85786944089552990653640447425576083659976645795096
66024396409905389607120198219976047599490197230297
64913982680032973156037120041377903785566085089252
16730939319872750275468906903707539413042652315011
94809377245048795150954100921645863754710598436791
78639167021187492431995700641917969777599028300699
15368713711936614952811305876380278410754449733078
40789923115535562561142322423255033685442488917353
44889911501440648020369068063960672322193204149535
41503128880339536053299340368006977710650566631954
81234880673210146739058568557934581403627822703280
82616570773948327592232845941706525094512325230608
22918802058777319719839450180888072429661980811197
77158542502016545090413245809786882778948721859617
72107838435069186155435662884062257473692284509516
20849603980134001723930671666823555245252804609722
53503534226472524250874054075591789781264330331690

Appendix B

Problem 18

```
                            75
                          95  64
                        17  47  82
                      18  35  87  10
                    20  04  82  47  65
                  19  01  23  75  03  34
                88  02  77  73  07  63  67
              99  65  04  28  06  16  70  92
            41  41  26  56  83  40  80  70  33
          41  48  72  33  47  32  37  16  94  29
        53  71  44  65  25  43  91  52  97  51  14
      70  11  33  28  77  73  17  78  39  68  17  57
    91  71  52  38  17  14  91  43  58  50  27  29  48
  63  66  04  68  89  53  67  30  73  16  69  87  40  31
04  62  98  27  23  09  70  98  73  93  38  53  60  04  23
```

Bibliography

[1] H. S. Wilf, *Generatingfunctionology.* 888 Worchester Street, Suite 230 Wellesley, MA 02482: A K Peters, Ltd., 2006.

[2] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Math.* Addison-Wesley Publishing, second ed., 1994.

[3] R. A. Mollin, *Fundaemntal Number Theory with Applications.* Chapman & Hall/CRC, 2008.

[4] Y. Zhang, "Bounded gaps between primes," *Annals of Mathematics*, 2013.

[5] G. E. Andrews, *Number Theory.* Dover Publications, 1994 Reprint.

[6] P. Duscart, *Autour de la fonction qui compte le nombre de nombres premiers.* PhD thesis, l'Université de Limoges, 1998.

[7] D. Jungnickel, *Graphs, Networks and Algorithms*, vol. 5 of *Algorithms and Computation in Mathematics.* Springer-Verlag, second ed., 2004.

[8] E. Bach and J. Shallit, *Algebraic Number Theory*, vol. 1. The MIT Press, 1996.

Index