Calvin Todd
CS 325
5.28.21

Portfolio Assignment:

I.

1.

i. **Description:** For the first implementation of the k-palindrome method we are going to use a brute force approach. We will iterate over every possible combination of the k-palindrome until we either find a k-palindrome or iterate through every possible option ensuring that there is no palindrome.

ii. **Pseudocode:**

```
checkPalindrome_1(string, k):
    if k is 0:
        if string.reverse() == string:
            return True
        else:
            return False
    else:
        if string.reverse() == string:
            return True
        else:
            for letter in string:
                truth = checkPalindrome_1(string – letter, k – 1)
                if truth is True:
                    return truth
    return truth
```

2. See method *checkPalindrome_1* code in *Palindrome.py*
3. The time complexity is $\boldsymbol{O}(\boldsymbol{n^k})$ where n is the number of letters in the string, and k is the maximum number of letters allowed to be removed from the string.
4.

i. **Description:** Since this problem has an optimal solution and overlapping sub-problems, we are going to use a Dynamic Programming approach with an array that is (length of the string) by (length of the string). The x-axis of the array will represent the string, while the y-axis will represent the reversed string. Each location, (i, j) in the array will represent the minimum amount of characters we must remove from string (of up to length i) and reversed string (up to length j) to create a Palindrome. Finally, we will check to see if the final position in the array is equal to or less than *k*, representing that this is indeed a *k-palindrome* or not. It is important to note that a string of no characters is indeed considered a palindrome for this solution.

ii. **Psuedocode:**

```
checkPalindrome_2(string, k):
    reversed = string.reverse()
    n = length of string
    dp_array = [[0 for range(n + 1)] 0 for range(n + 1)]
    for j in dp_array:
```

```
                    for i in j:
                        if string[i] or reversed[j] is 0:
                            dp_array[j][i] = i + j
                        if string[i] == reversed[j]:
                            dp_array[j][i] = dp_array[j-1][i-1]
                        else:
                            dp_array[j][i] = min(dp_array[j-1][i] + 1,
                            dp_array[j][i-1] + 1)
            if dp[n][n] <= 2*k:
                    return True
            else:
                    return False
```

5. See method *checkPalindrome_2* code in *Palindrome.py*
6. The time complexity of this method is $\boldsymbol{\Theta(n^2)}$ where *n* is the number of letters in the string.
7. Yes my second implementation is an improvement in in time complexity as $\boldsymbol{\Theta(n^2)}$ < $\boldsymbol{O(n^k)}$
8. Since this problem has overlapping sub problems and an optimal solution, I realized I could use Dynamic Programming to solve this. I chose a bottom up approach utilizing an array to help me store answers so there was no need to recalculate certain calculations.

II. Problem B: Pattern matching

1.

i. **Description:** We will use a bottom-up Dynamic Programming approach to solve the pattern matching problem. By creating a 2d array of string by pattern, we can slowly build an array of Boolean values of whether the pattern at the location in the 2d array matches the string at the same location in the 2d array. Through the logic we can see that if the characters match, or one character is the '?' symbol then the problem we are now answering has already been solved at the location one back and one up in the array. Further, if the symbol is '*' and either one spot up in the array, or one spot back in the array, is True, then that spot in the array is also True. By this logic, we can return the last spot in the array, and that will be the answer to whether the pattern matches the string.

ii. **Psuedocode:**

```
patternmatch(string, p):
        n = length of string
        m = length of p
        dp_array = [[0 for range(m + 1)] for range(n + 1)]
        for j in range(n + 1):
                for i in range(m + 1):
                    if i or j are 0:
                        if i and j are 0:
                                dp_array[j][i] is True
                        else if p[i-1] is '*':
                            if dp_array[j-1][i] or dp_array[j][i-1] is
                            True:
                                dp_array[j][i] is True
                        else:
```

```
                                        dp_array[j][i] is False
                                else if p[i-1] is the same as string[j-1] or p[i-1] is
                                '?':
                                        dp_array[j][i] is dp_array[j-1][i-1]
                                else if p[-1] is '*':
                                        if dp_array[j-1][i] or dp_array[j][i-1] is
                                        True:
                                            dp_array[j][i] is True
                                        else:
                                                dp_array[j][i] is False
                                else:
                                        dp_array[j][i] is False
                        return dp_array[n][m]
```

2. See method *patternmatch* code in *Patternmatch.py*
3. The time complexity of this method is $\boldsymbol{\Theta(n * m)}$ where *n* is the length of the string and *m* is the length of the pattern.
4. I again selected Dynamic Programming approach since I had an optimal solution and overlapping subproblems.

III. Problem C: Get Tesla

1.

i. **Description:** To solve this I am going to use a combination of Dynamic Programming and a maximum spanning tree variant. By making the greediest choice available, I can confirm that I am keeping the highest score possible, and losing as little HP as possible. By keeping a second array of the scores at certain points, then I can utilize backtracking as well to make sure my greedy choices lead to an optimum solution.

ii. **Pseudocode:**

```
getTesla(M):
        score_array = [[None for i in range(len(M[j]))] for j in
        range[len(M)]
        m = length M
        n = length M[m-1]
        last vertex = ( n-1, m-1 )
        lowest = M[0][0]
        score = M[0][0]
        current = (0, 0)
        neighbors = []
        while last_vertex is not in neighbors:
                if right neighbor in range:
                        negihbors.append(right neighbor)
                        score[right neighbor] = current + right neighbor
                if bottom neighbor in range:
                        neighbors.append(bottom neighbor)
                        score[left neighbor] = current + left neighbor
                next = max(neighbors)
                if next < lowest:
                        lowest = next
                current = next
                score = current
```

Calvin Todd
CS 325
5.28.21

```
score = score + last vertex score
if score < lowest:
        lowest = score
if lowest > 0:
        return 1
else:
        return abs(lowest) + 1
```

2. See method *getTesla* code in *GetTesla.py*
3. The time complexity of this method is $\mathbf{O}(\boldsymbol{n^2})$ where *n* is the number of vertexes in the graph.
4. I used a Dynamic Programming Top Down Approach to store already calculated values. Further I used a greedy approach with a modified Prim's Algorithm (I used more the theory behind the algorithm than the algorithm itself) to ensure I was making the optimum choice for the problem's goals. My algorithm's time complexity should have an upper bound of $\boldsymbol{n^2}$, but rarely approach it.

Debriefing:

1. This was about 6 hours for me from start to finish. I feel it was a perfect time amount for a end of the quarter homework/portfolio assignment.
2. Moderate! I was challenged but not overwhelmed. I felt I had all the tools I needed to solve the problems, through I had to think through and review a bit.
3. 90%
4. I liked this assignment. I felt the questions were good examples of the major concepts in this class. Particularly the third getTesla question. I would be very interested to see a list of answers once the grading period was over to see how my approach has stacked up against some of the best found approaches!