CS 136 Fall 2008
Mark Giesbrecht

Assignment 4
Due Sunday, October 26, 2008 at 4pm.

All coding questions are to be implemented in the C language. The prototypes of all functions which are required in the questions should be stored in an appropriately named `.h` file, which you should `#include` into your `.c` file of solutions, and into your `.c` file of drivers.

Questions with non-programming answers should be neatly written and deposited into the CS136 assignment box outside of MC4065. Please use a cover sheet. The answers do not need to be typeset.

For each question, we will assume you have added the lines

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

1. Arithmetic modulo a prime p is at the heart of much of modern cryptography. In this question you will implement some basic functions for working in $\mathbb{Z}_p$. In this assignment we will assume that $0 \leq p < 2^{15}$ which will avoid numerical overflows in what follows.

   (a) Write a function with prototype `bool isprime(int p);` which returns `true` if `p` is prime, and `false` otherwise. Do this by trial division: divide by all numbers such that if `p` is *not* prime, then one of these divisions must have no remainder. Be clever about it; don't do more divisions than you have to!

   For parts (b) through (e) assume that p is prime.

   (b) Write functions

   - `int addp(int a, int b, int p);` ➡ $(a+b) \operatorname{rem} p$.
   - `int subp(int a, int b, int p);` ➡ $(a-b) \operatorname{rem} p$.
   - `int mulp(int a, int b, int p);` ➡ $(a*b) \operatorname{rem} p$.

   (c) Write the function `int powp(int a, int e, int p)`. Your function should produce $a^e \bmod p$. Use the repeated-squaring method described in tutorial. Your function should work when e is both positive and negative (see the 1d below).

   (d) Write the function `int invp(int a, int p);` ➡ $(1/a) \operatorname{rem} p$.
   Hint: you may use "Fermat's Little Theorem" which says $a^{p-1} \operatorname{rem} p = 1$ for all $a \not\equiv 0 \bmod p$.

   (e) The discrete logarithm of a with respect to "base" b modulo p is the smallest $k > 0$ such that $b^k \equiv a \bmod p$. Write a function with prototype `int dlogp(int a, int b, int p);` which returns the discrete logarithm of `a` with respect to base `b` modulo `p`. If no such `k` exists, then return 0.

   Place all your functions in a file called `a4q1-modp.c` and all the headers in `a4q1-modp.h`. You should also provide a file a4q1-modp-driver.c which contains a `main` function and provides appropriate tests through assertions.

2. We saw in class how to read an `int` from standard input using `scanf`. In fact, `scanf` also returns an `int`, which returns the number of data items read, or the special value `EOF` when the end of file is reached. Write a function `sumThree`, with prototype

   ```
   int sumThree();
   ```

   which reads `int`s from standard input until the end of file. It then returns the sum of the largest three numbers in the file.

   Place your function `sumThree` in a file called `a4q2-sumThree.c`, and a header with the prototype for `sumThree` in `a4q2-sumThree.h`. Also provide a file `a4q2-sumThree-driver.c` with a `main` function which provides appropriate tests.

3. Using the definition of O-notation, prove the following:

   (a) $3n^2 + n \log n$ is $O(n^2)$.

   (b) $30000n^2$ is $O(n^2 \log n)$.

   (c) $(n^2 - n)^2$ is not $O(n^3)$.

   (d) $n \log_2 n$ is *not* $O(n)$.

   You may use, without proof, the fact that $1 < \log_2(n) < n$ for $n \geq 3$. Remember to use the formal definition of O-notion.

4. An interesting sequence is defined as $J_0 = 1$, $J_1 = 2$ and $J_i = 2J_{i-2} - J_{i-1}$ for $i \geq 2$. The first few numbers in the sequence are $1, 2, 0, 4, -4, 12, -20, 44, -84, \ldots$

   (a) Consider the following simple Scheme algorithm to compute J_n =(*seq n*):

   ```
   (define (seq n)
        (cond [(= n 0) 1]
              [(= n 1) 2]
              [else (- (* 2 (seq (- n 2))) (seq (- n 1)))]))
   ```

   Let $T(n)$ be the total number of multiplications used by the preceding algorithm to compute (*seq n*). Prove by induction on n that

   $$T(n) \geq \theta^{n-1} - 1$$

   where $\theta = (1 + \sqrt{5})/2$ (the so called "golden ratio").

   (b) Describe an algorithm which computes J_n with cost $O(n)$. Indicate why it has the cost $O(n)$. Do not use the formula from part (c) below. You should give the algorithm in Scheme or C, but you do not need to implement it.

   (c) Prove that $J_n = 4/3 - (-2)^n/3$ by induction on n.

5. In CS 135 (and in a CS 136 tutorial) you saw how to compute $a^e \bmod n$ for positive integers a, e and n using two different methods. This is an important computation in many cryptography systems.

   The first method simply multiplied repeatedly:

   ```
   (define (mod-exp1 a e n)
   ```

```
(cond
  [(zero? e) 1]
  [else (modulo (* (mod-exp1 a (sub1 e) n) a) n)]))
```

Assuming that $m \in \{0, \dots, n-1\}$ and $e \in \{0, \dots, n-1\}$, let $T_1(e)$ be the number of *multiplications* needed to compute (*mod-exp1 a e n*) on any legal input. Give an explicit formula for $T_1(e)$.

The second function was more clever and is sometimes referred to as "repeated squaring" or "square and multiply":

```
(define (mod-exp2 a e n)
  (cond
    [(zero? e) 1]
    [(even? e) (modulo (mod-exp2 (sqr a) (quotient e 2) n) n)]
    [else (modulo ( * (mod-exp2 (sqr a) (quotient e 2) n) a) n)]))
```

Let $T_2(e)$ be the number of *modular reductions* (i.e., uses of the function *modulo*) needed to compute (*mod-exp2 a e n*). Describe $T_2(e)$ by a recurrence relation, indicating what the different cases are. Give an estimate for $T_2(e)$ in O-notation.

Hint. Assume that $2^{\ell-1} \le e < 2^{\ell}$. What is the cost of the algorithm in terms of ℓ? What is the relationship between ℓ and n?

6. **Bonus.** Prove the following statement by induction on n: if $T(n)$ has the property that $T(0)$ is $O(1)$ and $T(n) \le 3T(\lfloor n/3 \rfloor) + f(n)$, where $f(n)$ is a function from $\mathbb{N} \to \mathbb{N}$ which is $O(n)$, then $T(n)$ is $O(n \log n)$. Here $\lfloor n/3 \rfloor$ is the *floor* of $n/3$, the largest integer which is less than or equal to $n/3$ (i.e., $n/3$ rounded down to the nearest integer).