Brad Hoffman

Assignment #1

January 16, 2019

Problem 1

The tensor product, also known as the Kronecker product is the multiplication of each element of a matrix *A* by all of matrix *B***.** An example can be seen in the Assignment 1 description, which I have copied below:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \otimes \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{1,2}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \\ a_{2,1}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{2,2}\begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} \end{bmatrix}$$

At first, this problem seemed really intense because there was no easy way to get the proper indices for the result matrix based on the the indices being used to calculate the sum. I spent a good amount of time trying different combinations of indices, until I decided to search the internet. Upon doing so, I realized that numpy actually has a function for this *np.kron()* where you supply two matrices and it will spit the result out for you. That was too easy, so I took a crack at fixing my indices again. The problem ended up being that I was not looping through the rows in the correct order. I had nested for loops in the order of: matrix a rows, matrix a columns, matrix b rows, matrix b columns. The correct order turned out to be matrix a rows, matrix b rows, matrix a columns, matrix b columns, and that allows for an easier way to index the resulting matrix. On the right, you can see the randomly generated matrices, A and B, the numpy.kron() result, and the result from the non-numpy implementation.

```
Matrix A
[[1, 3], [4, 4]]


Matrix B
[[5, 4], [0, 1]]


Numpy kron() function
[[ 5  4 15 12]
 [ 0  1  0  3]
 [20 16 20 16]
 [ 0  4  0  4]]


Non-Numpy implementation
5 4 15 12

0 1 0 3

20 16 20 16

0 4 0 4
```

Problem 2

This problem was a little easier to implement, but still took me a while to figure out how to properly use the stack implementation. Essentially, a string is given and the goal of the program is to count the number of valid substrings. In this case, our strings only contain parentheses, so the goal is to count the max length for the number of valid parentheses that are consecutive.

When using a stack, it first needs to be initialized with a -1 to serve as our base for the first index. The max length also needs to be initialized to 0. After that, we can begin iterating through each character in our string. If the character is a “(“, then we can go ahead and append the current index. If the character is not, then we need to pop from the current index. After doing so, if the stack is empty, then we need to append the current index to serve as the new base. If the stack is not empty, we take the max value of the current max length and the current index minus the value on the top of the stack.

The problem I was having arose from using the popped value as the one I subtract from the current index to generate my max values. Once I got that sorted out, the program worked flawlessly, as can be seen in the results below.

Problem 3

If this problem was to simply display the k most frequent words given a list of words, this problem would’ve taken a few minutes. But since we had to print all words that repeated the same amount of times *in the order of occurrence in the string*, the problem was a lot more complicated. First off, it is easy to get a list of words by count from a list through using python’s Counter package and the *most_common()* method. Once that was done you could extract the k top word pretty easily. But first, you have to check whether there are any other words that have a similar count. Luckily, *most_common()* will return the list of counts in order from highest to lowest. To check if they’re the same, loop through the words starting from 0 and add the words to a new list. If the current word has a lower count than the previous word, go ahead and stop because there is no need to continue. That’ll give you a list of words with the highest count.

Now, in order to sort them by index, we have to take the words, compare them to the original list and store their first occurring indices in a new array. Sort the indices in order to get first occurence, then create a new array using those indices to grab the associated words. You then have a list of words with the highest count in order of occurrence in the original string.

The tricky part comes into play when you have to satisfy the k value and also displaying all words that have that same count. So if you have 3 words with a count of 2, 2 words

with a count of 1, and a k value of 4, you should be displaying all 5 words, to satisfy the constraint of displaying all words with equal value. That way one or more words aren't left out. To do this, I took the list of words that have the highest count and are in order of occurrence in the string, and created a *final_words* array. If the length of this array was more than my *k* value, I could end. If it was not, then I'd have to keep searching until the length of this array was.

The next steps include taking the words from the *final_words* array, and removing them from the original list of words as well as the list of word counts. Once that was done, I could repeat the process all over again until my length of the *final_words* array was greater than or equal to my k value. Results can be found below. Both images have a k value of 4.

```
['love', 'i', 'coding', 'coding', 'i', 'love', 'Deep', 'learning', 'i', 'love', 'coding']
['love', 'i', 'coding', 'Deep', 'learning']
```

```
['do', 'you', 'love', 'you', 'love', 'me', 'me', 'i', 'love', 'you', 'do']
['you', 'love', 'do', 'me']
```

Problem 4

Given our arrays A and B of size n and m, we had to generated a new matrix C of size n x m. That was easy to do. From there, we had to find a rectangle within the matrix C that had the largest sum. The rectangle could be 1 cell and up.

I implemented a solution using dynamic programming, so the process starts by creating a left bound and a right bound at the 0th index. From there, the right bound will increment until it reaches the end, at which, the left bound will increment and the right bound will be reset to the left bounds new index. This occurs until both the left and right bound have met at the other end of the matrix. Along the way, the goal is to use dynamic programming to sum our rectangles in order to find the one with the highest some. At each starting point, we will take the current column of values, then when the right bound is incremented, the new column is added to the current column. This array is then sent through the kadane algorithm in order to determine the max value. Normally, we would keep track of the left, right, up, and down bounds so that we could plot out the actual rectangle within our matrix, but this assignment only called for the max value. That means the kadane algorithm is a simple comparison of the current max value, which is initially set to a very small number, and the values of the array added up thus far. This is done through looping through our temporary array of values that are

generated with each increment of bounds. In the end a max value will be generated, that can be seen below for the input given in the project.