The following are solutions to problems found on the Project Euler website. The problem descriptions are given in the code itself. Although this sheet may contain a restatement of the problems, it is mostly intended for a more thorough and mathematical explanation of the solutions as opposed to bare code. Not all problems may appear. For benchmarking purposes, here is the relevant system data for the machine that all programs were tested on. The source code can be found at `https://github.com/patl1/project_euler`.

CPU info:

vendor id	:	GenuineIntel
model name	:	Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz
cpu MHz	:	800.000
cache size	:	3072 KB
cpu cores	:	2

Other info:

Operating System	:	Fedora 18 (Spherical Cow)
Kernel Version	:	3.9.6-200.fc18.x86_64
Memory	:	4GB DDR3 SDRAM, 1066 MHz
GCC Version	:	gcc (GCC) 4.7.2 20121109 (Red Hat 4.7.2-8)

Problem 1

Multiples of 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.

Firstly, the classic summation formula for a series of numbers is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. In general, this can be extended to $\sum_{i=a}^{b} i = \frac{b(a+b)}{2}$ where b is the maximum number and a is the minimum number (proof omitted). Since here the summation is every third or fifth number, we have that the number of terms to sum is not b but $\frac{b}{a}$. For ease of reading, we define $n = \lfloor \frac{b}{a} \rfloor$. Then we have $\sum_{\substack{i=a \\ i=i+m}}^{b} i = \frac{n(a+b)}{2}$ where m is the multiple. With a little bit of algebraic manipulation, we arrive at $\sum_{\substack{i=a \\ i=i+m}}^{b} i = \frac{na(1+\frac{b}{a})}{2} = \frac{na(1+n)}{2}$. While it seems trivial, replacing $(a+b)$ with $a(1+n)$ is necessary to insure that the summation has the correct number of integers and does not accidentally include a number that is too high because of rounding errors. So now we have $\sum_{\substack{i=a \\ i=i+m}}^{b} i = \frac{na(1+n)}{2}$. To find the summation of all multiples of three and five, we can use this formula for $m = 3$ and $m = 5$. However, this double-counts numbers where i is a multiple of both 3 and 5 (a multiple of 15). So we subtract out all multiples of fifteen to go from double-counting them to single-counting. The final mathematics looks like this:

$$\sum_{\substack{i=1 \\ i=i+3}}^{999} i + \sum_{\substack{i=1 \\ i=i+5}}^{999} i = \frac{\lfloor \frac{999}{3} \rfloor (3)(1 + \lfloor \frac{999}{3} \rfloor)}{2} + \frac{\lfloor \frac{999}{5} \rfloor (5)(1 + \lfloor \frac{999}{5} \rfloor)}{2} - \frac{\lfloor \frac{999}{15} \rfloor (15)(1 + \lfloor \frac{999}{15} \rfloor)}{2}$$

Assuming one's button-pushing skills are correct, this expression should evaluate to 233,168.

Problem 2

The iterative solution is relatively basic, and needs no explanation.

The slightly more mathematical solution is based on the ratios of the Fibonacci numbers. As the Fibonacci sequence approaches infinity, the ratio of the one number to the next in sequence approaches the golden ratio, $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803398875$. We use the fact that $\frac{F_{n+1}}{F_n} \approx \phi$ for all integers n. Every third number in the sequence is even, since the sum of an even and an odd must be odd, and the sum of two odds must be even. Then we can see that

$$\begin{aligned}\frac{F_{n+3}}{F_n} &= \frac{F_{n+3}}{F_{n+2}}\frac{F_{n+2}}{F_n} \\ &= \frac{F_{n+3}}{F_{n+2}}\frac{F_{n+2}}{F_{n+1}}\frac{F_{n+1}}{F_n}\end{aligned}$$

and since for all integers n, $\frac{F_{n+1}}{F_n} \approx \phi$,

$$\begin{aligned}\frac{F_{n+3}}{F_n} &= \left(\frac{F_{n+3}}{F_{n+2}}\right)\left(\frac{F_{n+2}}{F_{n+1}}\right)\left(\frac{F_{n+1}}{F_n}\right) \\ &= (\phi)(\phi)(\phi) \\ &= \phi^3\end{aligned}$$

Therefore, the ratio of each even number to the next even number in the sequence is roughly ϕ^3.

The performance methods between the two methods is of interest. The iterative method definitely executes more instructions as it goes through all elements of the Fibonacci sequence, whereas the golden ratio method does not. However, the golden ratio method uses floating point numbers, which are inexact and take more time to compute on average. Each calculation must also be rounded to the appropriate even number so that the calculation is correct, another step which takes time to execute.

The two methods were timed to see which would run faster. The results are summarized below. For benchmarking info and machine specs, please refer back to the beginning of the paper. Timings were taken using internal C functions in < `sys/time.h` >. These are the averages taken over 1000 runs of the code, and the units are standard metric seconds.

```
Iterative solution time:       0.000000743s
Multiplicative solution time:  0.000014412s
```

Both implementations were very quick. Surprisingly, the iterative solution (in C) performed faster than the more "intelligent" multiplicative solution. Not only faster, but two orders of magnitude faster, on average. This is most likely due to the extra multiplications, power functions, rounding, and casting present in the multiplicative solution that is not present in the iterative one. Presumably these extra functions, especially the `pow()` function call, provided a fair amount of overhead, enough so that the simple additions in the iterative solution were faster. Presumably, there would be a given input integer where the extra data computed by the iterative solution slows down the code enough to be slower than the multiplicative solution. This limit was not explored.

Curiously, the Octave simulation did not always agree with the above result. However, the timing functions for Octave were not as precise, and could not give good enough numbers to be confidently included.

The final answer is 4,613,732.

Problem 3

Prime Factorization: The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143?

This one is actually easier to see by looking at the code, but only include a snippet will be included here.

```
1   long numm = 600851475143;
2   int greatestfactor = 1;
3   int x;
4   int STOP = floor(sqrt(numm));
5   for(x = 2; x < STOP; x++)
6   {
7     while(numm % x == 0)  //this x is a factor
8     {
9       greatestfactor = x;
10      numm /= x;     //to find duplicates of this factor
11    }
12  }
```

A thorough explanation and breakdown of the code is a bit unnecessary. However, there are a few key things to note.

1. The stopping condition (`STOP`) is at $\lfloor\sqrt{\texttt{numm}}\rfloor$. There is no need to search any number higher than the square root, as those numbers already have a smaller "complement" that has already been checked.

2. It divides out primes as it goes along (line 10). For example, the prime factorization of 90 is 2,3,3,5. After finding 2, the "new" number to factorize is 45. After the first 3, the "new" number is 15. After the second 3, the "new" number is 5. After finding the 5, the "new" number is 1, so it's done.

3. It finds multiple roots. Lines 7-11 take care of multiple root finding by checking the number multiple times for the same root.

4. There is no concept of "primeness" programmed in here anywhere, but it still finds all the prime numbers. This is because is starts at the beginning and then works its way up, dividing the primes out as it goes along. Point number 2 somewhat hits on this. After diving by 2 as many times as possible, there won't be any even-numbered factors left. Similarly, after finding as many roots of 3 as possible, there won't be any multiples of three left. The same with 5, 7, 11, and so on. Of course, extraneous numbers such as 4, 6, 8, and 10 are still checked, but they won't ever find anything.

5. The largest prime factor, what we were interested in since the beginning, is the last one found.

The main upside to this algorithm is its simplicity. It's relatively easy to follow. However, it isn't the most efficient. As stated in the previous paragraph, numbers like 2, 4, 6, 8, 9, and so on are still checked, even though they would be mathematical impossibilities. Likewise, nested loops also have a tendency to slow things down (sometimes). The stopping condition ($\lfloor\sqrt{\texttt{numm}}\rfloor$) is also a bit high. It could be programmed to have both loops quit as soon as `numm` reaches 1. The added branches would add complexity, but in terms of time saved in checking extraneous values, it could be useful.

The C implementation of this algorithm, although it appears to be slow, actually finishes rather quickly. The linux `time` command shows that the program takes roughly 0.2 seconds of real time to finish. The Octave implementation is much worse, taking roughly 6.7491 seconds (over a 50-run average) to calculate the same value. Although this discrepancy is curious, the main lesson in its existence shows the value of compiled routines versus interpreted for speed purposes.

For those who are curious but lazy, the greatest prime factor of 600851475143 is 6857.

Problem 4