CS241
Spring 2019
Homework 1

In this homework (and in this class in general), log = base-2 logarithm unless otherwise specified.

1. **Abstract data types**

   You are designing an ADT called `Time` to represent a time of day; that is, the time that you'd see on a clock on the wall. Conceptually, a time consists of an hour component, and minute component, and an AM/PM designation. A Time has no associated date or day of the week. In this question you will think about designing this data type in two different ways.

   Recall that an ADT is primarily defined by *what* the operations are that it can do, and now *how* it does those operations. Assume the operations we care about are as follows:

   - Get the hour component of the time.
   - Get the minute component of the time.
   - Get the AM/PM component of the time.
   - Compare this `Time` instance against another `Time` instance to figure out which one is earlier.
   - Get the different in minutes between this `Time` instance and another `Time` instance. (e.g., the difference between 1pm and 2:30pm is 90 minutes).

   For instance, if we were designing a C++ class for this, the public portion might look like this:

```
class Time
{
  public:
  int getHour() const;
  int getMinute() const;
  string getAMorPM() const;
  bool isEarlier(const Time & otherTime) const;
  int differenceInMinutes(const Time & otherTime) const;
}
```

   (a) Describe two different ways of implementing this ADT in terms of what variables (and corresponding types) you would use to implement the private section of the `Time` class. There are a number of different implementations you can choose --- try to pick one that could make certain operations easier or more straightforward to implement, and then pick another that makes a different group of operations easier to implement.

(b) For each of your implementations, describe how you would implement each of the five operations above. You don't need to write C++ code (but you can if you want to); a sentence or phrase or two for each operation is fine.

(c) Contrast your two implementations in terms of which operations are easier or more straightforward to implement in one of your designs, and which are easier or more straightforward to implement in your other design.

2. **Big-oh ranking**

Order the following big-oh complexities in increasing order (from slowest to fastest). It is possible some of them are actually in the same complexity category. If that is the case, make it clear which ones have the same complexity.

n^2, 3^n, $\sqrt{n}$, 1, n*log(n), 2^n, n!, $2^{\log(n)}$, n^3, n, $n^2\log(n)$, log(n), 2^{n+1}

3. **Big-oh complexity**

Assume each formula below represents the running time T(n) of some algorithm. For each formula, give the lowest big-oh complexity possible (the tightest bound).

Here, log represents the base-2 logarithm.

(a) $5 + n^2 + 25n$
(b) $50n + 10n^{1.5} + 5n*\log(n)$
(c) $3n + 5n^{1.5} + 2n^{1.75}$
(d) $n^2\log(n) + n*\log(n) + n*(\log(n))^2$
(e) $2^n + n^{10}$
(f) $n*\log(n) + 8n + n*(\log(n))^2$

4. **Big-oh complexity proof**

Assume we have analyzed an algorithm and its run time is determined to be

$T(n) = n^3 + 2n + 3$

(a) What is the big-oh running time of this algorithm? (This should be easy.) Call this function f(n).

(b) Now, prove your answer.

In other words, find a constant c and a number n_0 such that for all numbers n >= n_0, T(n) <= c * f(n). Draw a graph showing T(n) and c * f(n). Label your axes and where n_0 is.

5. **Big-oh complexity analysis of a recursive algorithm**

   Assume we have an recursive algorithm whose running time we've determined to be T(n) = 2T(n/2) + 1, with T(1) = 1. Determine the non-recursive T(n) for this algorithm (using the iterative substitution method from class), and then determine the big-oh running time. Show your work.

6. **Big-oh complexity analysis of code**

   Determine the big-oh running time for the following algorithms in terms of n. (No justification needed.)

   a. Matrix addition:

```
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < n; j++)
  {
    c[i][j] = a[i][j] + b[i][j];
  }
}
```

   b. Matrix multiplication:

```
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < n; j++)
  {
    c[i][j] = 0;
    for (int k = 0; k < n; k++)
    {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

   c.

```
while (n >= 1)
{
  n = n/2;
}
```

d.

```
x = 1;
for (int i = 1; i <= n-1; i++)
{
  for (int j = 1; j <= x; j++)
  {
    cout << j << endl;
  }
  x = x * 2;
 }
```