CSS 342 Big-oh Practice Problems Solution
*** These solutions are for your personal use only. ***

Note -- best to print these using some non-proprotional font, like courier new,
(for correct formatting)
-- the character ^ is used for exponentiation here (it isn't in C++)

1. Consider the definition:
T(n) = O(f(n)) if there exists constants n0 and c > 0
so that T(n) <= c*f(n) for all integers n > n0.

Find values for n0 and c to prove the following are big-oh relationships.
(a). Prove 2^(n+10) is O(2^n)

2^(n+10) = 2^10 * 2^n add exponents when you multiply

So let n0 = 1 and c = 2^10.
Then 2^(n+10) <= 2^10 * 2^n for all n > 1.

(b). Prove n^10 is O(3^n)

This one is easier to see with experimentation. 3^n is an exponential function which eventually grows very fast while n^10 is a polynomial, a polynomial with a large power, but still a polynomial. The question is when does 3^n surpass n^10? I wrote a simple program comparing 3^n to n^10. Here is the output showing when 3^n surpasses n^10. You can see once 3^n starts growing, passing n^10, it grows big very fast. (Output is from using an 8-byte long int.)

```
n = 30   n^10 =      590490000000000
         3^n  =      205891132094649

n = 31   n^10 =      819628286980801
         3^n  =      617673396283947

n = 32   n^10 =     1125899906842624
         3^n  =     1853020188851841

n = 33   n^10 =     1531578985264449
         3^n  =     5559060566555523

n = 34   n^10 =     2064377754059776
         3^n  =    16677181699666569

n = 35   n^10 =     2758547353515625
         3^n  =    50031545098999707
```

So let n0 = 32 and c = 1 . Then n^10 <= 1*3^n for all n >= 32.

This is the way to attack it if you want c = 1. An alternative arithmetic approach is to choose n0 and figure out what c must be.

(c). Prove that O(n) + O(n) = O(n).

Let T1(n) = O(n). Then there exists positive constants c1, n1
such that T1(n) <= c1 * n for all n > n1.
Let T2(n) = O(n). Then there exists positive constants c2, n2
such that T2(n) <= c2 * n for all n > n2.

To show O(n)+O(n) = O(n), we must show there exists positive
constants c, n0 such that T1(n) + T2(n) <= c * n for all n > n0.

```
     By above, T1(n) + T2(n) <= c1*n + c2*n   for all n > max(n1,n2).
     So let  c = c1 + c2  and  n0 = max(n1,n2).
     Then T1(n) + T2(n) <= c1*n + c2*n    for all n > n0
                        <= (c1 + c2) * n
                        <= c * n

     Thus,  T1(n) + T2(n) = O(n) + O(n) = O(n).

-------------------------------------------------------------------------------
2.  Give a big-oh upper bound on the running time of the trivial selection
    statement
         if (C) { }
    where  C  is a condition that does not involve any function calls.

Whether the condition C is true or false, the running time of the if is O(1).

-------------------------------------------------------------------------------
3.  Repeat the last problem for the trivial while-loop
         while (C) { }

If the condition C is false, the running time of the while-loop is O(1).
If the condition is true, the while-loop executes forever and the running
time is not defined.

-------------------------------------------------------------------------------
4.  Give a rule for the running time of a selection statement in which
    we can tell which branch is taken, such as
         if (1 == 2)
             something O(f(n));
         else
             something O(g(n));

In this case, the running time is O(g(n)).  In general, the running time is
that of the branch taken.

-------------------------------------------------------------------------------
5.  Give a rule for the running time of a degenerate while-loop, in which
    the condition is known to be false right from the start, such as
         while (1 != 1)
            something O(f(n));

If the condition C is know to be false, the running time is O(1), the constant
time to evaluate the condition.

-------------------------------------------------------------------------------
6.  Give an analysis of the running time (find tight big-oh) of a function
    which finds the average of the elements of an array A[0..n-1].
              n-1                 n
              ---                ---
              \                  \
T(n) = O( 1 + /   1 ) =   O( 1 + /   1 ) = O(1+n).
          ^   ---                ---
          |   i=0                i=1
          |
          |for initialization before loop and arithmetic after loop

Thus the running time of the entire program is O(n).

-------------------------------------------------------------------------------
7.  The following is a program fragment that applies the powers-of-2 operation
    to the integers i from 1 to n.  Give a tight big-oh upper bound of the
```

running time. A discussion is sufficient proof of complexity.

```
for (i = 1; i <= n; i++) {
   m = 0;
   j = n;
   while (j > 0) {
      j /= 2;
      m++;
   }
}
```

The assignment statements each take O(1) time. The running time of the while-loop is O(log n) because the value of j is cut in half each time through the loop. Since the while loop is nested in the for loop, log n execution steps happens each time through the loop. Thus the running time of the for loop is O(n log n).

---

8. Give an analysis of the running time (find tight big-oh) for the following segments of code.

(a).
```
for (sum = 0, int i = 1; i <= n; i++)
   for (int j = 1; j <= n*n; j++)
      sum++;
```

```
 n    n^2          n
---   ---         ---
\     \           \
/     /    1  =   /    n^2  = n^3   So T(n) = O(n^3)
---   ---         ---
i=1   j=1         i=1
```

(b).
```
for (sum = 0, int i = 1; i <= n; i++)
   for (int j = 1; j <= i; j++)
      sum++;
```

```
 n     i           n
---   ---         ---
\     \           \        n(n+1)
/     /    1  =   /    i  = ------       So T(n) = O(n^2)
---   ---         ---         2
i=1   j=1         i=1
```

(c).
```
for (int i = 0; i < n; i++)
   sum++;
for (int i = 0; i < n; i++)
   sum++;
```

```
n-1     n-1         n       n
---     ---        ---     ---
\        \         \        \
/   1 + /   1  =   /   1 + /   1  = n + n = 2n    So T(n) = O(n)
---     ---        ---     ---
i=0     i=0        i=1     i=1
```

(d).
```
for (sum = 0, int i = 1; i <= n; i++)
   for (int j = 1; j <= i*i; j++)
      if (j % i == 0)
         for (int k = 1; k <= j; k++)
            sum++;
```

First, analyze the if statement. Consider only the j loop (deal with the i loop later). The "if" is executed i^2 times or at most n^2 times. But, it is only true O(n) times because it is true exactly i times. Thus the innermost loop, the k loop, is only executed O(n) times. The k loop still goes up to j which is at most n^2.

If this is hard to see, try it with numbers. If n (worse case for i) is 10, then n^2 (worse case for j) is 100. So j is 100 when i is 10. Consider when j % 10 is true (as j runs through the ints up through 100) . It's true 10 times (n times) (when j is 10, 20, 30, ..., 100).

$$\sum_{i=1}^{n} \left(i * \sum_{k=1}^{j=O(n^2)} 1 \right) = \sum_{i=1}^{n} (i * n^2) = n^2 * \sum_{i=1}^{n} i = n^2 * \frac{n(n+1)}{2}$$

So, T(n) = O(n^2*(1/2 n^2 + 1/2 n)) = O(1/2 n^4 + 1/2 n^3) = O(n^4)