COLLABORATORS: N/A

1: Big oh and running times

(a) (1) $O(n^2)$ (other terms are smaller asymptotically)
(2) $O(1/n)$ (as $n \to \infty$, this is the term that dominates. Some of you wrote $O(1)$, which is technically correct, just as $O(n^3)$ is also a valid answer for part (1). But we would like to write bounds that are as tight as possible.)

(b) We can assume that $a \geq b$ (otherwise we have one extra iteration, but the subsequent iterations always satisfy $a > b$). The best way to come up with the following proof is by going through a few numeric examples.

A simple observation is that after two recursive steps, the first argument would be $a \% b$. Now, the key thing to note is that for any $a \geq b$, $a \% b < a/2$. This is because by definition, $a = rb + (a \% b)$, for some $r \geq 1$ and $a \% b < b$. Thus $a > 2(a \% b)$.

This means that after two recursive steps, the first argument reduces by a factor at least 2. Thus the total number of recursive steps is at most $2 \log_2 n$, which is $O(\log n)$ (as $a, b \leq n$).

2: Eager or not?

(The implicit assumption here, which at least one student wasn’t sure about, is that computation cannot be paused and re-started on a different machine.)

If the student starts the computation right away, it takes 7 years. If she starts after 2 years, it would take $2 + 3.5 = 5.5$ years. If she started after 4 years, it would take $4 + 1.75 = 5.75$ years. Waiting longer would clearly take time $> 6$ years. Thus the optimum is to wait 2 years.

3: Square vs Multiply

Suppose we have two $n$ digit numbers $a, b$ that we wish to multiply. Let $A()$ be the algorithm for squaring that takes time $O(n \log n)$.

Now, observe that $ab = \left(\frac{(a+b)^2-a^2-b^2}{2}\right)$. Thus to compute $ab$, we can find $A(a+b) - A(a) - A(b)$, and divide by 2. The running time thus consists of first computing $(a+b)$ (time $O(n)$), three calls to $A()$ (time $O(n \log n)$), plus the time for division ($O(1)$ or $O(n)$, depending on how the division is done). Thus the overall time is $O(n \log n)$.

4: Graph basics

As $G$ is a simple graph, the degrees all lie in the range $[0, n-1]$. Thus the only way the degrees are all distinct is if there is precisely one vertex of degree $i$, for each $i = 0, 1, 2, \ldots, n-1$. However, if there is a vertex of degree $n-1$, it is connected to all the other vertices, and thus there cannot be a vertex of degree 0! This is a contradiction, and thus we cannot have all distinct degrees.

5: Basic Probability
(a) Any outcome of the tosses can be written as a string of length \(k\) (e.g. HTTTHHH, \ldots). There are \(2^k\) total possibilities, all of which are equally likely. The ones that contain precisely one heads are HTTTTT\ldots, THTTTT\ldots, \ldots, and thus there are \(k\) of them. Thus the probability is \(k/2^k\).

(b) There are \(k^k\) total colorings of the boxes (one way to see this is: a coloring is a string of length \(k\), and each position in the string is one of \(k\) colors). All these colorings are equally likely.

Now, the number of colorings in which the boxes all get different colors is exactly \(k! = k(k-1)(k-2)\ldots1\) (one way to see this is: we can color the first box with any of the \(k\) colors, and having done so, the second box can be colored with any of the remaining \((k-1)\) colors, and so on).

Thus the desired probability is \(k!/k^k\). (Interestingly, this turns out to be roughly \(e^{-k}\).)

(c) The probability of seeing heads for the first time after \(i\) throws is \((1/6)(5/6)^{i-1}\). Thus, if 
\[X\] is the random variable that is the index of the first heads, then \(\Pr[X = i] = (1/6)(5/6)^{i-1}\). Thus,
\[
\mathbb{E}[X] = \sum_i i \cdot \Pr[X = i] = \sum_{i \geq 1} i \cdot \frac{1}{6} \left(\frac{5}{6}\right)^i .
\]

By standard manipulations, this can be shown to be equal to 6. (There are many ways of doing this.)

Next, the probability of not seeing a 1 for \(n\) steps is \((5/6)^n\). We need this to be < \(1/100\). A quick computation shows this to happen at \(n = 26\). (As \(\log(100)/\log(6/5) \approx 25.258\ldots\).)

6: Array Sums

Let us describe an \(O(n^2 \log n)\) time algorithm.

**Algorithm:** First, sort the elements of \(A\). Now, for every choice of \(0 \leq i < j < n - 1\), compute \(A[i] + A[j]\), and then check (using binary search) if \(-(A[i] + A[j])\) is present in the array \(A[j + 1, \ldots, n - 1]\). Output YES if the search is successful. If the search above fails for all \(i, j\), output NO.

**Correctness:** If there exist indices \(i < j < k\) with \(A[i] + A[j] + A[k] = 0\), then the search for \(-(A[i] + A[j])\) must succeed. Likewise, if the algorithm succeeds, we have found three indices such that the above holds.

**Running time:** The initial sorting takes \(O(n \log n)\) time. Then, we perform \(n^2\) binary searches. Each takes time \(O(\log n)\), thus the overall run time is \(O(n^2 \log n)\).

[With a bit more care, one can solve this problem with running time \(O(n^2)\).]