CompSci 100e Program Design & Analysis II
Spring 2011 Rodger & Forbes Analysis Questions

Due Thursday, March 3

You should write up your solutions to these problems on your own and compose your answers in LaTeXor other suitable system for creating files. Submit a **PDF** of your solutions using Eclipse under assignment name *written1.*

Your submission must include a `README` with the standard information: name and NetID, people with whom you consulted, resources used, and hours spent.

1. Consider the following three algorithms for determining whether anyone in the room has the same birthday as you.

   Algorithm 1: You say your birthday, and ask whether anyone in the room has the same birthday. If anyone does have the same birthday, they answer yes.

   Algorithm 2: You tell the first person your birthday, and ask if they have the same birthday; if they say no, you tell the second person your birthday and ask whether they have the same birthday; etc, for each person in the room.

   Algorithm 3: You only ask questions of person 1, who only asks questions of person 2, who only asks questions of person 3, etc. You tell person 1 your birthday, and ask if they have the same birthday; if they say no, you ask them to find out about person 2. Person 1 asks person 2 and tells you the answer. If it is no, you ask person 1 to find out about person 3. Person 1 asks person 2 to find out about person 3, etc.

   (a) For each algorithm, what is the factor that can affect the number of questions asked (i.e., the *problem size*)?

   (b) In the worst case, how many questions will be asked for each of the three algorithms?

   (c) For each algorithm, say whether it is constant, linear, or quadratic in the problem size in the worst case.

2. We write $f(n) \in O(g(n))$ (aloud, this is "$f(n)$ is in big-Oh of $g(n)$") to mean that the function f is eventually bounded by some multiple of $|g(n)|$. More precisely,

   $$f(n) \in O(g(n)) \text{ iff } |f(n)| \leq c \cdot |g(n)|, \text{ for all } n > n_0,$$

   for some constants $c > 0$ and n_0. That is, $O(g(n))$ is the *set* of functions that "grow no more quickly than" $|g(n)|$ does as n gets sufficiently large. Somewhat confusingly, $f(n)$ here does not mean "the result of applying f to n," as it usually does. Rather, it is to be interpreted as the *body of a function* whose parameter is n. Thus, we often write things like $O(n^2)$ to mean "the set of all functions that grow no more quickly than the square of their argument."[1]

   Suppose $T_1(n) \in O(f(n))$ and $T_2(n) \in O(f(n))$. Answer whether the following are true or false and give justification. A justification for being true is a simple mathematical proof, while you can prove something to be false by giving a *specific* counterexample. You should use the above definition of big-Oh in both cases.

   (a) $T_1(n) + T_2(n) \in O(f(n)$

[1] Questions adapted from the notes of Paul Hilfinger

(b) $T_1(n) - T_2(n) \in O(f(n)$

(c) $T_1(n)/T_2(n) \in O(1)$

(d) $T_1(n) \in O(T_2(n))$

3. An algorithm takes 0.5ms for input size 100. How long will it take for input size 500 if the running time is the following? What assumptions do you have to make in order to answer this problem?

   (a) $O(n)$

   (b) $O(n \log n)$

   (c) $O(n^2)$

   (d) $O(n^3)$

4. By doubling the capacity of an array used to store an `ArrayList`, we pay constant amortized time for each *add* operation. Suppose that allocating a `ArrayList` containing M elements takes $M/2 + 10$ time units, copying M elements from one `ArrayList` to another takes M time units, and an add takes 1 time unit plus the amount of time (if any) required to increase the size of the `ArrayList`.

   (a) If the capacity of the `ArrayList` increases by 100 (that is, 100 more elements, not 100 times as many), how long will N *add* operations take?

   (b) If the capacity of the `ArrayList` doubles each time the array fills up, how long with N *add* operations take?

   (c) If the capacity of the `ArrayList` increases by a factor of 1.5 each time, how long will N *add*s take?

5. What is the big-Oh of `calc`, in terms of n, the size of the array `v`? Briefly justify your answer.

```
int subcalc1(int[] v1)
{
  int sum = 0;
  for (int i=0; i < v1.length; i++)
    sum = sum + v1[i]*v1[i]*v1[i];
  return sum;
}

int subcalc2(int[] v2)
{
  int sum = 0;
  for (int i=0; i < v2.length; i++)
    for (int j=0; j < i; j++)
      sum = sum + v2[i]*v2[j];
  return sum;
}

int calc(int[] v)
{
  return subcalc1(v) + subcalc2(v);
}
```

6. What is the big-Oh of each method below, in terms of `n`? Briefly justify your answers.

   (a)
```
public int calc2(int n){
    int sum = 0;
    for(int k=0; k < n; k++){
      sum++;
```

```
  }
  for(int k=0; k < n; k++){
    sum++;
  }
  return sum;
}
```

(b)
```
int power2(int n)
{
  int prod = 1;
  while (prod < n)
    prod = prod * 2;

  return prod;
}
```

(c)
```
public int clunk(int n){
  int sum = 0;
  for(int j=0; j < n; j++){
    for(int k=0; k < j; k++){
      sum++;
    }
  }
  return sum;
}
```

(d)
```
public int goduke(int n){
  int sum = 0;
  for(int k=1; k <= n; k = k * 2){
    sum++;
  }
  return sum;
}
```

7. **Extra Credit**: Big Omega gives the lower bound for a function. More precisely, $f(n) \in \Omega(g(n))$ means that for all $n > n_0$, $|f(n)| \geq c|g(n)|$ for $n > n_0$, for some constants $c > 0$ and M. That is, $\Omega(g(n))$ is the set of all functions that "grow at least as fast as" g beyond some point.

   Show $n! \in \Omega(2^n)$.