Exercise 5.2-2 from Assignment 3

I solved this problem in class for two reasons. First, it was more difficult than I expected, in contrast to exercise 5.2-1, and second because it has some interesting counting tricks. In both problems you are asked to determine the probability that exactly so many people are hired by the hiring algorithm.

Exercise 5.2-1 has two questions: What is the probability of hiring exactly one person and what is the probability of hiring exactly $n$ people? These are counting problems. There are $n!$ ways to arrange $n$ people, that is, there are $n!$ permutations of $n$ distinct values. So computing the probability for the first question is done by dividing $n!$ into the number of permutations for which the algorithm hires one person. Remember that the first person interviewed is always hired so a little thought determines that the first person in the sequence must be the best and that everyone else after can appear in any order. Because there are $n - 1$ others than the best, they can appear in any one of $(n - 1)!$ orders. So the probability is $(n - 1)!/n! = 1/n$.

But the solution to 5.2-2 was trickier. It also relies on counting but figuring out how to count is more subtle. Here is my solution.

Assume that the first person in the sequence has rank $n - i$. We know that $i$ can’t be 0 because the first person can’t be the best. So $1 \leq i \leq n - 1$.

The first person interviewed is always hired. Let’s say that this person has rank $n - i$ where $1 \leq i \leq n - 1$. The person with the maximum rank, $n$, is always hired. And this means that the $i - 1$ people with rank strictly between $n$ and $n - i$ must appear after the maximum rank person. There are a total of $i$ people that are placed somewhere in the $n - 1$ positions after the first and such that the leftmost of those must be the best person. The other $i - 1$ will appear to the right of the best person and can be in any order.

So first we count how many ways we can select the $i$ positions in which to place the top $i$ people in $n - 1$ slots and discrete math tells us that there are $\binom{n - 1}{i}$ ways to do this.

Once those are selected, we next count how many ways we can insert the $i - 1$ people with ranks $n - i + 1$ to $n - 1$. They can be in any order so there are $(i - 1)!$ ways to do this.

Now we have the remaining $n - i - 1$ people to place and they can be in any order so there are $(n - i - 1)!$ ways to do this.
Putting it all together, recalling that the person with rank \( n - i \) is first, there are 
\[
\binom{n-1}{i} (i-1)! (n-i-1)!
\]
ways everyone else can be arranged. Let’s simplify that:
\[
\binom{n-1}{i} (i-1)! (n-i-1)! = \frac{(n-1)!}{i! (n-i-1)!} (i-1)! (n-i-1)! \\
= \frac{(n-1)!}{i!} (i-1)! \\
= \frac{(n-1)!}{i}
\]

Now we sum over all of the possible values of \( i \):
\[
\sum_{i=1}^{n-1} \frac{(n-1)!}{i} = (n-1)! \sum_{i=1}^{n-1} \frac{1}{i} \\
= (n-1)! \Theta(\lg n)
\]

This gives us a count of the number of permutations that cause the hiring algorithm to hire exactly two people. To find the probability that this happens, we divide the count by the total number of permutations:
\[
\frac{(n-1)! \Theta(\lg n)}{n!} = \Theta\left(\frac{\lg n}{n}\right)
\]

**Chapter 7: Quicksort**

I explained briefly how quicksort works. Instead of recappping that here I’ll point you to section 7.1. I next dealt with analyzing the best case and the worst case performance. Recall that when analyzing a sort we count the number of comparisons between array elements and let \( T(n) \) denote that count on an array of \( n \) elements.

The **QUICKSORT** algorithm calls the **PARTITION** algorithm once and itself twice.

In the best case, we assume that the **PARTITION** algorithm always divides the array into two equal or almost equal partitions. It’s easy to see that **PARTITION** performs \( n - 1 \) comparisons, each between the pivot and every other element. The recursive calls are then sorting arrays each of length \( n/2 \). This yields the recurrence relation:
\[
T(n) = 2T(n/2) + (n - 1)
\]

This fits the criterion for case 2 of the master theorem and we immediately get that
\[
T(n) = \Theta(n \lg n).
\]

In the worst case, we assume that the **PARTITION** algorithm always divides the array into one partition of size \( n - 1 \) and one of size 0. So now the first recursive call is sorting an
array of length \( n - 1 \) and the second an array of length 0. This yields the recurrence relation:

\[
T(n) = T(n - 1) + (n - 1)
\]

This doesn’t fit any of the criteria for the master theorem. Applying the substitution method shows that:

\[
T(n) = \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2} = \Theta(n^2).
\]

Section 7.3 suggests a very simple change to quick sort: Select the pivot randomly. Now we’re interested in the average case analysis. There will always be some arrangement of the elements in an array that causes quick sort to perform \( \Theta(n^2) \) comparisons but perhaps this is extremely rare. Average case analysis determines what should happen the vast majority of the time.

To do that, we need to perform a probabilistic analysis. We introduce the random variable \( X \), which is the number of comparisons performed in the \textsc{Partition} function of the sort. This is difficult to determine for a particular call to \textsc{Partition} so we will bound overall the number of comparisons done by all calls.

Let \( z_i \) denote the \( i^{\text{th}} \) smallest element in the array \( A \). Let \( Z_{ij} = \{z_i, \ldots, z_j\} \), that is, \( Z_{ij} \) is the set of elements \( z_i, z_{i+1}, z_{i+2}, \ldots, z_j \). The question will now be: When does the algorithm compare \( z_i \) and \( z_j \) for particular values of \( i, j \)? Note that this comparison happens at most once because it only happens if either is chosen as a pivot. And once a value is chosen as a pivot and used to partition, it is never considered again. The comparison might not happen at all because \( z_i \) and \( z_j \) could be on different sides of a pivot.

Let \( X_{ij} \) be the random variable corresponding to the event that \( z_i \) and \( z_j \) are compared. By what we just said above this will be either 0 or 1. This makes \( X_i \) an indicator random variable. It also means that adding these up gives us \( X \):

\[
X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}
\]

Because we don’t know whether \( X_{ij} \) is 0 or 1 we will have to use expected values:

\[
E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]
\]

Recall that the expected value of a sum is the sum of the expected values so we have:
\[
E \left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]
\]

By the lemma in section 5.2, the expected value of an indicator random variable is the probability that the event associated with it occurs so \( E(X_{ij}) \) is the probability that \( z_i \) will be compared to \( z_j \).

The only time two elements are compared is when one is the pivot. See the book on pages 182-183 for a more complete explanation of this. This means that the probability that two elements are compared is the disjoint probability that either is chosen as a pivot.

Prior to any element in \( Z_{ij} \) being chosen as a pivot, all of the elements are in the same partition because they are all either less than or greater than any pivot chosen previously.

Once an element in \( Z_{ij} \) is chosen as a pivot and \( Z_{ij} \) is partitioned, \( z_i \) and \( z_j \) will never be compared to each other again. So the only way they can be compared is if one of them is chosen as the pivot.

There are \( j - i + 1 \) elements in \( Z_{ij} \) so the probability of one being chosen is \( 1/(j - i + 1) \) so the probability of either \( z_i \) or \( z_j \) being chosen is \( 2/(j - i + 1) \).

So we have:

\[
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]
\]

\[
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}
\]

To continue, let’s change the index variable on the inner sum. Introduce a new variable \( k \) and let \( k = j - i \). The inner sum’s lower limit, which is \( j = i + 1 \), can be rewritten as \( j - i = 1 \). The upper limit, which is \( j = n \), can be rewritten as \( j - i = n - i \). Now substituting \( k \) for the \( j - i \) that occurs in the lower limit, the upper limit, and the summation term, we get:

\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1}
\]

We’ll perform one more change on the inner sum to make it easier to evaluate. Change the upper limit from \( n - i \) to \( n \). Doing so increases the number of terms being summed...
and so increases the sum meaning that instead of equality we will now have that the new sum is larger than the old one:

\[ \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \]

The inner sum now looks familiar, especially if we move the 2 outside. Then we get:

\[ \sum_{k=1}^{n} \frac{2}{k} = 2 \sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg n) \]

Putting that \( \Theta \)-term inside the outer sum gives us the expected number of comparisons:

\[ E[X] < \sum_{i=1}^{n-1} \Theta(\lg n) = \Theta(n \lg n) \]

Because it’s a less-than relationship we would normally have to show that \( E[X] \) is greater than some sort of \( n \lg n \) term to establish that it is exactly \( n \lg n \). We will do this when we establish a lower bound on the number of comparisons needed to sort.

At this point I summed up our discussion of divide-and-conquer algorithms and turned next to….

**Chapter 15: Dynamic Programming**

When we use the divide-and-conquer approach to solving a problem, we first divide the original problem into a collection of sub-problems and then we individually conquer, that is solve, each sub-problem. The last step, if necessary, is to combine the solutions to the sub-problems into a solution for the original problem.

I want to focus on the conquer step. We solve the sub-problems independently of each other and the important word here is “independently”. We can solve them that way because they have nothing to do with each other. When merge sort divides an array into two halves, how the first half gets sorted has nothing to do with how the second half gets sorted. In fact, they could be sorted in parallel.

This independence of sub-problems is a hallmark of a problem amenable to solution by a divide-and-conquer algorithm. As you might expect, that independence doesn’t always exist.

I started with the problem of computing the \( n \)th Fibonacci number. Let \( F(n) \) denote that number. By the definition of Fibonacci numbers, there are two base cases: \( F(1) = 1, F(2) = 1 \). For \( n \geq 3 \) we have the recursive definition \( F(n) = F(n - 1) + F(n - 2) \).
Computing $F(n)$ involves solving, that is computing, the sub-problems $F(n - 1)$ and $F(n - 2)$. But these are not independent. Computing $F(n - 1)$ means computing $F(n - 2)$ and $F(n - 3)$. Computing $F(n - 2)$ means computing $F(n - 3)$ and $F(n - 4)$. The sub-problems share a sub-sub-problem: $F(n - 3)$.

In fact, if you write the straightforward recursive method for Fibonacci, you quickly realize that it takes far too long to compute even for small values of $n$.

As we discussed in class, the right way to do this is to compute not from $F(n)$ down to $F(1)$ but from $F(1)$ up to $F(n)$.

Another way is to use the top-down approach but as you compute each value $F(k)$ for the first time, store the result in an array. After that simply look up the value rather than recomputing it. This is called memoization.

These two approaches when combined describe very briefly the dynamic programming approach. We will solve problems bottom-up and while doing so store solutions to sub-problems in a table.

I introduced the concept of an optimization problem. These are problems where you are trying to compute an answer that is either the maximum or the minimum. For example, in the shortest path problem you are trying to minimize the distance traveled in going from one place to another. In a network flow problem, you are trying to maximize the number of packets you can push through.

Our first optimization problem is the checkerboard profit problem. You are given an $n \times n$ checkerboard where each square has a positive payoff value. Starting at a square of your choice on row 1 (at the bottom), you are allowed to move to a square in the next row up that is above or diagonally adjacent to your current square. When you land on a square you earn its payoff. The problem is to discover the path from the first row to the top row that maximizes your total profit.

Let $p[i, j]$ denote the payoff of landing on square $i, j$ and let $q[i, j]$ denote the maximum profit that one can earn among all of the paths that end at square $i, j$.

We can define the computation of $q$ using a recurrence relation:

$$
q[i, j] = \begin{cases} 
0 & \text{if } j < 1 \text{ or } j > n \\
 p[i, j] & \text{if } i = 1 \\
 p[i, j] + \max\{q[i - 1, j - 1], q[i - 1, j], q[i - 1, j + 1]\} & \text{otherwise}
\end{cases}
$$

Computing $q$ with a top-down recursive method results in an algorithm requiring $O(2^n)$ time for an $n \times n$ board. A better way is to compute the maximum profit of each square from the bottom up.
For example, if the payoff matrix is:

\[
p = \begin{bmatrix}
4 & 8 & 2 & 3 & 5 \\
2 & 7 & 5 & 10 & 2 \\
3 & 2 & 4 & 1 & 5 \\
6 & 5 & 7 & 7 & 9 \\
8 & 7 & 5 & 9 & 8
\end{bmatrix}
\]

We can compute the values of \( q \) starting in row 1 by copying the values in row 1 of \( p \). For row 2 of \( q \) we use the values from row 1, and so forth. When we're done we have the profit matrix:

\[
q = \begin{bmatrix}
31 & 35 & 35 & 36 & 38 \\
20 & 27 & 25 & 33 & 25 \\
17 & 18 & 20 & 19 & 23 \\
14 & 13 & 16 & 16 & 13 \\
8 & 7 & 5 & 9 & 8
\end{bmatrix}
\]

\[q^* = 38\]

So the maximum profit we can earn is 38 (square 5,5). Tracing back we see that the path that got us there went as follows: 1,4; 2,4; 3,5; 4,4; 5,5.

Next week, we'll dive into chapter 15.