Problem Set 5 Solutions

Reading: State machine handout.
Remember that correctness involves termination plus the right answer!

Problem 1. Read the description of the RSA cryptosystem from the textbook, pp 146-148.

(a) Given primes $p = 173$ and $q = 149$ generate encryption and decryption keys, e and d respectively, such that $e \cdot d = 1 \bmod (p-1)(q-1)$. For "security," make $e, d \geq 20$.

(b) Convert the word "MIT" to a number by first mapping each letter to its position in the alphabet, e.g. A = 0, B = 1, C = 2, ..., Z = 25, and then gathering the numbers (M, I, T) into a single number $26^2 * M + 26 * I + T$. Find the RSA-encrypted version of "MIT" and show that it decrypts properly.

(c) What happens when you apply the encrypt/decrypt operation to "YOU"? Why?

a. Since 3 does not divide either $p-1$ or $q-1$, any power of 3 is relatively prime to $(p-1)(q-1)$. Thus 81 ($= 3^4$) is a suitable choice for e. We then use Euclid's algorithm on $e = 81$ and $(p-1)(q-1) = 25456$ to get $-11 \cdot 25456 + 3457 \cdot 81 = 1$. So
$81 \cdot 3457 \equiv 1 \pmod{25456}$, allowing us to make d be 3457.

b. The word "MIT" converts to the number $26^2 \cdot 12 + 26 \cdot 8 + 19 = 8339$. Since $p \cdot q = 25777$, we encrypt 8339 to $(8339)^{81} \bmod 25777 = ((8339)^3 \bmod 25777)^{27} \bmod 25777 =$
$(25056)^{27} \bmod 25777 = ((25056)^3 \bmod 25777)^9 \bmod 25777 = (17996)^9 \bmod 25777 =$
$((17996)^3 \bmod 25777)^3 \bmod 25777 = (3697)^3 \bmod 25777 = 10083$.

We decrypt 10083 to $(10083)^{3457} \bmod 25777$. In binary, 3457 is represented as 110110000001. We can use the fast exponentiation algorithm presented in recitation to compute $(10083)^{3457} \bmod 25777$. The following are the values returned by the recursive calls (and the original call) listed in order from the base case to the original call. The cases where the argument was odd are followed by an asterisk (*).

10083*
2401
4680*
17727
24899
14414*
776
13977*

18423
1170
2719
20739
16876
15080
1706
8339*

The last value (the one returned) is the same as the number we encrypted, so the value decrypted properly.

c. The value does not decrypt properly, because "YOU" corresponds to the number $16608 = 173 \cdot 96$. Since this number is not relatively prime to p, the correctness argument for the code does not apply to this number. You should check the argument to see where it breaks down.

Problem 2. Consider the following algorithm PolyEval, which takes a list of arguments $a_0, \ldots, a_n$ and an argument x, and is supposed to compute the value of the polynomial

$$\sum_{i=0}^{n} a_i x^i.$$

Prove that the algorithm is correct.

PolyEval($a_0, \ldots, a_n, x$)

```
s = 0
for i = n downto 0
    s = x * s + a_i
return s
```

Lemma 0.1 *After the kth pass through the for loop, the value of s is*

$$\sum_{j=n+1-k}^{n} a_j x^{j+k-1-n}$$

Proof. We use induction on k.

Base Case: For $k = 1$, there has been exactly one pass through the loop. At the beginning of the loop, the value of s was 0. After the loop body is performed, the value of s is $x * 0 + a_n = a_n x^0$, since the value of i during the pass was n.

Inductive Step: By the inductive hypothesis, after the kth pass through the for loop,

$$\sum_{j=n+1-k}^{n} a_j x^{j+k-1-n}$$

During the $(k+1)$st pass, the value of i is $n-k$, so s gets assigned the value

$$x \cdot \sum_{j=n+1-k}^{n} a_j x^{j+k-1-n} + a_{n-k}$$

$$= \sum_{j=n+1-k}^{n} a_j x^{j+(k+1)-1-n} + a_{n+1-(k+1)} x^{n+1-(k+1)+(k+1)-1-n}$$

$$= \sum_{j=n+1-(k+1)}^{n} a_j x^{j+(k+1)-1-n}$$

■

By the lemma, the value of s after the $(n+1)$st (and final) pass through the loop is

$$\sum_{j=n+1-(n+1)}^{n} a_j x^{j+(n+1)-1-n} = \sum_{i=0}^{n} a_i x^i$$

which is exactly the value we wanted computed. Since this is the value returned by the algorithm, it gets the right answer.

Note that since we are allowed to presume that the for loop terminates, proving termination becomes vacuous.

Problem 3. Write a (pseudocode) program that when given a natural number n returns the binary representation of n. (the output of the program must be a string over the alphabet $\{0,1\}$). Use standard primitive operations. Prove that your program is correct.

Procedure Bin(d)

$c := d$
$i := 0$
while $(c/2 > 0)$ {

$\quad b[i] := c \bmod 2$

$\quad c := c/2$

$\quad i := i + 1$

}
$b[i] = a$

The above while loop terminates because any number d can be divided by 2 and give nonzero results at most $\lfloor \log_2(d) \rfloor$ times. Now we prove that the program is correct.

Let d have the representation $d = \sum_{i=0}^{\infty} a_i 2^i$. Note that $d/2^k = \sum_{i=k}^{\infty} a_i 2^{i-k}$.

Theorem 0.2 *When $i = k$ we have the following invariants*

$b[k] = a_k$

$c = d/2^{k+1}$

Proof.

BASE CASE: When $i = 0$, $b[0] = d \bmod 2 = a_0$, and $c = d/2 = d/2^1$.

INDUCTIVE STEP: Suppose $i = k + 1$. Then by the inductive hypothesis, $c = d/2^{k+1}$, so we have $b[k+1] := d/2^{k+1} \bmod 2 = a_{k+1}$ and $c := c/2 = d/(2^{k+1} \cdot 2) = d/2^{k+2}$.

So the algorithm terminates in $\lfloor \log_2(d) \rfloor + 1$ steps with the ith order bit in $b[i]$.

■

Problem 4. Consider the following algorithm to find the greatest common divisor of a collection of numbers $a_1, \ldots, a_n$. It uses algorithm Euclid for finding the GCD of two numbers. Prove it is correct (you might want to prove a lemma about GCD(a,GCD($b_1, \ldots, b_k$))).

MultiEuclid($a_1, \ldots, a_n$)

```
r = a_1
for i = 2 to n
    r = Euclid(r, a_i)
return r
```

Lemma 0.3 *For any positive integers $b_1, \ldots, b_k$ and any common divisor d of $b_1, \ldots, b_k$, $d \mid GCD(b_1, \ldots, b_k)$.*

Proof. We provide a high-level argument. You should verify that you could make our argument into a more careful one.

From what we have shown about prime factorizations, one positive integer divides a second positive integer if and only if the prime factorization of the first number is contained in the prime factorization of the second. Thus a number is a common divisor of $b_1, \ldots, b_k$ if and only if its prime factorization is contained in the common part of their prime factorizations. It follows that the greatest common divisor of $b_1, \ldots, b_k$ is the number whose prime factorization *is* the common part of the prime factorizations of $b_1, \ldots, b_k$. It also follows that any common divisor d of $b_1, \ldots, b_k$ divides GCD($b_1, \ldots, b_k$), because d's prime factorization is contained in that of GCD($b_1, \ldots, b_k$). ■

Lemma 0.4 *For any positive integers $a, b_1, \ldots, b_k$, GCD($a, b_1, \ldots, b_k$) = GCD(a,GCD($b_1, \ldots, b_k$)).*

Proof. Let $g = \text{GCD}(a, b_1, \ldots, b_k)$ and $c = \text{GCD}(a, \text{GCD}(b_1, \ldots, b_k))$. Since $c \mid a$ and $c \mid \text{GCD}(b_1, \ldots, b_k)$, c is a common divisor of $a, b_1, \ldots, b_k$. Thus $c \leq g$, because g is the *greatest* common divisor of $a, b_1, \ldots, b_k$. Since g is a common divisor of $a, b_1, \ldots, b_k$, we can apply the above lemma to get that g is a common divisor of a and $\text{GCD}(b_1, \ldots, b_k)$. Since c is the greatest such number, $g \leq c$. Since $c \leq g$ and $g \leq c$, $g = c$. ∎

Lemma 0.5 *After the kth pass through the for loop, the value of r is GCD($a_1, \ldots, a_{k+1}$).*

Proof. We use induction on k.

Base Case: k = 1. Clearly, after the first pass through the for loop the value of r is $\text{GCD}(a_1, a_2)$.

Inductive Step: By the inductive hypothesis, after the kth pass through the for loop, the value of r is $\text{GCD}(a_1, \ldots, a_{k+1})$. During the $(k+1)$st pass through the for loop, the value of i is $(k+2)$. So after the $(k+1)$st pass through the for loop, the value of r is $\text{GCD}(\text{GCD}(a_1, \ldots, a_{k+1}), a_{k+2})$. By the above lemma, this value is $\text{GCD}(a_1, \ldots, a_{k+2})$. ∎

Thus, the value of r after the $(n-1)$st pass (the last one) through the for loop is $\text{GCD}(a_1, \ldots, a_n)$. Since this is the value returned by the algorithm and is the one we want, the program gets the right answer. Note that again the proof of correctness is vacuous since we are allowed to assume that for loops terminate.

Problem 5. The following code estimates the value of $\frac{1}{1-x/2}$ to within an additive error of $1/1024$ for $0 < x < 1$. Prove that it is correct.

```
estimate(x)
{
  Fprev = 0;
  Fnext = 1;
  while (Fnext - Fprev > 1/1024) {
    Fprev = Fnext;
    Fnext = (Fprev * x / 2) +1;
  }
  return(Fnext);
}
```

Theorem 0.6 *In the nth step of the algorithm we have:*
$Fprev = \sum_{i=0}^{n-1} (\frac{x}{2})^i$
$Fnext = \sum_{i=0}^{n} (\frac{x}{2})^i$

Proof. BASE CASE: In the 0th step Fprev $= 0 = \sum_{i=0}^{-1}\left(\frac{x}{2}\right)^i$, Fnext $= 1 = \sum_{i=0}^{0}\left(\frac{x}{2}\right)^i$.

INDUCTIVE STEP: At the $(n+1)$th step we have:
Fprev := Fnext $= = \sum_{i=0}^{n}\left(\frac{x}{2}\right)^i$
Fnext := (Fprev$*x/2) + 1 = x\sum_{i=0}^{n}\left(\frac{x}{2}\right)^i/2 + 1 = \sum_{i=0}^{n+1}\left(\frac{x}{2}\right)^i$.

∎

From the above theorem, termination follows trivialy because Fnext - Fprev $= \left(\frac{x}{2}\right)^n$ becomes less than $1/1024$ for sufficiently high n.

To see that the algorithm computes $\frac{1}{1-x/2}$ within an error of $1/1024$, note that
$\frac{1}{1-x/2} = \sum_{i=0}^{\infty}\left(\frac{x}{2}\right)^i = \sum_{i=0}^{n}\left(\frac{x}{2}\right)^i + \frac{1-(x/2)^{n+1}}{1-x/2}$
But, $\frac{1-(x/2)^{n+1}}{1-x/2} = \frac{(x/2)^n}{2/x-1} < (x/2)^n$.

Problem 6. The mode of a list is an element appearing as frequently in the list as any other. Let $A[i..j]$ denote the sublist of A with indices between i and j, inclusive. Consider the following algorithm.

```
Mode(A)
  if (size[A] = 1)
    return(A[1])
  if (A[1] = Mode(A[2..size[A]]))
    return(A[1])
  else
    return(Mode(A[2..(size[A])]))
```

What is the fallacy in the following argument that Mode returns the mode of A? Explain the fallacy, and give a counterexample showing the algorithm fails.

Proof by induction on the length of the input. The base case is trivial. So assume the hypothesis true for inputs of length n. Consider an input of length $n+1$. Then by induction, each of the two recursive calls properly returns the mode of its argument. Thus, if $A[1]$ is the mode, the if test succeeds, and the function returns the correct value. If $A[1]$ is not the mode, then clearly removing $A[1]$ doesn't change the mode, so the second recursive call succeeds and returns the correct value.

Counterexample: {2,2,3 } (the program returns 3).

The fallacy in the proof is the implicit assumption that the mode of a list is unique. When a sublist of two distinct elements is passed in the program, it arbitrarily returns one of them, and this may not end up being a mode of the whole list.

Problem 7. A *search tree* is either:

- An *empty tree* with no nodes
- A *root node* with *key* k, and
  - a *left subtree,* all of whose nodes have keys less than k
  - a *right subtree,* all of whose nodes have keys exceeding k.

The following fallacious algorithm Find, when called with key k and (a, b) search tree T, should print "found it" if and only if k is the key of a node in the search tree.

```
Find(k, T)

    c =key(root-node(T))
    if (k = c)
        print "found it"
    else if (k < c)
        Find(k,left-subtree(T))
    else
        Find(k,right-subtree(T))
```

(a) (3 points) What is wrong with the above algorithm? Add a starting if-test that will fix the problem.

(b) (7 points) Prove that your (fixed) algorithm does what it should.

Fixed algorithm:

```
Find(k, T)

    if (empty) print "Key not in the tree"
    c =key(root-node(T))
    if (k = c)
        print "found it"
    else if (k < c)
        Find(k,left-subtree(T))
    else
        Find(k,right-subtree(T))
```

Theorem 0.7 *For a tree of length n, the algorithm terminates in at most n steps. Furthermore, if the key is in the tree it prints the message "found it", otherwise it prints the message "Key not in the tree".*

Proof.

BASE CASE: If the tree is empty the algorithm terminates in 0 steps with the message "Key not in the tree".

INDUCTIVE STEP: Given a tree of length $n+1$, the algorithm checks if the key is in the root. If it is, it prints "found it", and terminates in 1 step. Otherwise, it calls the algorithm for a subtree of length n and, by the inductive hypothesis, the algorithm terminates in at most $1+n$ steps. Furthermore, if the key is in the tree, the algorithm looks at the correct subtree because if $k < c$, it calls Find on the left subtree which by definition contains keys less than c, while if $k > c$, it calls Find on the right subtree which contains keys greater than c.

■

Problem 8. Consider a coin exchanger described by the following state machine.

The state set Q consists of values for state components:

$$\begin{array}{l} \mathit{total} \in \mathbb{N} \\ \mathit{total}' \in \mathbb{N} \\ \mathit{cents} \in \mathbb{N} \\ \mathit{dimes} \in \mathbb{N} \\ \mathit{quarters} \in \mathbb{N}. \end{array}$$

Here *total* is an initial amount of money (in cents) given as input to the machine. All other variables are initially set to 0. When the machine terminates (no more transitions are possible) the variables *cents*, *dimes* and *quarters* should be the correct change corresponding to the input *total*. total' is an auxiliary variable.

The transition function δ is defined by the following rules.

give-quarter
Precondition:
$total - total' \geq 25$
Effect:
$total' := total' + 25$
$quarters := quarters + 1$

give-dime
Precondition:
$25 > total - total' \geq 10$
Effect:
$total' := total' + 10$
$dimes := dimes + 1$

give-cent
Precondition:
$10 > total - total' > 0$
Effect:
$total' := total' + 1$
$cents := cents + 1$

Prove using the invariant method that the algorithm is correct, that is, when the machine terminates the total amount of money described by the variables $cents$, $dimes$, $quarters$ is exactly $total$.

Our first invariant is that for every reachable state $total - total' \geq 0$. In all initial states, $total$ is nonnegative and $total'$ is zero. So the invariant clearly holds for all initial states. Furthermore, each transition leaves $total$ alone and adds an amount to $total'$ that is less than or equal to $total - total'$, thus preserving the invariant.

Our second invariant is that for every reachable state $total' = 25 \cdot quarters + 10 \cdot dimes + cents$. In all initial states, all of the variables in this equation are zero, so equality holds. It is easy to check that each transition adds the same amount to both sides, preserving equality.

The machine terminates when no transitions are possible. This means that $total - total' \leq 0$. By our first invariant, $total - total' \geq 0$, so $total - total' = 0$. That is, $total = total'$. By our second invariant, $total' = 25 \cdot quarters + 10 \cdot dimes + cents$. So $total = 25 \cdot quarters + 10 \cdot dimes + cents$. This equality is what we wanted to show.

Problem 9. (optional)

Consider the following procedure Bump, which operates on an infinite array of bits $b[0], b[1], b[2], \ldots$ that are all initially 0.

Bump(k)

```
if b[k] = 0 then
    b[k] = 1
else
    b[k] = 0
    Bump(k + 1)
end if
```

What will the $b[i]$ contain if Bump(0) is called n times? Prove it.

The $b[i]$ will contain the binary representation of n.

First, we prove a lemma about calls made when the bits $b[i]$ are not necessarily in their initial configuration.

Lemma 0.8 *If there is a greatest index i such that $b[i] = 1$, then for $0 \leq n \leq i$, calling Bump$(i-n)$ increases*

$$\sum_j b[j]2^j$$

by 2^{i-n}.

Proof. We use induction on n.

Base Case: $n = 0$. Then calling Bump$(i-n)$ changes $b[i]$ from a 1 to a 0 and calls Bump$(i+1)$, which changes $b[i+1]$ from a 0 to a 1. The first change causes

$$\sum_j b[j]2^j$$

to go down by 2^i. The second change causes this sum to go up by 2^{i+1}. So the net effect of the call is to increase

$$\sum_j b[j]2^j$$

by 2^i.

Inductive Step:

Case 1: $b[i-(n+1)] = 0$. Then calling Bump$(i-(n+1))$ changes $b[i-(n+1)]$ from a 0 to a 1, which increases

$$\sum_j b[j]2^j$$

by $2^{i-(n+1)}$.

Case 2: $b[i-(n+1)] = 1$. Then calling Bump$(i-(n+1))$ changes $b[i-(n+1)]$ from a 1 to a 0 and calls Bump$(i-n)$. The flip of $b[i-(n+1)]$ causes

$$\sum_j b[j]2^j$$

to go down by $2^{i-(n+1)}$. By the inductive hypothesis, calling Bump$(i-n)$ increases

$$\sum_j b[j]2^j$$

by 2^{i-n}. So the net effect of the call to Bump$(i-(n+1))$ is to increase

$$\sum_j b[j]2^j$$

by $2^{i-(n+1)}$. ■

We now go back to considering what happens if we start with the initial configuration of the bits $b[i]$.

Theorem 0.9 *If Bump(0) is called n times, the $b[i]$ will contain the binary representation of n (with an infinite number of extra zeroes).*

Proof. We use induction on n.

Base Case: n = 0. Then the bits are left in their initial configuration, which is the binary representation of 0.

Base Case: n = 1. Then $b[0]$ is flipped from a 0 to a 1 by the single call to Bump(0). This leaves the bits containing the binary representation of 1.

Inductive Step: By the inductive hypothesis, if Bump(0) is called n times, the $b[i]$ will contain the binary representation of n (with an infinite number of extra zeroes). If we make an $(n+1)$st call to Bump(0), then by the lemma

$$\sum_j b[j]2^j$$

increases by $2^0 = 1$. The value of the sum was originally n, so it ends up being $n+1$. The only way the sum can be $n+1$ is if the bits contain the binary representation of $n+1$. ■