Problem Set 7 - Math

Question E

What a difficult question. I think combinatorics and maths will forever be a difficult concept for me to wrap my head around. I started this question by trying the strategy that was covered in the lectures, where we count the total number of permutations where we win, and divide it by the number of permutations possible for drawing cards, but it really wasn't getting me anywhere - I was finding it really difficult to reconcile the fact that we have different length'ed permutations, some that would end prematurely due to a win or a loss - and how to count those into my final count.

The different lens approach I had to take was rather, thinking of playing the game step by step, and computing the probability of winning at each step. If we break down each game step (i.e. having chosen i elements), to, having chosen i elements such that j is the last card face chosen, we can compute the probability of winning at that state (i, j) by just:

- Probability of getting to state (i, j) * probability of choosing j again as the next card.

This, as I noticed, was starting to look like a 2d DP - and it was… If we try and calculate the probability of getting to state (i, j), it gives us our recurrence onto prior states. The probability of reaching the state (i, j) would just be:

$$p(i,\ j) \ = \ \sum_{k=1}^{j-1} p(i\ -\ 1,\ k) \ \ * \ \frac{count[j]}{n-i+1}$$

We only need to sum up all cases where the previous element is less than j, since if a previous element was equal to j and we are choosing j again, we would win - and if a previous element was greater than j, we would have lost.

Everytime we calculate this probability, we would then multiply it with the probability of choosing j again to get the probability of winning at that state p(i, j), and add that to our final sum of probabilities. Probability of choosing j again would be given by:

$$\frac{count[j]-1}{n-i}$$

Finally, in regards to implementation specific stuff, I had to keep a 2d array that stored the cumulative sum of probabilities, such that every csumdp[i][j] stored the sum of probabilities of choosing i elements with the last value < j. This could then be used to calculate the next p state in this line:

```
        p[i][j] = modmul(csumdp[i-1][j-1], modmul(c[j], inv[n - i + 1]));
```

Looking back, p didn't actually relate back explicitly to any previous value of p, so I could have probably made it temporary, and kept csumdp as my only dp cache.

Otherwise, very hard question, took me a few days, but done!