CompSci 2CO3:
Assignment 3

Mohammad Omar Zahir - zahirm1
Talha Amjad - amjadt1

April 13, 2021

Question 1:

Case 1: Prims Algorithm

Prim's algorithm creates an MST by starting at any random vertex, and looking at all the edges that are connected to that vertex and traverses them based on the smallest weight. A companion 'visited' array is also maintained. From the starting vertex, we traverse the smallest weighted ege to get to the resulting vertex, which is then added to the list of visited vertices. We will then look at all the vertices that we have visited so far and find the smallest weighted edge from those that we have access to. We will continue picking vertices from edges in this manner but with one condition: the resulting vertex from following an edge must not be in the visited array, that is to say that we have not yet visited it. This ensures that we have no repetitive edges that are pointing to the same vertex, and ensures that we are always obtaining the minimum weighted edge path. We will continue this minimum edge searching for all our vertices, and assuming each component is connected, we will eventually be able to reach every node in the MST with the minimal edge path.

Because Prim's algorithm requires that a graph is connected, as the graph is traversed from node to node, a slight modification in the algorithm will have to be made. For the most part, the algorithm will stay the same, as we can use Prim's algorithm to determine the Minimum spanning forest simply by repeatedly applying the process above. Once we have traversed all the edges in a component, we will then begin to get edges that we have already visited. In this case, when the only edges left take us to the same vertices, we will then compare our 'visited' array with the set of all the vertices in the entire graph. If there are vertices that are not present in our visited array, we will then go to any of these such vertices, and apply Prim's algorithm to them as usual. This continues until both our visited array, and the total vertices in the array represent the same vertices, that is to say we have visited all the vertices in the entire graph, obtaining each individual component's MST.

Case 2: Kruskal's algorithm

Kruskal's algorithm creates an MST by simply looking at all the edges of the graph, and adding these edges one-by-one by smallest to largest. We will continue adding edges in this method until we reach a vertex from that edge that would create a cycle in the MST, and we do not add any such edge. Continuing in this manner, we will eventually be able to obtain a minimum path that should traverse all vertices in the graph such that there are no cycles.

If we were to use Kuskal's algorithm to obtain the minimum spanning forest for a disconnected graph, we would simply just apply the algorithm, without any modifications, regardless of the fact that there are independently connected components. Because Kruskal's algorithm looks at all the edges of the graph, and simply ignores the ones that create cycles, we can be assured that the algorithm will reach all the components in the graph, even if they are part of a different component. Therefore we can confirm that an MST for each of these distinct components will be obtained, resulting in our Minimum spanning forest.

Question 2:

There are two situations that could arise from such a possibility.

The first situation is when the node that we are removing is not a node in the MST that we are given for the old graph. If that is the case, that would mean that the old MST is still a valid MST for the new graph without the node, as all things that were connected previously are still connected.

Alternatively, the other situation arises when the node that is to be removed is already in the MST, meaning that we would have to remove that node from the MST, which would not be a correct MST for the updated graph. Because we have removed a node from the MST, and since by definition there are no cycles in an MST, that would mean that we now have two separate connected components in the MST. Since we know that the node that was removed did not disconnect the graph, we know that there must exist at least one other weighted edge that would connect to nodes in both connected portions of the graph. We can then use kruskal's algorithm to find such a weighted edge.

Question 3:

For this question, we will use the logic that if we have two sets of vertices, namely S and T, and assuming that such an edge exists that will connect these two subsets of vertices, that would mean that there is a shortest path E that connects two vertices that are in S and T respectfully, such that it has a minimum edge weight. Because we know that such a path has to exist, if there does exist another edge connecting a node in S and T, it would have to be greater than or at least equal to our minimum edge E.

To determine the shortest path , we can iterate over all the edges of a single subset, S, and look at all the edges that are directly connected to it. We will assume that we have access to the edges that the node is pointed out towards, and not into it. We will go over all such edges and check whether the node that is connected to this edge from the other side is in the subset T. If it is, we will then add this edge to the list EDGES that is being maintained to store all such possible edges. We will then repeat this process for all the edges of all the vertices in the subset S. After this is done, we will then look at all the edges from the list of edges we have, and simply return the edge with the shortest edge weight. If two edges have the same minimum edge weight, returning either of them is sufficient.

The following pseudocode is presented for the algorithm outlined above. In this pseudocode we assume we are using the directed edge data type from the lectures, and we assume that each vertex has a list of all the edges where the vertex is the .from() vertex for that edge.

```
procedure minEdgePath (subset S, subset T)
        EDGES = []
        sets = [S,T]
        for vertex in S
                for edge in vertex.edges()
                        if edge.to() in T:
                                EDGES += [edge]
        min = Infinity
        minEdge = null
        for edge in EDGES
                if edge.weight() < min
                        minEdge = edge
        return minEdge
```

We know that this algorithm runs in O(E+V) because although there are two nested for loops and while the first one iterates over all the vertices and the second iterates over all the edges, this is done independently. As such, because we know that O(E+V) is smaller than O(E log V), the algorithm is within O(E log V).

Question 4:

To determine a monotonic path from a node s to every other node in a graph, there would be two situations, namely decreasingly and increasingly monotonic, with the algorithms outlined below.

Decreasing Path

Starting from our source node, we will look at all the outgoing edges. We will begin by adding these edges by minimum edge weight, along with the connected vertex in a tuple in the form (e,v) which will be added to a minimum priority queue. We will continue this process for all the outgoing edges for the source until there are none left. Once this is done we will access the first element from our priority queue, since it is sorted by ascending order, meaning the smallest element is at the front. We would pop this element and then perform the same procedure on this based on the following two criteria. The weight of the edge that is pointed away from that edge must be smaller than the incoming edge for that vertex(monotonic). This would ensure a monotonically decreasing path. The second criteria that we must ensure is that when checking a new possible edge for a path, the new cost of going to the edge must be smaller than the current cost of going to the edge. This is accomplished through the relaxation of the outgoing edges from the popped vertex in the priority queue, which should be done in increasing order. We then add each of these outgoing edges into the priority queue as we relax each one. The relaxing is facilitated by the default cost of going to each vertex as infinity and the cost is updated by checking the sum of the weight of the edge and the node that is incoming from it. If it is smaller than the cost that is already there, that is the new shortest path. If it is larger, that means that a shorter path exists to that node from somewhere else, so no changes should be made.

Increasing Path

A very similar procedure will then be done for strictly increasing, but certain things will be done in the opposite manner. Firstly, rather than sorting in increasing order, we would sort the outgoing edges in decreasing order, meaning we would look at the largest edge weight first, and continue moving down from there. Similarly, we will do the same when popping out from the priority queue, removing them in decreasing order. The other thing to alter would be that in the condition where we check that the edge after is smaller than the edge before, we will instead check if the edge after is larger than the edge before, as we are now trying to achieve a monotonically increasing path. Furthermore, we must also relax the current node's outgoing edges in decreasing order.

To keep track of the paths at the end we will be maintaining an array of nested arrays where each nested array at a specific index will be the minimum path to the vertex at that index, that is to say that the indexes are the same values as the node. By default the value at each index will be an empty list. While we are traversing through the graph, it is assumed that we will be keeping a track of the path that we have thus been traversing to get to each vertex. Whenever we relax edges, and if the relaxation results in an edge path that is smaller than the current cost at that node, we will simply update the index in the list of paths that is being maintained at the value of the vertex, meaning we will go to index 5 for vertex 5. Once there, we will update the path with the minimum path of edges that we have now obtained from the process of relaxation in the array for that vertex. This way, the path stored in the list of paths for each vertex will be guaranteed to be the minimum cost path to that edge. At the end, we will have traversed all the edges that can be traversed, and will have determined the smallest possible paths for all such vertices as well. Because we would have been updating the paths for each vertex in the list of paths, we will now have the smallest paths in the list which we can now return as a list of lists(paths).

Question 5:

Key-Indexed counting is a type of sorting algorithm that is very effective when dealing with small integers. The way this algorithm works is that the first step is to count the number of occurrences that a key value has. Then, using a for loop, we can look at the key to access a value in count[] using an int array count[] and also increment that value. We can then use count[] to compute the starting index positions for the items in a given key. However, in order to get the starting index positions for each key, we can sum the # of occurrences of small values in the given key. Furthermore, looking at the second for loop in the algorithm, we can take a look at each key value represented by r. We can see that the sum of the counts for each value that is less than r + 1 would be equal to the value when the key values less than r + count[r] are summed up. This allows us to transform count[] into an index table where data can be sorted in the right order and represents the starting index positions.

This table is an example that shows where each character is positioned in the given array.

i	0	1	2	3	4	5	6	7	8	9	10	11
L[i]	a	b	c	f	f	b	b	d	f	b	e	a

Using the example table above. As the algorithm runs through the first for loop, the # occurrences for each key are counted and stored as shown in this table.

r	a	b	c	d	e	f
count[r]	2	4	1	1	1	3

Furthermore, this is what the table becomes when the algorithm runs through the second for loop. The # of a's are 2 but then the count of b's becomes 6 because of 2(# of a's) + 4(4 is the # of b's as seen in the table above). This pattern is continued for the rest of the keys in the table.

r	a	b	c	d	e	f	-
count[r]	0	2	6	7	8	9	12

For the most part we will use the same algorithm that is presented in the textbook, especially the first two for loops, where we set up the count array with the corresponding frequency values. The difference comes in the final two for loops with the introduction of the auxiliary array, which we will try to eliminate through the algorithm that we are doing below. In the final for loop, we will look at the main array L and set the first value equivalent to the key value that will be stored correspondingly in the sound array. At the beginning, this would mean that the first value in the count array, which would be a in the example from the slides, should be put in that index for the main array. The IndexValue() in the second for loop will represent the arbitrary values for the things that are being ordered. We assume that a one-to-one mapping for such things has been made prior to this, like a = 0, b = 1, and so on, which facilitates the IndexValue() calling so we are indeed putting the value corresponding to the index and not the index itself. We then increment this value at count and stay at the same index index until the following condition mentioned in the if statement is met. We will only increment the index at count that we are looking at when the value at that index is the same as the value in the next index. This would mean that we have reached the maximum amount of values that we must enter for the specified character, and we must start adding the next value. This is also the only situation in which we will increment the i to get to the next index as a result.

While the original algorithm uses the auxiliary function for the movement of the values and moving them back, which increases our space complexity, because we are avoiding the use of an auxiliary function and are updating the values directly in the manner specified above, we will be performing the algorithm in constant memory.

```
procedure nonStableKeyIndexedCounting(L, R)
    count = []
    for i in L:
      count[i + 1] = count[i + 1] + 1

    for (r = 0; r < R; r++)
      count[r + 1] += count[r]

    for (i = 0; i < R;)
      L[count[i]] =  count.IndexValue(i)
      count[i] = count[i] + 1
      if count[i] == count[i + 1]
        i += 1
```

Question 6:

The following algorithm determines whether two strings are cyclic rotations of each other. This is done by first checking if the lists are equal in length, done in constant time. We then double one of the strings, as this will ensure that the second string would be preserved within the first if they are in fact cyclic rotations. For instance, if we are to use the word "hello" and "elloh" from the question, we would double the word example to obtain "h**elloh**ello" and we can see the word "**elloh**" within it since it is a cyclic rotation.

We will then loop over the length of this doubled string, which would be a for loop for 2n, which is still within O(n) complexity. We will then keep track of the same sequence of letters, one-by-one, with the aims of having this counter be equivalent to the length of the string. If it is equivalent, that would mean that the correct sequence of all the letters was found and the strings are cyclic rotations of each other, otherwise it is not. All of these comparisons are done in O(n) time in the worst case as there are only constant time operations like if statements and array indexing within the for loop, which do not add to the complexity. Thus this algorithm performs this operation in O(n) time.

```
public static boolean cyclicRotation(String s1, String s2){
    if (s1.length() != s2.length()){
        return false;
    }
    String ds = s1 + s1;
    int t = 0;
    int len = ds.length();
    for (int i = 0; i < len; i++) {
        if (t == s2.length())
            break;
        if (ds.charAt(i) == s2.charAt(t))
            t++;
        else
            t = 0;
    }
    return (t == s2.length());
}
```

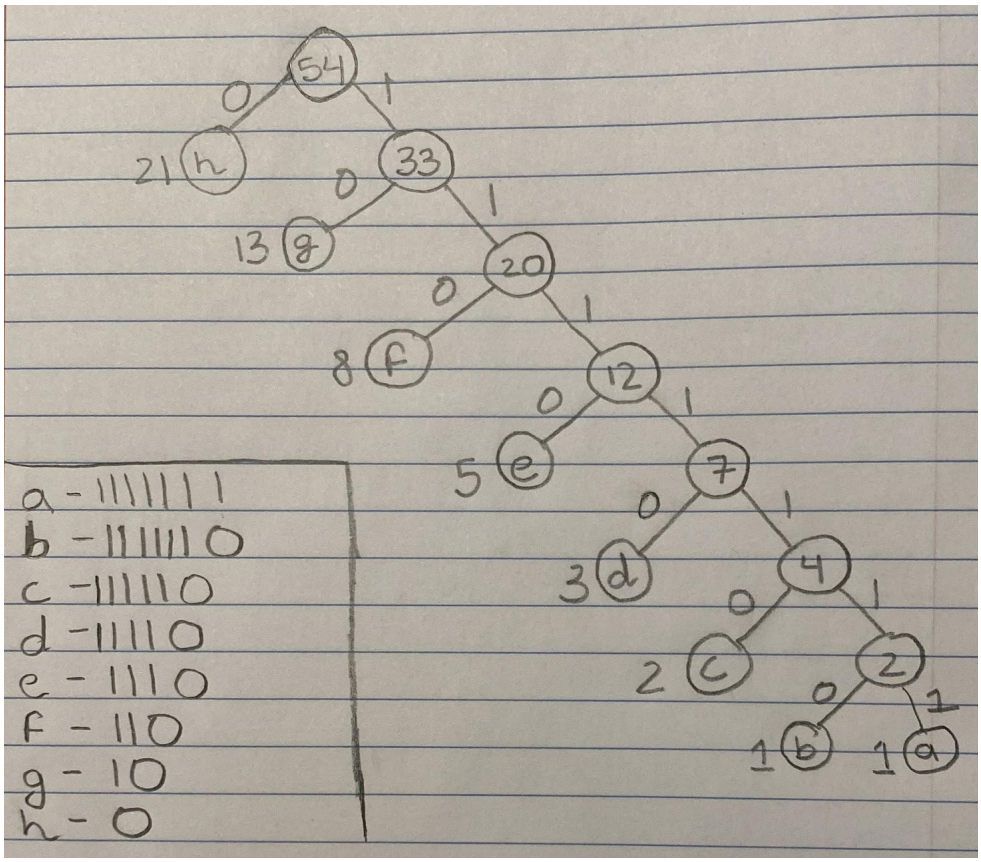
Question 7:

We can see that the following algorithm runs in O(n). We first use the KMP algorithm, as given to us from the question, to determine the indexes of the occurrences of one string to the other. Because this is not inside any form of for loop, this would be a constant time addition of O(n). After some declaration statements, which take O(n) time as it loops over the length of the array, we can see that there are only if statements within the for loop that take constant time, so they do not add to the complexity. There are also array indexing operations within the for loop, but they also take a constant amount of time so they do not add to the complexity of the entire algorithm. At the end, because we used the KMP algorithm in O(n) time, and the below algorithm, which independently takes only O(n) time in the for loop, that would mean that the overall complexity of the entire algorithm to determine the starting index of the longest running tandem repeat would still be in O(n).

```
public static int longestRepeat(String s1, String s2) {
    int[] arr = KMP(s1, s2);
    int diff = s2.length();
    int maxCount = 0;
    int count = 0;
    int currStartIndex = 0;
    int maxStartIndex = 0;
    boolean start = true;
    for (int i = 0; i < arr.length-1; i++) {
        if (arr[i + 1] - arr[i] == diff){
            if (start) {
                start = false;
                currStartIndex = i;
            }
            count += 1;
        }
        else {
            start = true;
            count = 0;
        }
        if (count > maxCount) {
            maxCount = count;
            maxStartIndex = currStartIndex;
        }
    }
    return arr[maxStartIndex];
}
```

Question 8:

i) The following optimal Huffman code can be observed for the given values below.



ii) From the given Huffman codes that can be seen above, we can start to see a pattern where the *k*th position in the given sequence of *n* values corresponds to a Huffman code of

1*(n - k) 0

That is to say that there will be n-k many 1's followed by a single 0 at position k. The only exception is that of the first element, which will have

1*(n-1)

Or (n-1) many 1's as it's Huffman code.

We know that this pattern will follow for all the members of the Fibonacci sequence due to the characteristic of the Huffman coding of grouping only those values with the same frequencies. Because we know that the Fibonacci sequence is an infinitely ascending sequence with no duplicates, with the exception of the first two elements, we can confirm that every new element will be added as a separate left-leaning branch within the larger tree, as is the case in the example. This would verify that the pattern is maintained for all the members of the Fibonacci sequence, assuming that all the left leaning edges will have values of 0 and the right-leaning edges have values of 1.