Project Euler Problems

Michael E. Conlen

June 16, 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

1 Sum of Natural Numbers Divisible by 3 and 5 **1**

1.1 Introduction 1

1.2 Direct Computation 2

2 Sum of Even Fibonacci Numbers **5**

2.1 Introduction 5

2.2 Order of F_n 6

2.3 Computing `fib(n)` Directly 10

2.4 Sum of Even Fibonacci Numbers 10

2.5 Finding the Index 12

2.6 Direct Computation 14

2.7 Generalization 15

3 Largest Prime Factor of n **17**

3.1 Introduction 17

3.2 Remarks 18

4 Largest Palindrome Product **19**

4.1 Introduction 19

4.2 Palindromic Numbers 20

4.2.1 Introduction 20

4.2.2 Predecessor and Successor 20

4.2.3 Computing Integer from Representation 23

4.2.4 Finding an Answer 25

5 Smallest Multiple **29**

5.1 Introduction 29

5.2 Computing the Greatest Common Divisor 31

5.3 Solution 33

6 Sum Square Difference **35**

6.1 Introduction 36

6.2 Exponential Generating Functions 36

6.3 Sum of Integers 37

6.4 Sum of Integers Squared 40

6.5 Solution 40

7 The n^{th} Prime Number **42**

7.1 Introduction 42

7.2 Sieve of Eratosthenes 43

7.3 Sizing the Sieve 45

8 Largest Product **47**

8.1 Introduction 48

8.2 Other Considerations 48

A Factoring Integers **53**

A.1 Introduction 53

A.2 `factorN()` 53

Listings

1.1 Problem 1: Naïve Solution 2

1.2 Problem 1: C Solution 4

2.1 Problem 2: Naïve Solution 6

2.2 Problem 2: `fib(n)` Direct 11

2.3 Problem 2: Direct Solution 14

2.4 Problem 2: General Solution 16

3.1 Problem 3: Naïve Solution 18

4.1 Problem 4: `palindromeSuccessor()` 22

4.2 Problem 4: `palindromePredecessor()` 24

4.3 Problem 4: `palindromeInteger()` 25

4.4 Problem 4: `twoFactor()` 26

4.5 Problem 4: `main()` 27

5.1 Problem 5: `gcd()` 32

5.2 Problem 5: `lcm()` 33

5.3 Problem 5: `main()` 34

6.1 Problem 6: `SSD()` 41

7.1 Problem 7: `sieveE()` 44
7.2 Problem 7: `findBound()` 46
8.1 Problem 8: `main()` 49
8.2 Problem 8: Shifting 50
8.3 Problem 8: Shifting 52
A.1 `factor.h` 54
A.2 `factor.c:factorN()` 55

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

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

thus

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

We rewrite equation 2.3

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

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

$$F_k = \frac{\varphi^k - \psi^k}{\sqrt{5}} \tag{2.7}$$

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 1.618033988749895
#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

```
#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.

* See Appendix A.

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 Remarks

There are a number of more efficient algorithms for larger composite numbers to investigate including the general number sieve.

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()`

```
1  int palindromeSuccessor(palindromeN *n)
2  {
3      double       k;
4
5      k = log10(n->a + 1);
6      if(k - floor(k) < 1e-15 && (int)k >= 1) {    // k in N
7          if(n->b == PALINDROME_ODD) {
8              n->a = (n->a + 1)/10;
9              n->b = PALINDROME_EVEN;
10         } else {
11             n->a++;
12             n->b = PALINDROME_ODD;
13         }
14     } else { // k not in N
15         n->a++;
16     }
17     return(0);
18 error0:
19     return(-1);
20 }
```

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()`

```
1  int palindromePredecessor(palindromeN *n)
2  {
3      double k;
4
5      if(n->a == 0 && n->b == PALINDROME_ODD) {
6          n->b = PALINDROME_UNDEF;
7          return(0);
8      }
9      if(n->a == 1 && n->b == PALINDROME_ODD) {
10         n->a--;
11         return(0);
12     }
13     k = log10(n->a);
14     if(k - floor(k) < 1e-15) {
15         if(n->b == PALINDROME_ODD) {
16             n->a--;
17             n->b = PALINDROME_EVEN;
18         } else {
19             n->a = (n->a - 1)*10 + 9;
20             n->b = PALINDROME_ODD;
21         }
22     } else {
23         n->a--;
24     }
25
26     return(-1);
27 }
```

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()`

```
1  int twoFactor(uint64_t *list, uint64_t len, uint64_t *x,
       uint64_t *y, uint64_t n)
2  {
3      uint64_t     i;
4
5      if(n >= (uint64_t)1 << len) goto error0; // n is out of
           bounds
6      *x = 1; *y = 1;
7      for(i=0; i<len; i++) if((n & ((uint64_t)1 << i)) == 0) *y
           *= list[i]; else *x *= list[i];
8      return(0);
9
10 error0:
11     return(-1);
12 }
```

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()`

```
#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;
```

```
25
26      for(i=0; i<len; i++) {
27          x += 9*pow(10, i);
28      }
29      n = x*x;
30      strlen = (uint64_t)floor(log10(n)) + 1;
31      p.b = ((strlen%2) == 0) ? PALINDROME_EVEN :
            PALINDROME_ODD;
32      if((str = malloc(sizeof(char)*(strlen + 1))) == NULL)
            goto error0;
33      sprintf(str, "%llu\n", n);
34      halfLen = (strlen/2.0);
35      if(p.b == PALINDROME_ODD) halfLen++;
36      p.a = 0;
37      for(i=0; i< halfLen; i++) {
38          p.a += (str[i] - '0')*pow(10, halfLen - 1 - i);
39      }
40      if(n < palindromeInteger(p)) palindromePredecessor(&p);
41      x = pow(10, len - 1);
42      min = x*x;
43      while((curN = palindromeInteger(p)) >= min) {
44          factorN(curN, &factorList, &factorLen);
45          for(i=0; i< ((uint64_t)1 << factorLen); i++) {
46              twoFactor(factorList, factorLen, &x, &y, i);
47                  if(len == floor(log10(x)) + 1 && len == floor
                        (log10(y)) + 1) {
48                      printf("Integer %llu factors into %llu
                            and %llu\n", curN, x, y);
49                      goto done;
50                  }
51          }
52          palindromePredecessor(&p);
53      }
54      printf("found no solutions\n");
55  done:
56      exit(0);
57  error0:
58      exit(-1);
59
60  usage:
61      fprintf(stderr, "%s -n N\n", argv[0]);
62      exit(-1);
63  }
```

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 $\text{lcm}(a, b) = n$.

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, $c \mid a$. □

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 $\ell = ab$. □

Lemma 5.5. *Let* $a, b \in \mathbb{N}\backslash\{0\}$ *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 $c = 1$. □

Theorem 5.6. *Let* $a,\ b \in \mathbb{Z}\backslash\{0\}$. *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. □

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()`

```
1  uint64_t gcd(uint64_t a, uint64_t b)
2  {
3      uint64_t     t;
4      while(b != 0) {
5          t = b;
6          b = a % t;
7          a = t;
8      }
9      return(a);
10 }
```

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

$$a = bq_0 + r_0$$

$$b = r_0q_1 + r_1$$

$$r_0 = r_1q_2 + r_2$$

$$\dots$$

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

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()`

```
1 uint64_t lcm(uint64_t a, uint64_t b)
2 {
3     return((a*b)/gcd(a, b));
4 }
```

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 $(x^2 + 3x + 1)\, e^x$. 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()`

```
1 unsigned long long int SSD(unsigned long long  n)
2 {
3     return((3*n*n*n*n+2*n*n*n-3*n*n-2*n)/(12));
4 }
```

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 con-

cerning 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()`

```
1  int sieveE(uint64_t x, uint64_t **list, uint64_t *len)
2  {
3      uint64_t     i, j, count=0;
4      uint64_t     *s;
5
6      if(x < 2) goto error0;
7      if((s = calloc(x+1, sizeof(uint64_t))) == NULL) goto
           error0;
8      s[0] = 1; s[1] = 1;
9      for(i=2; i<=x; i++) {
10         for(j=2; j*i<=x; j++) {
11             s[i*j] = 1;
12         }
13     }
14     *len = 0;
15     for(i=2; i<=x; i++) if(s[i] == 0) (*len)++;
16     if((*list = malloc(sizeof(uint64_t)* *len)) == NULL) goto
           error1;
17     for(i=2; i<=x; i++) if(s[i] == 0) (*list)[count++] = i;
18
19     free(s);
20     return(0);
21 error1:
22     free(s);
23 error0:
24     return(-1);
25 }
```

* 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()`

```
1  int main(int argc, char *argv[])
2  {
3      uint64_t     n=0, maxProd = 0, curProd = 1, cur, len;
4      char         copt;
5
6      while((copt = getopt(argc, argv, "n:")) != -1) {
7          switch(copt) {
8              default:
9                  goto usage;
10         }
11     }
12     len = strlen(string);
13     for(cur = 4; cur <len; cur++) {
14         curProd = (string[cur - 4] - '0') * (string[cur - 3]
               - '0') * (string[cur - 2] - '0') * (string[cur -1]
                - '0') * (string[cur] - '0');
15         if(curProd > maxProd) maxProd = curProd;
16     }
17     printf("maxProd = %llu\n", maxProd);
18     exit(0);
19 error0:
20     exit(-1);
21 usage:
22     fprintf(stderr, "Usage: %s\n", argv[0]);
23     exit(-1);
24 }
```

Listing 8.2: Problem 8: Shifting

```
1  int main(int argc, char *argv[])
2  {
3      uint64_t     i=0, n=0, maxProd = 0, curProd = 1, cur, len;
4      char         copt;
5
6      while((copt = getopt(argc, argv, "n:")) != -1) {
7          switch(copt) {
8              default:
9                  goto usage;
10         }
11     }
12     len = strlen(string);
13     for(i=0; i<len; i++) string[i] -= '0';
14     for(cur = 4; cur <len; cur++) {
15         curProd = string[cur - 4] * string[cur - 3] * string[
               cur - 2] * string[cur -1] * string[cur];
16         if(curProd > maxProd) maxProd = curProd;
17     }
18     printf("maxProd = %llu\n", maxProd);
19     exit(0);
20 error0:
21     exit(-1);
22 usage:
23     fprintf(stderr, "Usage: %s\n", argv[0]);
24     exit(-1);
25 }
```

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

```
1  int main(int argc, char *argv[])
2  {
3      uint64_t     n=0, maxProd = 0, curProd = 1, cur, len,
           zeroCount = 5;
4      char         copt;
5
6      while((copt = getopt(argc, argv, "n:")) != -1) {
7          switch(copt) {
8              default:
9                  goto usage;
10         }
11     }
12     len = strlen(string);
13     curProd = (string[0] - '0') * (string[1] - '0') * (string
           [2] - '0') * (string[3] - '0') * (string[4] - '0');
14     for(cur = 5; cur <len; cur++, zeroCount++) {
15         if(string[cur] == '0') { string[cur] = '1'; zeroCount
               = 0; }
16         curProd = (curProd * (string[cur] - '0'))/(string[cur
               -5] - '0');
17         if(curProd > maxProd & zeroCount > 4) maxProd =
               curProd;
18     }
19     printf("maxProd = %llu\n", maxProd);
20     exit(0);
21 error0:
22     exit(-1);
23 usage:
24     fprintf(stderr, "Usage: %s\n", argv[0]);
25     exit(-1);
26 }
```

Appendix A

Factoring Integers

A.1 Introduction

Factoring integers is a key component of several problems; therefore a factoring library exists to factor integers and return a list of the prime factors. The header, `factor.h`, is given in listing A.1. The code in `factor.c` is given in listing A.2.

A.2 `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 A.1: `factor.h`

```
1  #ifndef FACTOR_H
2  #define FACTOR_H
3
4  #include <config.h>
5  #include <stdint.h>
6
7  #if HAVE_CUNIT_CUNIT_H
8  int init_factor(void);
9  int clean_factor(void);
10 void     unit_factorN(void);
11 void     unit_twoFactor(void);
12 #endif
13
14 int factorN(unsigned long long n, unsigned long long **list,
       unsigned long long *len);
15 int twoFactor(uint64_t *list, uint64_t len, uint64_t *x,
       uint64_t *y, uint64_t n);
16 #endif
```

Listing A.2: factor.c:factorN()

```
1  int factorN(unsigned long long n, unsigned long long **list,
       unsigned long long *len)
2  {
3      unsigned long long  sn, i, *f;
4      queue              *q;
5
6      if((q = queueCreate(NULL)) == NULL) goto error0;
7      sn = sqrt(n);
8      while((n & 1) == 0) {
9          f = malloc(sizeof(unsigned long long));
10         *f = 2;
11         queueEnqueue(q, f);
12         n = n >> 1;
13     }
14     sn = sqrt(n);
15     for(i=3; i<=sn; i+=2) {
16         while(n%i == 0) {
17             f = malloc(sizeof(unsigned long long));
18             *f = i;
19             queueEnqueue(q, f);
20             n = n/i;
21             sn = sqrt(n);
22         }
23     }
24     if(n != 1) {
25         f = malloc(sizeof(unsigned long long));
26         *f = n;
27         queueEnqueue(q, f);
28     }
29     *len = queueLength(q);
30     *list = malloc(sizeof(unsigned long long) * (*len));
31     for(i=0; i< *len; i++) {
32         f = queueDequeue(q);
33         (*list)[i] = *f;
34         free(f);
35     }
36     queueDestroy(q);
37     return(0);
38 error0:
39     return(-1);
40 }
```

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. Molin, *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.

Index

Euclid's lemma, 30

Euclidean division, 31

exponential generating function, 36

Fibonacci number, 5

gcd, *see* greatest common divisor

generating function, 7

    exponential, 36

greatest common divisor, 30

lcm, *see* least common multiple

least common multiple, 29

palindromic number, 20

prime, 17

Sieve of Eratosthenes, 43