Homework 4

Raymond Lin, 304937942

June 3, 2020

Problem 1

a)

Terabytes of Data to Process	10	10	10	10	10	10	Totals
No reboots	8	4	2	1	0.5	0.25	15.75
1 reboot	8	4	-	8	4	2	26
2 reboots (optimal)	8	-	8	-	8	4	28

b)

Idea:
The idea of this algorithm is to use dynamic programming to store the optimal solution (in this case, the highest possible amount of data processed) by day in an array. Let's call this array, OPT. The value at $OPT[i]$ represents the highest possible amount of data processed by day i + 1 (assuming OPT is 0-indexed). The first thing to note is that the amount of data processed in one day is upper bounded by both the amount of data available (given in array x) and the amount of data our machine can possibly process after some days (given by array s).

The value at $OPT[i]$ can be calculated by taking the maximum between two cases. The first case, is if we reboot on the previous day, which means that our computer can process the maximum amount of data in a single day, given by $s[0]$. Then, we take the minimum between this value and the current available data on that day. Then, we sum this value with $OPT[i-2]$, skipping $OPT[i-1]$, because that was the day we rebooted. The second case, is if we do not reboot the previous day, meaning that the amount of data we can process is dependent on the previous day's maximum. We need to record the number of days since the previous reboot, and we can simply use a variable j. We take the minimum between this value and the current available data on that day. Then, we sum this with $OPT[i-1]$, because we didn't reboot on the previous day. The equation for $OPT[i]$ can be written as follows:

$$OPT[i] = max(OPT[i-2] + min(s[0], x[i]), OPT[i-1] + min(s[j], x[i])) \quad (1)$$

After coming up with the top-down DP approach, the bottom-up approach is also easily seen. We can simply build the OPT array by starting at the beginning. We fill the first two elements of the array with $max(x[0], s[0])$ and $max(x[1], s[1])$ respectively, because there is no point in rebooting at either of these steps. Then, we can build the rest of the array using the above equation.

This algorithm runs in $O(n)$, because we loop through the length of the input arrays once in order to build the OPT array. The last element in the OPT array is our solution.

Algorithm (Bottom-Up):

```
let x = array containing amount of data for each day
let s = array containing the profile of our computer system, with no reboots
let n = size of x = size of s // we assume x and s are of the same length
let OPT = our DP array, initialized with size n
let j = 0 // variable used to keep track of how many days since rebooting

for i in range(n):
  if i = 0 or i = 1:
    OPT[i] = max(x[i], s[i])
  else:
    let opt1 = OPT[i-1] + min(x[i], s[j])
    let opt2 = OPT[i-2] + min(x[i], s[0])
    if opt1 > opt2:
      OPT[i] = opt1
      j++
    else:
      OPT[i] = opt2
      j = 0

return OPT[n-1]
```

Problem 2

Idea:
The idea of this algorithm is quite similar to the sequence alignment problem that was gone over in lecture. However, there are a few modifications to note. First, it is important to notice that we cannot perform replacement, as we could with the sequence alignment. The rules of the problem state that we can only insert characters into the string to make it a palindrome, not replace them. Second, it is important to note that the sequence alignment problem has two input strings, while our current palindrome problem only has one. As a result, we must generate our own second string. Because a palindrome is defined as a string that can be read the same forwards and backwards, we can reverse our input string as the second string.

Taking note of the differences, we can come up with a DP algorithm. We can use a matrix to store the cost of inserting characters into our two strings. Let's call this matrix OPT, where one axis represents the characters of our input string, x, and the other represents the characters of $reverse(x)$. We can define the δ value to be 1 for simplicity. Then, each insertion to either string will increment the cost by 1. We do not have an α value, because we cannot perform replacements. After initializing our first row and column with the values $1, ..., n$, where $n = length(x)$, we can build OPT. Each entry of $OPT(i, j)$ is the minimum between $OPT(i-1, j)$, $OPT(i, j-1)$, and potentially $OPT(i-1, j-1)$ only if there is a match between $x[i]$ and $reverse(x)[i]$. This relation can be represented with the following equation.

$$OPT(i,j) = \begin{cases} min(OPT(i,j-1)+1, OPT(i-1,j)+1, OPT(i-1,j-1)) & if x[i] = reverse(x)[i] \\ min(OPT(i,j-1)+1, OPT(i-1,j)+1) & otherwise \end{cases}$$

Our answer is located at $OPT(n, n)$, because at this point, we have included insertions to x and $reverse(x)$ such that they match each other. It must be true that both are now palindromes. The value at that position represents the total number of insertions to both x and $reverse(x)$ such that they both match. This is true because, according to our equation above, we only add 1 to the current value $OPT(i, j)$ if we choose the previous element to be the element directly under it, $OPT(i-1, j)$, or the element directly to the left of it, $OPT(i, j-1)$. Thus, in order to get the number of insertions performed to x, we simply divide the value at $OPT(n, n)$ by 2. This is because x is the reverse of $reverse(x)$, and any insertion done to x must have a corresponding change done to $reverse(x)$ (not necessarily the exact same change, but a corresponding change). Thus, the number of insertions done to x must be the same as that of $reverse(x)$.

The time complexity of this algorithm is $O(n^2)$, because we construct OPT in polynomial time. Once we construct the matrix, we simply extract the last value in the matrix and perform a constant time operation to the value and return it. Constructing OPT takes $O(n^2)$ time, because its size is n^2. Axis 1 represents x, and axis 2 represents $reverse(x)$, and it is clear that they are both size n.

Algorithm (Bottom-Up):

```
let x = our input string
let y = reverse(x)
let n = length(x) = length(y)
let OPT = our DP matrix

// fill the first row and column with our values
for i in range(n + 1):
   OPT[0][i]=i
   OPT[i][0]=i
```

```
// fill our DP matrix bottom-up
for i in range(1, n+1):
   for j in range(1, n+1):
      if x[j]=y[i]:
         OPT[i][j] = min(OPT(i,j-1)+1, OPT(i-1,j)+1, OPT(i-1,j-1))
      else:
         OPT[i][j] = min(OPT(i,j-1)+1, OPT(i-1,j)+1)

// extract the last value and return its value / 2
return OPT[n][n] / 2
```

Problem 3

Idea:
The idea of this algorithm is to use DP to save the maximum cardinality of the independent set at the subtrees rooted at every node. We will have a DP array called m of length equal to the number of vertices in the tree. At each recursive call, we process the current subtree and return the maximum cardinality of this independent set at the current subtree. We must determine if we want to select the current root node to be in the independent set. If any of its children are also selected, then the root node cannot be selected. Thus, we also need to store in the DP array whether the root node of every subtree was selected to be in the independent set or not. We can simply use a boolean variable for this. Thus, each node's entry in the DP array ($m[i]$) contains a two tuple that contains the following: a number that represents the maximum cardinality of the subtree rooted at the node ($m[i][0]$) and a boolean that represents whether or not the node was selected to be in the independent set ($m[i][1]$). There are two cases. The maximum cardinality of the subtree rooted at a node can either be the sum of the maximum cardinalities of all its children or the sum of the maximum cardinalities of all its grandchildren plus itself. More precisely, it follows this equation:

$$m[i][0] = max(1 + \Sigma m[j][0] \; \forall \; j \; are \; grandchildren \; of \; i, \Sigma m[j][0] \; \forall \; j \; are \; children \; of \; i) \tag{2}$$

Then, in order to retrieve the actual set of independent nodes, we can perform a linear search on m, and whichever nodes have their boolean value ($m[i][1]$) as true, then it is in the independent set.

This algorithm runs in $O(|V|)$, because we are using recursion, which implies DFS and that we only visit each node once. Since the graph is a tree, we do not need to worry about the number of edges, because of the property that in all trees, the number of edges in a tree is always $|V| - 1$.

Algorithm:

```
let root = root node of our tree
let m = our DP array of size |V|
let indSet = ∅

// define a helper function that finds the maximum cardinality of the independent
// set of a particular subtree
def maxCard(node):
   if node = null:
      return 0
   if m[node] != null:
      return m[node][0]
   if node.children.empty():
      m[node] = (1, true)
      return m[node][0]

   // find the max cardinalities of the node's children and grandchildren
   let sumChildren = 0
   let sumGrandChildren = 1
   for v in node.children:
      sumChildren += maxCard(v)
      for w in v.children:
         sumGrandChildren += maxCard(w)

   if sumGrandChildren > sumChildren:
      m[node] = (sumGrandChildren, true)
   else:
      m[node] = (sumChildren, false)

   return m[node][0]
```

```
// call the helper function to fill m
maxCard(root)

// obtain the independent set
for each node, v, in the tree:
   if m[v][1] = true:
      indSet = indSet ∪ v

return indSet
```

Problem 4

Claim: Vertex Cover $\leq_p$ Hitting Set

Proof:
Suppose we have a black box that can solve the hitting set problem. Consider an arbitrary instance of the vertex cover problem on a graph $G = (V, E)$. If we can reduce the vertex cover problem into the hitting set problem, then we are done. If we ask whether or not the vertex cover problem on graph G can be solved with a set of size $\leq k'$, then we should also be able to ask whether or not the hitting set problem on graph G and some set of of subsets of G, $B = \{...\}$, can be solved with a set of size $\leq k'$.

We can do this by populating the set B with subsets each of size two nodes. Each of these two nodes in the same subset must have an edge connecting them. In other words, suppose *edge* $e = (u, v) \in E$ and $B_e = B_{(u,v)} = \{u, v\}$. Then $B = \{B_e\} \forall\ e \in E$. If we pass G and B into the black box for the hitting set problem, we can solve it.

Intermediary Claim: If G has a vertex cover of size k' and we transform the problem using the above method, then G with its set of subsets, B, has a hitting set of size k'.

Proof:

- If G has a *vertex cover* of size k', then edge $e = (u, v)$, and either $u \in$ *vertex cover* or $v \in$ *vertex cover* $\forall (u, v) \in E$.
- Then it follows that $(u, v) \cap$ *vertex cover* $\neq \emptyset\ \forall (u, v) \in E$
- This is equivalent to saying that $B_e \cap$ *vertex cover* $\neq \emptyset\ \forall B_e \in B$
- By definition, G with its set of subsets, B, has a hitting set of size k'.

□

Problem 5

Claim: 3-colorable $\leq_p$ 4-colorable

Proof:
Suppose we have a black box that can solve the 4-colorable problem. If we can reduce the 3-colorable problem into the 4-colorable problem, then we are done. Suppose we have some graph G and we want to know whether or not G is 3-colorable by using the black box. We must make some modification to G and pass this modified graph G' into the black box. The following property must hold after the modification: if G' is 4-colorable, then G is 3-colorable.

We can modify G in the following way. We can add a single node to G, such that this node is connected to all of the other nodes in G. We will call this new graph G'. Our claim is that if G' is 4-colorable, then G is 3-colorable.

Claim: If G' is constructed using the above method, and G' is 4-colorable, then G is 3-colorable.

Proof:

- By construction of G', we know $\exists$ a node that is connected to all other nodes in the graph
- Call this node v
- Color the graph with 4 colors. Since v is connected to every node in G, it must be true that v is the only node in G' with its color
- Remove v from G', G' becomes G again
- Since v was the only node with its color, G now has 3 colors
- Thus, G is 3-colorable.

□