CS3114 Fall 2012 Homework Assignment 1
Sample Solutions

1. Here are some questions that test your working knowledge of how fast computers operate. Is disk drive access time normally measured in milliseconds (thousandths of a second) or microseconds (millionths of a second)? Does your RAM memory access a word in more or less than one microsecond? How many instructions can your CPU execute in one year if the machine is left running at full speed all the time? DO NOT use paper or a calculator to derive your answers.

Disk drive seek time is somewhere around 10 milliseconds or a little less. RAM memory requires around 10 nanoseconds – much less than a microsecond. Given that there are about 30 million seconds in a year, a machine capable of executing at one billion instructions per second would execute about 30 million billion ($3 * 10^{16}$) instructions in a year.

---

2. For each of the following pairs of functions, determine whether $f(n)$ is in $O(g(n))$, $g(n)$ is in $O(f(n))$, or $f(n)$ is $\Theta(g(n))$. (Read Section 3.4.5 of the book for help.)

(i) $f(n) = \sqrt{n}$, $g(n) = \log(n^2)$.

$f(n)$ is in $\Omega(g(n))$. Since n^{c_1} grows faster than $\log n^{c_2}$ for any constants c_1 and c_2, the ratio of $\sqrt{n}$ to $\log n^2$ approaches infinity as n approaches infinity.

(ii) $f(n) = \log(n^2)$, $g(n) = \log n$.

$f(n)$ is in $\Theta(g(n))$. Since $\log(n^2) = 2 \log n$, the limit goes to $1/2$ (a constant).

(iii) $f(n) = 2^n$, $g(n) = 10n^2$.

If we take the log of both sides, we can easily see that the limit of $\log f(n)/\log g(n)$ goes to infinity. Thus, $f(n)$ is in $\Omega(g(n))$.

(iv) $f(n) = 2^n$, $g(n) = 3^n$.

$f(n)$ is in $O(g(n))$. $3^n = 1.5^n 2^n$, and if we divide both sides by 2^n, we see that 1.5^n grows faster than 1.

---

3. Give the best *lower* bound that you can for the following code fragment, as a function of the initial value of n.

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
   else
     n = n / 2;
```

Do you think that the upper bound is likely to be the same as the answer you gave for the lower bound?

The best lower bound I know is $\Omega(\log n)$, since a value cannot be reduced more quickly than by repeated division by 2. There is no known upper bound, since it is unknown if this algorithm always terminates. But many values have costs that are much greater than $\log n$. So it looks like the upper bound is greater than the lower bound.

---

4. A typical array-based list implementation stores references to the list data elements. A typical linked list implementation stores in each link node a reference to the data element and a reference to the next link node. Determine the size of a pointer on your machine. Use this and the equation from Section 4.1.3 of the textbook to determine the breakeven point beyond which the array becomes more space efficient than the linked list.

Answer 1: The following holds whether you assume that your pointers are 4 bytes (32-bit machine) or 8 bytes (64-bit machine). The linked list stores 2 pointers/link node plus the data field for each of n nodes, while the array stores 1 pointer in every one of the D array slots plus the data field for the n slots with data. We can ignore the space for the data since this is the same for both representations. Thus, the break-even point comes when $P * D = 2P * n$, or when $n = D/2$. The array becomes more space efficient whenever it is more than half full.

---

5. A palindrome is a string that reads the same forwards as backwards. Using only a fixed number of stacks and queues, the stack and queue ADT functions, and a fixed number of `int` and `char` variables, write an algorithm to determine if a string is a palindrome. Assume that the string is read from an input stream one character at a time. The algorithm should output TRUE or FALSE as appropriate.

```
bool palin() {
  Stack<char> S;
  Queue<char> Q;

  while ((c = getc()) != ENDOFSTRING) {
    S.push(c);
    Q.enqueue(c);
  }
  while (!S.isEmpty()) {
    if (S.top() != Q.front()) return FALSE;
    char dum = S.pop();
    dum = Q.dequeue();
  }
  return TRUE;
}
```
