Diego Escobedo February 13, 2020

6.046 Problem Set 1

Collaborators: *Mikael Nida, Ritaank Tiwari, Temi Omitogoon*

Problem 1

(A) By Master's theorem, we know that a = 8, b = 2, d = 2, and $\log_b$a = 3. Since $\log_b$a > d, then the problem can be solved in O($n^{log_b a}$) = O(n^3)

(B) Let us define a variable m s.t. n = 2^m. So, our recurrence becomes T(2^m) = 8T($2^{m/4}$) + $\log^4(2^m)$. We create a new function S(m) = T(2^n). So, we get S(m) = 8S($m/4$) + m^4. We can use Master's Theorem, where a = 8, b = 4, and d = 4. $log_b a = 3/2$, but since d is bigger, we get a running time of O(n^4), and if we substitute it back in, then we get a total running time of O($log^4(n)$).

(C) In this case, our recurrence chain extends n times. Since we are doing 3n work at each level, that means our algorithm runs in $3n^2$, or O(n^2)

(D) We can use our intuition to intuit that since 1/7 and 2/5 add up to less than 1, we believe that we can guess a solution of the form T(n) $\leq$ cn for all n < k for some k, using strong induction. We will show that T(k) $\leq$ ck. T(k) = T(k/7) + 2T(k/5) + c_2k. for some fixed c_2. Now, by our inductive hypothesis, T(k) = 2/5*ck + 1/7*ck + c_2k, which is equivalent to (17/35*c + c_2)k $\leq$ ck whenever c $\geq$ 35/16*c_2. This proves our claim for a large enough c, by induction.

(E)
1. $log(log(n^{20}))$ = O($log_9(n)$) = O($log(n)$) = O($log(n^5)$) = o($log^9(n)$) = o($n^{(1/5)}$) = o(n)
2. $log(n!)$ = O($nlog(n^2)$) = o($n^2 loglogn$) = o(n^7) = o(3^n)
3. n^{logn} = o(5^n) = o($log(n)^{n-1}$) = o($log(n)^n$)

Problem 2

(A) Since each of the dice rolls is independent (we 'start again' if we don't roll a 5 or a 7), then the chance of rolling a 5 is the chance of getting one of the following sets: $\{(1,4),(4,1),(2,3),(3,2)\}$. This is $4/36 = 1/9$th chance of rolling a 5 on a given roll. Furthermore, it is easy to see that chance of rolling a 7 is the chance of getting one of the following sets: $\{(1,6),(6,1),(2,5),(5,2),(3,4),(4,3)\}$. This is $6/36 = 1/6$th chance of rolling a 7 on a given roll. Therefore, on each roll, the chance of not rolling a 5 or a 7 is 26/36. Therefore, $\mathrm{P}(E_n) = (\frac{26}{36})^{n-1} * \frac{1}{9}$

(B) From the hint, we know we want to get $\sum_{n=1}^{\infty}(\frac{26}{36})^{n-1} * \frac{1}{9}$ which is equivalent to $\frac{1}{9} * \sum_{n=0}^{\infty}(\frac{26}{36})^{n}$. This, in turn, is equal to $\frac{1}{9} * \frac{1}{1-\frac{26}{36}}$. This is ultimately equal to 2/5, which represents the chance of getting a 5 on the first try + the second try + the third try and so on and so forth. This is our desired answer because if we condition the dice roll on getting a 5 or a 7, we saw above that there are 10 outcomes that result in a 5 or a 7, and 4 of them are 5s. Therefore, since the dice rolls are independent, we don't really care what happened before, and if we are going til we get a success (5) or failure (7), then the chance at each new stage should be the same as originally.

(C)
1. The expected value for a 6 sided die is $\frac{1}{6} * (1+2+3+4+5+6) = 3.5$
2. $\mathrm{E[X]} = \sum_i E[X_i] = 350$
3. $\mathrm{Var[X]} = \sum_i Var[X_i] = \sum_i E[X_i^2] - E[X_i]^2 = \sum_i \frac{91}{6} - \frac{49}{4} = \frac{875}{3}$.
4. Using Chebyshev's inequality, we know that the upper bound on our desired probability is $\frac{var(x)}{k^2} = \frac{875}{3*50^2}$ which is roughly 11.6%.

Problem 3

(A) To deterministically determine whether the array is mostly-sorted or not, the problem is that we have to actually observe the 9/10n elements to determine whether it is mostly sorted or not. If we look at any less than 9/10n of the elements, say for example cn where c $< 9/10$, then we run the risk of accidentally finding a subset of length cn that is sorted when the rest of the array is not sorted. Therefore, to completely verify whether it is sorted, then we need to look at at least 9/10n items, which is $\Omega(n)$

(B) To design an adversarial input, we want to design one that has a minimal number of non-sorted triples while still being not mostly-sorted. The best option, if we have an array of length n, is to make the first n/2 numbers in the array the numbers n/2, n/2+1, n/2+2, ... , n. Then, we would make the second n/2 numbers in the array the numbers 1, 2, ... , n/2-1. Let us analyze this adversarial input. There are n-2 triples that we can sample. Out of these, there are only two that are unsorted, (n,1,2) and (n-1,n,1). Therefore, the chance of getting k sorted triples (which would lead our algorithm to declare that the array is indeed SORTED), is $(1 - \frac{2}{n-2})^k$. We want the probability of this to be less than 1/2 (because its an adversarial input and should therefore return UNSORTED), so let's set up an inequality to find a k that satisfies it. $(1 - \frac{2}{n-2})^k < 1/2$ is equal to $(1 - \frac{2}{n-2})^{(\frac{(n-2)*2}{2*(n-2)})k} < 1/2$ which is equal to $e^{-\frac{2k}{n-2}} < 1/2$ which is equal to $-\frac{2k}{n-2} < ln(1/2)$ which is equal to $k > 1/2 * (n-2)ln(1/2)$. Therefore, k is going to be $\Omega(n)$ to guarantee correctness, which again defeats our purpose.

(C) The key point here is that binary search works by comparing the desired element to a median. If the desired element is bigger than the median, it will look in the right subarray of the array we are currently looking at, and if it is smaller than the median, then it will go to the left subarray. Therefore, if at ANY point, the behavior of BINARY-SEARCH(A, x1, 0, n) and BINARY-SEARCH(A, x2, 0, n) differs, that means that one of them was smaller than the median, and the other one was bigger. Therefore, at some point, we must have determined that $x1 < median(subset) < x2$. QED

(D) Our algorithm proceeds as following: Randomly select 14 elements from the array. Run BINARY-SEARCH on each element to determine the index at which this element would be if it were sorted. Check whether the element at this index is indeed the element we originally ran BINARY-SEARCH on. If Terr is a mismatch between the expected and actual in ANY of them, return UNSORTED. Else, return SORTED.

Correctness: If the array is sorted, then the algorithm is guaranteed to return the correct answer. If it is mostly-sorted, then it doesn't matter what we output. In the adversarial case where the array has 9/10*n-1 elements sorted (i.e. it should return UNSORTED, but is

close to being almost-sorted), then we can simply imagine that we represent these 9/10*n-1 elements as m. Therefore, m < 9/10*n, and m/n < 9/10. Lets say that we choose k random elements to inspect in our algorithm. If we are in the adversarial case, then our worst-case is that all k of our chosen elements to inspect are within the m sorted elements. Because all the BINARY-SEARCH queries return correctly, this would return SORTED even though it should return the opposite. So, we want to find a k that is large enough such that the chance of this happening is less than 1/4. We have an m/n chance of choosing a single correctly sorted element, or a $(m/n)^k$ chance of choosing all k elements from the correctly sorted elements (which will not conclusively tell us whether or not it is sorted). We want our chance of failure to be less than 1/4, so we get the inequality $(m/n)^k < (9/10)^k \leq 1/4$. If we reorganize, we see that k has to be at least around 13.1, which must be 14 to meet the minimum required accuracy. Therefore, we can guarantee that our algorithm has at least a 3/4 chance of succeeding. We know that if we only chose 1 element, in the worst case we would have 9/10 chance of it indicating that the array is sorted when it is not. If we choose 14 of those numbers, however, the chances of only choosing sorted ones decreases significantly, to the point where we a false positive (declaring it SORTED when it is UNSORTED) would happen at most 1/4 of the time.

Runtime: We are running BINARY-SEARCH a constant number of times, which means we have an O(logn) runtime.