CS 6150: HW0 Solutions

August 31, 2021

1. Big oh and running times . . . .

   (a) Write down the following functions in big-O notation:

      1. **[2]** $f(n) = n^2 + 5n + 20$

         . . . .

         $O(n^2)$, since other terms are asymptotically smaller.

      2. **[2]** $g(n) = \frac{1}{n^2} + \frac{2}{n}$

         . . . .

         $O(\frac{1}{n})$, since as $n \to \infty$ this term dominates. $O(1)$ is technically a bound, just as $O(n^3)$ is also a valid answer for Question 1. But we would like to write bounds that are as tight as possible.

   (b) **[6]** Consider the following algorithm to compute the GCD of two positive integers a, b. Suppose a, b are numbers that are both at most n. Give a bound on the running time of $\textsc{Gcd}(a, b)$. (You need to give a formal proof for your claim.)

Algorithm 1 $\textsc{Gcd}(a, b)$

if $(a < b)$ return $\textsc{Gcd}(b, a)$;
if $(b = 0)$ return a;
return $\textsc{Gcd}(b, a\%b)$; (Recall: $a\%b$ is the remainder when a is divided by 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 $a\%b$.

Claim: For any a, b, where $a \geq b$, $a\%b < a/2$

Proof: We can represent a by the sum of smaller parts: $a = rb + (a\%b)$ for some $r \geq 1$ and $a\%b < b$. For $r = 1$, we know that $a\%b < a/2$, since if it was not, then either r must be greater than 1 or $a < b$, both of which contradict our assumptions. And for any $r > 1$, $a\%b < b < a/2$. 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 $2\log_2 n$, where $n = \max\{a, b\}$.

3 points for argument of decreasing by a factor of 2, 3 points for providing claim of $O(\log n)$

2. **[5]** Suppose I tell you that there is an algorithm that can square any n digit number in time $O(n \log n)$, for all $n \geq 1$. Then, prove that there is an algorithm that can find the product of *any two* n digit numbers in time $O(n \log n)$. [*Hint:* think of using the squaring algorithm as a subroutine to find the product.]

   . . . .

Suppose we have two n digit numbers a, b that we wish to multiply. Let $A()$ be the algorithm for squaring that takes time $O(n \log n)$. The key point is that we don't know anything about the workings of the squaring algorithm (it's a black-box, for our purposes). We need to use just the fact that such an algorithm *exists* to prove that multiplication can be done in $O(n \log n)$ time.

The key to the proof is noting that $ab = \frac{(a+b)^2 - a^2 - b^2}{2}$. Thus to compute $a * bs$, 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 $O(n \log n)$.

3 points for correct algebraic proof of correctness. 2 points for **explanation** of runtime.

3. Graph basics [**8**]

   Let G be a simple, undirected graph. Prove that there are at least two vertices that have the same degree.

   ....

   Clarification: You may assume that the graph G has at least two vertices.

   Proof. Let $G = (V, E)$ be a graph, and let $n = |V| \geq 2$. For the sake of contradiction, assume that the claim does not hold. Because simple graphs cannot have more than $n-1$ neighbors, $0 \leq \deg(v) \leq n-1$ for every $v \in V$. If there are no two vertices having the same degree, then all n vertices mush have distinct degrees, that is, $\{\deg(v) : v \in V\} = \{0, 1, \ldots, n-1\}$. Let $u \in V$ be the vertex with degree 0, and $w \in V$ be the vertex with degree $n-1$. u has no neighbors, but w must have all vertices except w in its neighbors. This is a contradiction. $\square$

   3 points for observing max degree of $n-1$. 5 points for proof

4. (a) [**3**] Suppose we toss a fair coin k times. What is the probability that we see heads precisely once?

   ....

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

   1 point for correct claim of $k/2^k$, 2 points for justification

   (b) [**4**] Suppose we have k different boxes, and suppose that every box is colored uniformly at random with one of k colors (independetly of the other boxes). What is the probability that all the boxes get distinct colors?

   ....

   There are k^k total possible outcomes. All these outcomes are equally likely. Now, the number of colorings in which all colors are used is exactly $k! = k(k-1)(k-2)\ldots 1$ (one way to see this is: we paint the first box with any of the k colors, and having done so, the second box can paint 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}.)

   2 points for correct numerator, 2 points for correct denominator (with reasonable justification)

   (c) [**5**] Suppose we repeatedly throw a fair dice with 6 faces. What is the expected number of throws needed to see a '1'? How many throws are needed to ensure a '1' is seen with probability $> \frac{99}{100}$?

   ....

   The probability of seeing 1 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 1, 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.)

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

2 points for correct number of throws, 6. 4 points for any reasonable justification

5. Tossing Coins ....
Suppose we have two coins, one of which is fair (prob[heads] = prob[tails] = 1/2), and one of which is biased with prob[heads] = 0.51. Suppose we toss the coin N times and let H_1 and H_2 be the number of heads observed, respectively.

   (a) **[3]** Intuitively, how large must N be so that we have $H_2 > H_1$ with "reasonable certainty"?

   ....

   If we toss N times, the expected value of H_1, H_2 are $0.5N$ and $0.51N$. The difference is $0.01N$. If this quantity is to be ≥ 1, we would expect $N \geq 100$. To be reasonably sure, it's safe to guess say $N = 200$. (Formaly computations like this are done via what are known as concentration inequalities.)

   2 points for any value higher than 100, 1 point for reasonable explanation

   (b) **[2]** Suppose we pick $N = 25$. What is the expected value of $H_2 - H_1$?

   ....

   We use the fact that $\mathbb{E}[H_2 - H_1] = \mathbb{E}[H_2] - \mathbb{E}[H_1]$ (this is called the *linearity of expectation*). Then the computation above gives the answer $0.01N = 1/4$.

   1 point for using linearity of expectation, 1 point for correct value.

   (c) **[2]** Can you use this to conclude that the probability of the event $(H_2 - H_1 \geq 1)$ is small?

   ....

   For those of you know, it is tempting to use the so-called Markov's inequality. However Markov's inequality is only applicable to non-negative random variables (which $H_2 - H_1$ is not). Thus we need to look at the distribution more carefully to show the desired bound.

   2 points for any cogent statement

6. Array Sums **[8]** ....
Given an array $A[1\ldots n]$ of integers, find if there exist indices i, j, k such that $A[i] + A[j] + A[k] = 0$. Can you find an algorithm with running time $o(n^3)$? [NOTE: this is the little-oh notation, i.e., the algorithm should run in time $< cn^3$, for any constant c, as $n \to \infty$.] [*Hint:* aim for an algorithm with running time $O(n^2 \log n)$.]

....

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 $O(n^2)$.]

2 points for algorithm description, 2 points for correctness argument, 2 points for running time argument, 2 points for algorithm with $o(n^3)$ running time, not $O(n^3)$