Recitation November 10

1. Three people line up to board a flight and everyone has a ticket with an assigned seat. However, the first person in line has lost his ticket and takes a random seat. After that, each person takes the assigned seat if it is unoccupied, and one of unoccupied seats at random otherwise. What is the probability that the 3rd person gets to sit in their assigned seat?

   Solution: The first person sits in their own position with probability 1/3. For this case, everyone else would also sit in their own seats. The first person sits in the 2nd person's seat with probability 1/3. If this happens the second person sits in the first persons seat with probability 1/2 and in the 3rd persons seat with probability 1/2. The third person gets to sit in their own seat in the first of these two cases (which happens with a probability of $1/3 \cdot 1/2 = 1/6$. If the first person sits in the third person's seat, then the third person can't sit in their own seat. So the probability of the third person sitting in their own seat is $1/3 + 1/6 = 1/2$.

   Visually, the following cases are possible (the probabilities are written in parenthesis):
   123(1/3)
   213(1/6)
   312(1/6)
   321(1/3)
   And only the first two cases result in 3 sitting at their own seat.

2. There are 4 people at a party, and everyone has checked in their coats. The coats get mixed up and after the party, everyone gets a random coat. What is the expected number of people who get their own coat?

   Solution: We want to calculate E[number of coats that get correctly assigned]. One way to calculate this would be to write down all 24 permutations and count how many coats are correctly assigned for each permutation and then take their average (sum/24). However, this method would no longer be efficient when n becomes large enough. The other option is to use linearity of expectation to calculate this value.

   We can break down the above expectation into a sum of 4 expectations as follows:

   $$\begin{aligned} &E[\text{number of coats that get correctly assigned}] \\ &= E[\text{number of coats that get correctly assigned to the first person}+ \\ &\quad \text{number of coats that get correctly assigned to the second person} + ...] \\ &= E[\text{number of coats that get correctly assigned to the first person}]+ \\ &\quad E[\text{number of coats that get correctly assigned to the second person}] + ... \end{aligned}$$

   Now calculating each of these individual expectations is easier. For any person, the number of coats to get correctly assigned to that person can either be just 0 or 1.

   The probability that any person gets back their own coat is 1/4. This is because all 24 permutations of the coats are equally likely to happpen, and exactly 6 of those permutations

will have i in the ith position for any i.

So the answer for each of those individual expectations is just $1/4 \cdot 1 + 3/4 \cdot 0 = 1/4$. Hence, the expected number of coats to get correctly assigned $= 4 \cdot 1/4 = 1$.

(Note that for any n this answer would still be just $n \cdot 1/n = 1$.)

3. (Modification of HW9 Q5) Design a probabilistic algorithm that takes an array A of length $2n$ with n X's and n Y's as an input and finds any two distinct i, j such that $A[i] == X$ and $A[j] == X$. The expected number of operations should be $\Theta(1)$ for every input array A. The algorithm and its complexity should be explained and justified.

Solution:
Let's analyze the complexity of the above randomized algorithm. The for loop runs k times

Algorithm 1 Randomized algorithm

```
function FindX1X2Deterministic(A)
    int ans[2]
    int found = 0
    for j = 1 to 2n do
        if A[j] == X then
            A[found++] = j
        end if
        if found == 2 then
            return (ans[0], ans[1])
        end if
    end for
end function

function FindX1X2(A)
    for k = 1 to n  do
        i ← random(1,2n)
        j ← random(1,2n)
        if A[i] == X && A[j] == x && i != j then
            return (i,j)
        end if
    end for
    return FindX1X2Deterministic(A)
end function
```

and each time, it checks for two random entries of A. The probability that both of them contain X is $1/4$ because exactly half the entries in A are X.

(Note: the probability would be a little less than $1/4$ in reality because we're also checking if $i == j$, but even if we take care of that, the probability is greater than $1/16$, and the algorithm runs in constant time as long as this probability is some constant. However, I'll be ignoring that $1/n^2$ probability here to make the proof easier to understand.)

The expected running time of the algorithm is going to be:
$E = 1 \cdot \Pr[\text{answer found in first iteration } (k = 1)] + 2 \cdot \Pr[\text{answer found in second iteration } (k = 2)] + \ldots + n \cdot \Pr[\text{answer found in } n\text{th iteration } (k = n)] + 2n \cdot \Pr[\text{answer found in the deterministic function}]$.
Thus,

$$\begin{aligned} E &= 1 \cdot 1/4 + 2 \cdot 3/4 \cdot 1/4 + 3 \cdot (3/4)^2 \cdot (1/4) + ... + n \cdot (3/4)^{n-1}(1/4) + 2n \cdot (3/4)^n \\ &= 1/4 \cdot \left(1 + 2 \cdot 3/4 + 3 \cdot (3/4)^2 + ... + n \cdot (3/4)^{n-1}\right) + 2n \cdot (3/4)^n \end{aligned}$$

Let's first calculate $S = 1 + 2 \cdot 3/4 + 3 \cdot (3/4)^2 + ... + n \cdot (3/4)^{n-1}$. If we multiply both sides by $3/4$, we get:

$$3S/4 = 3/4 + 2 \cdot (3/4)^2 + 3 \cdot (3/4)^3 + ... + n \cdot (3/4)^n$$

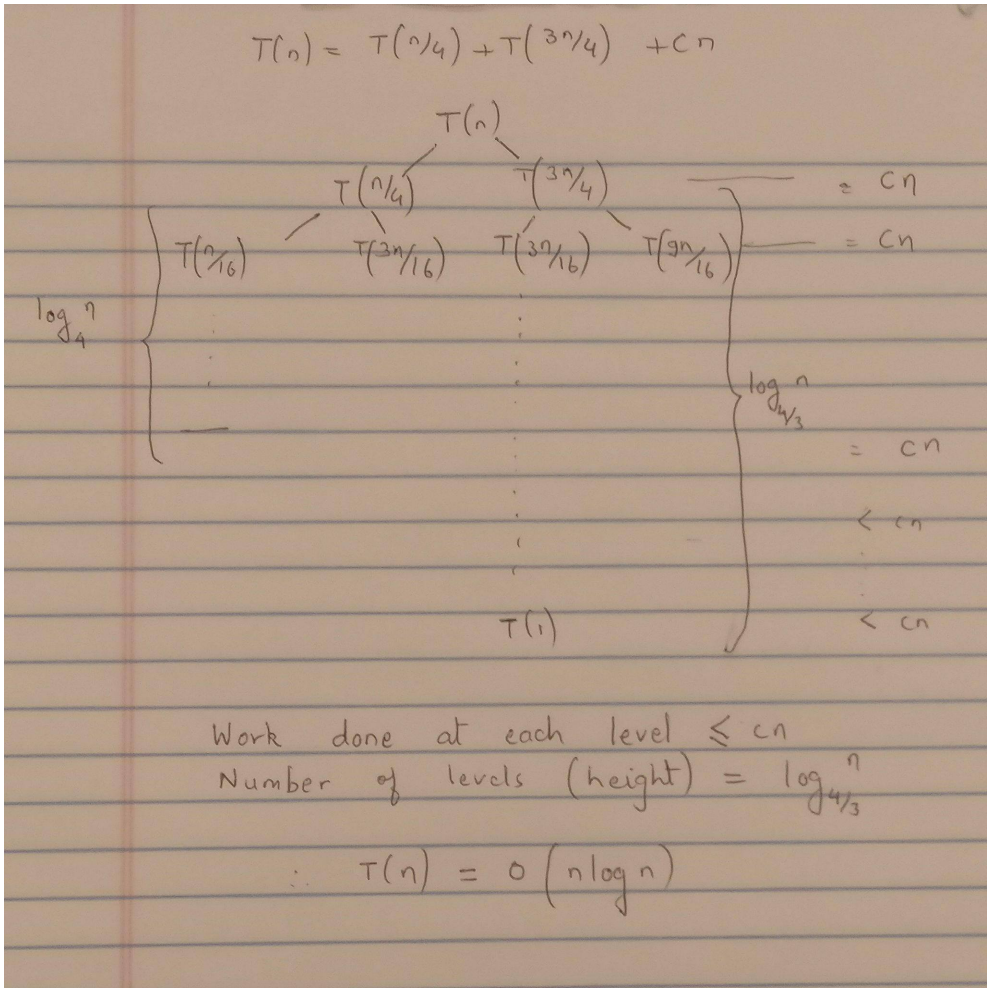
Subtracting this from S, we now get:

$$\begin{aligned} S - 3/4S &= 1 + 3/4 + (3/4)^2 + ... + (3/4)^{n-1} - n \cdot (3/4)^n \\ \therefore S/4 &= \frac{1 - (3/4)^n}{1 - 3/4} - n \cdot (3/4)^n \\ \therefore S/4 &= 4 \cdot (1 - (3/4)^n) - n \cdot (3/4)^n \\ \therefore S/4 &\leq 4 \\ \therefore S &\leq 16 \end{aligned}$$

Also, $\lim_{n \to \infty} n \cdot (3/4)^n = 0$. Thus, as n becomes large enough, E nears 16 which is a constant. Thus, $E = O(1)$

4. Let's suppose we have a function that takes a array of size n as input, and gives back the $n/4$th smallest number in the array in $O(n)$ time. What would be the time complexity of quicksort if we use this function to choose the pivot?
(Hint 1: If we use this function to find the pivot, the array would get divided into 2 arrays of sizes $n/4$ and $3n/4$. So, the question boils down to solving the recursion $T(n) = T(n/4) + T(3n/4) + O(n)$. This is a bit different from the recurrences we've seen so far. However, you can still use a recursion tree to see how this recurrence works. How much work is done at each level? How many levels are there?)
(Hint 2: The answer is $T(n) = O(n \log_{4/3} n)$.)

Solution:



Note: The below two questions are more difficult and thus optional. I've provided brief descriptions of the solutions and added a link if you're still curious about the solution.

5. *(*Hard, Optional)* Consider the following modification to the binary search algorithm that searches for an input x in a sorted array A.

    1. Intialize $L = 0, H = n - 1$
    2. Randomly choose an index m between L and H
    3. if $A[m] == x$: return i
    4. if $A[m] < x : L = m$
    5. else: $H = m$
    6. Go back to step 2

    What's the expected running time of this algorithm?

    Solution: $O(\log_n)$. The proof is not that easy. The main idea is that the algorithm runs in the same way that a randomly built binary search tree would work. And the result follows from the fact that the height of a randomly built binary search tree is $O(\log n)$. The proof is here.

6. *(*Hard, Optional)* Try solving problems 1 and 2 for n instead of 3 and 4.