CS 3510 - Spring 2009
Homework 1
Due: February 11

You must hand in this homework. Please work alone on this assignment. Do not use a calculator (except to check, if you want), and please show all of your work.

1. Run the extended Euclidean algorithm for $a = 697$, $b = 969$.

$$\begin{aligned}
969 &= 1 * 697 \\
697 &= 2 * 272 + 153 \\
272 &= 1 * 153 + 119 \\
153 &= 1 * 119 + 34 \\
119 &= 34 * 3 + 17 \\
34 &= 17 * 2 + 0 \\
17 &= 17 - 0
\end{aligned}$$

So gcd(969, 697) = 17. Now we work backwards to determine the values for x and y for the extended algorithm.

$$\begin{aligned}
17 &= 17 - 0 \\
&= 17 - (34 - 17 * 2) &&= -1 * 34 + 3 * 17 \\
&= -1 * 34 + 3 * (119 - 34 * 3) &&= -10 * 34 + 3 * 119 \\
&= 3 * 119 - 10 * (153 - 119 * 1) &&= 13 * 119 - 10 * 153 \\
&= 13 * (272 - 1 * 153) - 10 * 153 &&= 13 * 272 - 23 * 153 \\
&= 13 * 272 - 23 * (697 - 2 * 272) &&= -23 * 697 + 59 * 272 \\
&= -23 * 697 + 59(969 - 1 * 697) &&= 59 * 969 - 82 * 697
\end{aligned}$$

So

$$d = 17, x = -82, y = 59.$$

2. Prove or disprove: If a has an inverse modulo b, then b has an inverse modulo a.

   The statement is true.

   Proof. If a has an inverse mod b then there exists x such that $ax = 1 \bmod b$. It follows then that the gcd of a and b is 1. Therefore there exists y and z such that $ay + bz = 1$. Taking this mod$a$ gives us $bz = 1 \bmod a$. So $b$has an inverse mod$a$. □

3. Calculate $2^{125} \bmod 127$ using any method you choose. (Hint: 127 is prime).

$$\begin{aligned} 2^{125} \pmod{127} &\equiv 2^{7*17} * 2^6 \pmod{127} \\ &\equiv (2^7 \bmod 127)^{17} * 2^6 \pmod{127} \\ &\equiv 1^{17} * 2^6 \pmod{127} \\ &\equiv 2^6 \pmod{127} \\ &\equiv 64 \pmod{127} \end{aligned}$$

4. Consider an RSA key set with $p = 17, q = 23, N = 391$, and $e = 3$ (as in Figure 1.9). What value of d should be used for the secret key? What is the encryption of the message $M = 41$?

   To find d we use the extended gcd to find x and y s.t. $e * x + (p-1)(q-1) * y = 1$. Note that x is d, the inverse of $emod(p-1)(q-1)$.

$$\begin{aligned} (p-1)(q-1) &= 16 * 22 = 352 \\ 352 &= 117 * 3 + 1 \\ 3 &= 1 * 3 + 0 \end{aligned}$$

   So gcd(352, 3) = 17. Now we work backwards to determine the values for x and y for the extended algorithm.

$$\begin{aligned} 1 &= 1 - 0 \\ &= 1 - (3 - 1 * 3) = -1 * 3 + 4 * 1 \\ &= -1 * 3 + 4 * (352 - 117 * 3) = 4 * 352 - (4 * 117 + 1) * 3 \end{aligned}$$

   Taking the last equation mod 352, we find that

$$d \equiv -469 \pmod{352} \equiv 235 \pmod{352}.$$

Now we encode message M.

$$\begin{aligned} y &= M^e \bmod N \\ &= 41^3 \bmod 391 \\ &= 68921 \bmod 391 \\ &= 105 \end{aligned}$$

5. In an RSA cryptosystem, $p = 7$ and $q = 11$ (as in Figure 1.9). Find appropriate exponents d and e.

   We should choose an e such that it is relatively prime to $(p-1)(q-1)$. In this case, $p = 7, q = 11$, and so e should be relatively prime to 60. Since 60 has factors 2, 3, 4, 5, and 6, it suffices to choose 7. Now that we have e, we run the extended Euclidean algorithm in the same manner as the previous problem.
   Then $a = 7, b = 60$, and we calculate

$$60 = 8 * 7 + 4$$

$$7 = 1 * 4 + 3$$

$$4 = 1 * 3 + 1$$

$$3 = 3 * 1 + 0$$

   So gcd(60, 7) = 1. Now we work backwards to determine the values for x and y for the extended algorithm.

$$\begin{aligned} 1 &= 1 - 0 \\ &= 1 - (3 - 3 * 1) = -1 * 3 + 4 * 1 \\ &= -1 * 3 + 4 * (4 - 1 * 3) = 4 * 4 - 5 * 3 \\ &= 4 * 4 - 5 * (7 - 1 * 4) = 5 * 4 - 5 * 7 \\ &= 5(60 - 8 * 7) - 5 * 7 = 5 * 60 - 45 * 7 \\ d &= -45 \end{aligned}$$

   .

6. **Recurrences**
   Solve the following recurrences. You can use the Master Theorem, where applicable. Big-O notation is fine.

   (a) $T(n) = 2T(n/3) + n$
   We use the Master theorem. Then $a = 2$, $b = 3$ and $d = 1$ and we find that we are in case 1 because $d > \log_3 2$. So the running time is $T(n) = O(n)$

(b) $T(n) = 2T(n/3) + 1$
We use the Master theorem. Then $a = 2$, $b = 3$ and $d = 0$ and we find that we are in case 3 because $d < \log_3 2$. So the running time is $T(n) = O(n^{\log_3 2})$.

(c) $T(n) = 9T(n/2) + n^2$
We use the Master theorem. Then $a = 9$, $b = 2$ and $d = 2$ and we find that we are in case 3 because $d < \log_2 9$. So the running time is $T(n) = O(n^{\log_2 9})$.

(d) $T(n) = T(n/4) + 3$
We use the Master theorem. Then $a = 1$, $b = 4$ and $d = 0$ and we find that we are in case 2 because $d = \log_4 1 = 0$. So the running time is $T(n) = O(\log n)$.

(e) $T(n) = 3T(n/2) + 1$
We use the Master theorem. Then $a = 3$, $b = 2$ and $d = 0$ and we find that we are in case 3 because $d < \log_2 3$. So the running time is $T(n) = O(n^{\log_2 3})$.

(f) $T(n) = 2T(n-1) + 1$

Let $T(1) = c$. We want to sub in values for T(n-1), T(n-2), and so on using the original equation and see if a pattern develops.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 4T(n-2) + 2 + 1 \\ &= 4(2T(n-3) + 1) + 4 + 2 + 1 \\ &= 8T(n-3) + 4 + 2 + 1 \\ &\dots \end{aligned}$$

We notice that for each level deeper we go, the coefficient in front of $T(n-i)$ is increasing by two, at the ith level. So we can create a general form that determines $T(n)$ at level i. $T(n) = 2^i * T(n-i) + k$, where $k = 2^{i-1} + 2^{i-2} + ... + 2^0$. Since the added k is always of a lower exponent than the coefficient, we know the coefficient will dominate. So in the overall case, we know that this iteration can only occur $n-1$ times otherwise you run out of elements to operate on.

$$= 2^{n-1}T(1) + k$$

Also, once we get to this level there should only be a constant amount of work remaining.

$$T(n) = 2^{n-1} * c + k$$

So the running time is $T(n) = O(2^n)$.

7. **Stooge Sort**
Professor Randall thinks she has a new sorting algorithm. Here is the proposed algorithm. The input is a list $A[1 \ldots n]$ of n numbers, where n is a power of 2.

```
RandallSort(A)
1  if length(A) = 1,
2    then return A
3  RandallSort(A[1 ... n/2])
4  RandallSort(A[n/2 + 1 ... n])
5  for i = 1 → n/2
6    if A[i] > A[i + n/2],
7      then Swap(A[i], A[i + n/2])
8  RandallSort(A[1 ... n/2])
9  RandallSort(A[n/2 + 1, n])
```

```
Swap(k, l)
1  temp ← A(l)
2  A[l] ← A[k]
3  A[k] ← temp
```

(a) Analyze the running time of RandallSort by stating and solving the appropriate recurrence.
$T(n) = 4T(n/2) + O(n)$
We use the Master theorem. Then $a = 4$, $b = 2$ and $d = 1$ and we find we are in case 3 because $d < \log_2 4 = 2$. So the running time is $T(n) = O(n^2)$.

(b) Does the algorithm sort correctly? If yes, argue why. If no, give an example for which the above algorithm does not sort correctly.

No. Let $A = [1, 2, 4, 5, 3, 6, 7, 8]$. We divide the array into the sub-arrays [1, 2, 3, 5] and [3, 6, 7, 8]. These are already sorted, so let's ignore the sorting step and go to the swapping stage. For all values of i, $A[i] < A[i + n/2]$ so no swapping occurs. The two halves are again independently sorted yielding the same array that we had initially [1, 2, 4, 5, 3, 6, 7, 8], which is not in sorted order.

8. **Fixed Point**
Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(logn)$.

```
function(start, end, array)
1   if start == end
2     if A[start] == start
3       return start
4     else
5       return -1
6   i = floor ((end - start)/2)
7   if A[i] > i
8     return function(start, i - 1, array)
9   else if A[i] < i
10     return function(i+1, end, array)
11  else
12     return i
```

The idea behind this algorithm is that we look at the value of the midpoint. If it is larger than the value of its position then all values to the right of it will be larger than their position. This is because both i and the value at i must increase by at least 1 for the list to be sorted in ascending order and have no repeated values. This logic also applies if the value at the midpoint is less than i. All values to the left of the midpoint will be less than their value for i. So this results in three cases, one where the right half can be discarded, one where the left can be discarded and one where the sought point is found immediately. We are effectively allowed to cut the problem size in half for every recursive call. The recurrence for this algorithm is $T(n) = T(n/2) + c$. We use the master theorem to solve it. Then $a = 1, b = 2$, and $d = 0$ and we find we are in case 2 because $\log_a 1 = 0$. So the running time is $O(\log n)$.