**Big-oh problems**

What is the running time of each of the following examples, expressed in big-oh notation? Assume "n" is the input size to the algorithm (the amount of data the algorithm must process).

```
Algorithm A:
for (int x = 0; x < n; x++) {
  System.out.println("Hi!");
}

Algorithm B:
for (int x = 0; x < n; x++) {
  for (int y = 0; y < n; y++) {
    System.out.println("Hi!");
  }
}

Algorithm C:
for (int x = 0; x < n; x++) {
  for (int y = 0; y < n; y++) {
    for (int z = 0; z < n; z++) {
      System.out.println("Hi!");
    }
  }
}

Algorithm D:
for (int x = 0; x < 100; x++) {
  System.out.println("Hi!");
}

Algorithm E:
for (int x = n; x > 0; x--) {
  System.out.println("Hi!");
}

Algorithm F:
for (int x = 0; x < n/2; x++) {
  System.out.println("Hi!");
}

Algorithm G:
for (int x = 0; x < Math.log(n); x++) {
  System.out.println("Hi!");
}

Algorithm H:
for (int x = 0; x < n; x *= 2) {
  System.out.println("Hi!");
}
```

**Rules of Big-Oh Notation**

Remember that big-oh notation is used to quantify the running time of an algorithm. Specifically, it tells us how the running time of an algorithm *grows* as the amount of input (the input size) to the algorithm grows. In particular, the "O" of big-oh stands for **order**, as in order of growth.

There are a few common ways to determine the running time of an algorithm. Formally, we do this by determining a formula for the number of basic operations the algorithm does, usually culminating in a formula T(n), where n = the input size (the amount of data we give to the algorithm).

We can then turn the T(n) formula into a big-oh category. The big-oh categories represent groups of related T(n) formulas that all grow roughly at the same rate, as n gets "very big" (formally, as n approaches infinity).

The most common big-oh categories (at least for COMP 142) are (in order of slowest-growing to fastest-growing):

- O(1), corresponding to algorithms that run in **constant time**. These are algorithms that take the same amount of time no matter how much input they are given.

- O(log n), algorithms that run in **logarithmic time**. These are algorithms that run in time proportional to the logarithm of their input size. An equivalent way of looking at these algorithms is that as the input size doubles, the algorithm only slows down by a constant amount. By definition, these algorithms cannot examine their entire inputs (because if they did, they would take linear time).

- O(n), algorithms that run in **linear time**. These are algorithms that run in time proportional to their input size. As the input size doubles, the algorithm's running time also doubles.

- $O(n^2)$, algorithms that run in **quadratic time**. These are algorithms that run in time proportional to the square of their input size. As the input size doubles, the algorithm's running time quadruples.

- $O(2^n)$, algorithms that run in **exponential time**. These are algorithms that run in time proportional to 2 (or another constant number) raised to the power of their input size. These algorithms double in running time as the input size grows by a constant amount (often 1). Contrast this with linear-time algorithms, which only double in running time when the input size also doubles.

There are other big-oh categories (in fact, infinitely many), but these are the common ones you will encounter in this class.

To transform a T(n) formula into a big-oh category, we do the following:

- Most T(n) formulas have only one term, or multiple terms added together. If there are multiple terms, drop all terms except the fastest-growing one.
  *Example*: If $T(n) = 3n^2 + 4n + 19$, drop the 4n and the 19 because $3n^2$ is the fastest-growing term.
  *Example*: If $T(n) = 2n + 7\log(n) + 8$, drop the 7log(n) and the 8 because 2n is the fastest-growing term.

- Drop the coefficient (if there is one) on the single term remaining.
  *Example*: If $T(n) = 3n^2 + 4n + 19$, drop the 4n and the 19 because $3n^2$ is the fastest-growing term, then drop the 3 from the $3n^2$ to leave just $n^2$.
  *Example*: If $T(n) = 2n + 7\log(n) + 8$, drop the 7log(n) and the 8 because 2n is the fastest-growing term, then drop the 2 from the n to leave just n.

- Whatever is left is the big-oh category.
  *Note*: if there is no factor of n at all, such as in T(n) = 13, the big-oh category is T(1).

Normally, one can use shortcuts to skip straight from code or pseudocode to a big-oh category, without finding the actual T(n) formula. This is usually done by examining the code to count the loops or study any recursion in the code.