CSCI 189 Assignment 6 Problem Set

This assignment provides an opportunity to learn about functions and their importance in Haskell programming and in RSA cryptography.

Summary of Concepts in Section 4.4

A *function* f is a mapping between two sets S and T (denoted $f : S \rightarrow T$); a subset of $S \times T$ in which each member of S appears exactly once as the first component of an ordered pair. S is the *domain* of f and T is its *codomain*. If the pair (s, t) belongs to f then we write $t = f(s)$, and t is called the *image* of s under f. So a function is a special kind of binary relation.

A function can have more than one variable in its domain. The function $f : S_1 \times S_2 \times \ldots \times S_n \rightarrow T$ associates with each n-tuple in its domain a unique element in T. That is, we write $t = f(s_1, s_2, \ldots, s_n)$.

For example, the *floor* function computes the greatest integer less than or equal to a real number. It is denoted by $\lfloor x \rfloor$. It is a mapping $R \rightarrow Z$. In Haskell, it is written as `floor x`. For instance, $\lfloor 3.5 \rfloor = 3$ and $\lfloor -3.5 \rfloor = -4$.

For another example, the length of the *hypotenuse* of a right triangle is a function of the lengths of its other two sides x and y, given by the familiar formula $hypotenuse = \sqrt{x^2 + y^2}$. This function is a mapping $R \times R \rightarrow R$. In Haskell, this function can be defined in the following way:

```
hypotenuse x y = sqrt(x^2 + y^2)
```

As a third example, the reverse of a string is a function that simply creates its mirror image. For example, `reverse("hello") = "olleh"`. in Haskell, this is written `reverse "hello"`. Finally, the concatenation of two strings, denoted in Haskell by the operator `++`, creates a new string by placing the two side by side. For example, the Haskell expression `"hello " ++ "world"` creates the new string `"hello world"`.

The *range* R of a function is the set of image values $R \subseteq T$. A function is *onto* or *surjective* if $R = T$. A function is *one-to-one* or *injective* if no member of T is the image of two distinct elements of S. A function is *bijective* if it is one-to-one and onto. For example, the function $\lfloor x \rfloor$ is onto, since its range is the entire set Z. But it is not one-to-one since, for instance, $\lfloor 3.5 \rfloor = 3$ and $\lfloor 3.6 \rfloor = 3$. However, the function `reverse` is bijective (one-to-one and onto).

Let $f : S \rightarrow T$ and $g : T \rightarrow U$. Their *composition* $g \cdot f$ is $g(f(x))$. The composition of two bijections is a bijection. Moreover, if $f : S \rightarrow T$ and $g : T \rightarrow S$, then g is an *inverse* of f (denoted by f^{-1}) if $g(f(x)) = x$ for all $x \in S$. For example, the function `reverse` is an inverse of itself - e.g., `reverse(reverse("hello")) = "hello"`.

The set of all bijections of a set S onto itself are called the *permutations* of S. For example, there

are six permutations of a word with three distinct letters (like "HEY"). One of those bijections, `reverse("HEY")`, computes one such permutation. What are the other permutations of "HEY"?

Since we suspect that there are $n!$ permutations of an n-letter word, we hope that there is a general expression defining the number of different functions $f : S \to T$ that are bijections from S to T. It turns out that, if $|S| = m$ and $|T| = n$, this number is $\frac{n!}{(n-m)!}$. In the event that $n = m$ (e.g., when S and T are the same set), this number becomes $n!$, which provides a better basis for understanding the number of permutations for the word "HEY".

Two sets S and T are *equivalent* if there is a bijection $f : S \to T$. S and T are said to have the same *cardinality*.

Growth of Functions

Let f and g be two functions mapping nonnegative reals to nonnegative reals. Then f is the same *order of magnitude* as g if there exist n, c_0, and $c_1 > 0$ such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for all $n > n_0$. This is written $f = \Theta(g)$. Moreover, f is *big oh* of g, written $f = O(g)$, if there exist n_0 and $c > 0$ such that $f(x) \leq cg(x)$ for all $n > n_0$. So why is this important?

Growth of functions is important in computer science because it provides a mathematical basis for measuring the expected *speed* of a complex computer program. Computer programs are often called "algorithms," and the expected speed of an algorithm is called its *computational complexity*.

For example, suppose we have three algorithms A, A', and A" that do the same computation (compute the same function) but use three different strategies. Suppose the speed of A, A', and A" is $\Theta(n)$, $\Theta(n^2)$, and $\Theta(2^n)$ for different sizes of input, n. The table below shows the difference in speed for each of these algorithms for $n = 10$, 50, and 100.

Algorithm	n=10	n=50	n=100
A	.001 sec	.005 sec	.01 sec
A'	.01 sec	.25 sec	1 sec
A"	.1024 sec	3570 years	4×10^{16} centuries

Note how dramatically the speed of A" degrades. Problems whose solutions can't be found using polynomial time algorithms (i.e., algorithms like A and A') are called *intractable* problems.

Application: Cryptography

Crpytography provides a wonderful example of how certain functions and a knowledge of their computational complexity can be used to provide a means of secure message transmission across the Internet. (Of course, cryptography is an ancient art, and its modern uses provide only a glimpse of its rich heritage and impact on human history. For a fascinating glimpse of that rich history, try reading Dan Brown's book *The Da Vinci Code*.)

A *cryptosystem* is a strategy for encoding a message in a way that only the sender and the receiver, but not an eavesdropper, can read the message. A *public key* cryptosystem is one in which there are two keys - a public key and a secret key. The public key is held by all the potential senders of messages, and the secret key is known only to the receiver of those messages.

The *RSA public key cryptosystem* bases its integrity on the relative ease of finding large prime numbers and the difficulty (computational complexity) with which current algorithms can factor products of such numbers. Here is a summary of how it works (adapted from [Cormen et al., 2001]):

Let the set D be the domain of all messages M sent between Bob and Alice, and let Alice's public and secret keys, P and S, correspond to the functions P() and S(), which are bijections of D. Thus, P() and S() are permutations of D and can be efficiently computed, given P and S. Moreover, P() and S() are inverses of each other, so that any message $M = P(S(M)) = S(P(M))$. Here is how P and S are computed.

1. Select two large prime numbers p and q, with $p \neq q$.
2. Compute $n = pq$ and $m = (p-1)(q-1)$.
3. Find e such that $gcd(e, m) = 1$.
4. Compute d as the unique integer between 0 and m such that $e \cdot d \bmod m = 1$.
5. The public key P is the pair (e, n).
6. The secret key S is the pair (d, n).

The domain D for this scheme is the integers modulo n, and the functions P() and S() are computed as follows:

$P(M) = M^e \bmod n$, using the public key $P = (e, n)$.

$S(P(M)) = P(M)^d \bmod n$, using the secret key $S = (d, n)$.

The important point in this scheme is that the secret key $S = (d, n)$ is known only to Alice, and in particular the value of d is very difficult for others to compute. That is, for large values of n, the factoring of n is a very complex operation. In fact, current research suggests that by randomly selecting and multiplying two 512-bit primes, one can create a public key that cannot be "broken" in any feasible amount of time with current computing technology. In particular, the so-called "Pollard-Rho" algorithm for factoring a b-bit composite number n requires at most $2^{b/4}$ arithmetic operations. Looking back at the table for algorithm A" on the previous page, we see that for any value of $b > 50$ this factoring task is intractible with today's technology.

Let's explore how the RSA public key cryptosystem works, using small practical values for P and S. Suppose Bob wants to send the message "HEY" to Alice. He first encodes the message by translating it into a string of integers (let's use the standard ASCII coding scheme, where A=65, B=66, ...), so that M = the list [72,69,89]. Suppose Bob knows Alice's public key $P = (e, n) = (3, 319)$. Then

he can encrypt each of the letters in that message (using the function $P(72) = 72^3 \bmod 319 = 18$, etc.) and send the encrypted message P(M) = [18, 258, 298].

Since presumably only Alice knows her secret key $S = (d, n) = (187, 319)$, only she can decrypt Bob's message, (using the function $S(18) = 18^{187} \bmod 319 = 72$, etc.) recovering the original message M = [72, 69, 89].

You can try out these calculations and others by exercising the following functions in the accompanying `Asst6.lhs` Haskell file:

```
encrypt ms e n   - encrypt message M (= the list ms) using public key P = (e, n)
decrypt ps d n   - decrypt message P(M) (= the list ps) using secret key S = (d, n)
findSecretKey n  - discover the secret key belonging to the integer n = p*q
                   (use this function only with small values for n)
leastdivisor2 n  - computes the smallest divisor of n between 2 and n
leastdivisor m n - computes the smallest divisor of n between m and n
primes           - computes an infinite list of primes
factors n        - computes the prime factors of n
isPrime n        - tests whether or not n is prime
```

Finally, it is interesting to note that public key cryptosystems can also be used for verifying *digital signatures*. A *digital signature* is like a person's handwritten signature, in the sense that it is uniquely created by that person (ignoring the possibility of perfect forgeries) and others can recognize that fact. For instance, RSA cryptography can ensure that anyone in the world who knows Alice's public key will know that a message came from her and only her. That is, if Alice encodes a message M and sends it as $S(M)$ using her secret key, then anyone who knows her public key P can decode it by computing $P(S(M))$ and be assured that noone else could have sent that message.

Problems to be handed in

A. Section 4.4 (p 312), exercises 6bd, 8ghj, 14, 16, 20bc, 23cd, 29bc, 43a, 44a, 45bd, 53, 59.

B. After reading the lab tutorial `Asst6.lhs`, answer the following questions by defining and exercising appropriate Haskell functions:

1. Define a Haskell function "decypher" that decodes a message that is encoded using the Caesar cypher. For instance, the call `Asst6> decypher "Khoor Zruog!"` should return the result `"Hello World!"`.
2. Are "decypher" and "cypher2" inverses? What about "decypher" and "cypher"? Explain.
3. For Alice's public key $P = (3, 319)$, use an appropriate Haskell function to determine the integers p and q for which $n = pq$.
4. Can you easily compute Alice's secret key $S = (d, n)$? Explain.
5. Suppose Alice's public key were $P = (3, 106556839)$. What are the factors of this large number? Now can you use this information to easily determine Alice's secret key? Explain.