Project Euler Problems

Michael E. Conlen

June 2, 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.

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

```c
#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

```c
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,

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

[*] 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

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

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

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

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

```c
#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

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

$\square$

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

```c
#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

Naïvely we may divide n by the sequence of natural numbers $\{2, 3, 5, 7, \ldots, \lfloor\sqrt{n}\rfloor\}$ saving the result for further computation. This removes successively larger primes from the quotient until the last is found.

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 prove that there is no prime p such that $p > n_k$ and $p \mid n$.

We implement this algorithm in Listing 3.1.

Listing 3.1: Problem 3: Naïve Solution

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

int main(int argc, char *argv[])
{
    unsigned long long  n = 600851475143, sn;
    unsigned long long  i,j,k;

    sn = sqrt(n);
    while(n & 1 == 0) n = n >> 1;
    for(i=3; i< sn; i+=2) {
        while(n%i == 0) { n = n/i; sn = sqrt(n); }
    }
    printf("%llu\n", n);
    return(0);
}
```

3.2 Remarks

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

Index