Ethan Wong
CS 180

Final Exam

1. There is a simple approach that can be used based on Kruskal's algorithm. Simply put, we need to modify Kruskal's algorithm so it returns the maximum spanning tree instead of the minimum spanning tree. This can be done by having the sorting be in nonincreasing order instead of nondecreasing order. Then we return the edges that are not within said maximum spanning tree; these edges should be excluded so that the maximum spanning tree doesn't actually form a cycle. These edges, when removed, will create an acyclic graph. The edges will also total up to be the smallest total weight because all the edges with greater weights will be a part of the maximum spanning tree.

```
def algo(G):
  G = graph that we are analyzing (given as input to algorithm)
  outsideEdges = empty set
  MST = empty set
  V = vertices in G
  E = array of edges of G sorted in nonincreasing order based on their weight

  Use Kruskal's algorithm with nonincreasing order to find maximum spanning tree

  # Create individual "components" for each of the vertices
  for (vertex in V):
    MAKE-SET(vertex)

  # Fill MST and outsideEdges based on whether u and v are in the same "component"
  # This loop creates the actual maximum spanning tree and isolates the outside edges
  for (edge (a, b) in E):
    if (FIND-SET(a) == FIND-SET(b)):
        add (a, b) to outsideEdges
    else:
        add (a, b) to MST
        Union(a, b)
  return outsideEdges
```

The time complexity of this algorithm is purely a result of Kruskal's algorithm, because all the other operations are less demanding than Kruskal's. Kruskal's itself runs in O(mlogn) time. This is because the sort operation in Kruskal's algorithm takes O(mlogn). Thus, the overall time complexity for this solution is O(mlogn).

2. A) The obvious approach would be to iterate through the array and compare each element with its neighbors to see if it is a hill. However, this is O(n) time. An O(logn) solution can be achieved by using divide and conquer. The approach is loosely tied to Binary Search. There are a few scenarios that may occur for the different situations:
    - Array length = 1 — return the index of the only element
    - Array length = 2 — return the index of the larger element
    - Array length > 2
        - Find middle element and compare it with its neighbors
        - If value of mid is greater than both neighbors, return index of mid
        - If value of mid is less than its right neighbor, use the right half of the array
        - If value of mid is greater than its right neighbor, use the left half of the array

    This logic is based off the fact that if an element of the array (not the right-most element) is less than its right neighbor, then there will be a hill element somewhere on its right, because the elements on its right are either:
    - In decreasing order — the hill is the left-most element
    - In increasing order — the hill is the right-most element
    - First decreasing then increasing — the hill is the left-most element
    - First increasing then decreasing — the hill is at the pivot point from increasing to decreasing

    Based on these observations, the algorithm is as follows:

```
def find1DHill(arr):
  arr = 1D array of numbers we are investigating (given as input to algorithm)
  right = len(arr) - 1    # last index in array
  left = 0                # first index in array

  while (left < right - 1):
    mid = (left + right) / 2   # middle element of array
    if (arr[mid] > arr[mid-1]) and (arr[mid] > arr[mid+1]):
      return mid          # mid itself was a hill, so just return it
    if (arr[mid] ≥ arr[mid+1]):
      right = mid - 1     # use left half of the array
    else:
      left = mid + 1      # use right half of the array
  if (arr[left] < arr[right]):
    return right
  else:
    return left
```

    This algorithm is basically just a modified version of binary search. The loop will run at most logn times and all of the other operations are performed in constant time. This means the overall time complexity for the algorithm will be O(logn).

2. B) The core idea of this algorithm is similar to the idea of the previous algorithm, but it uses divide-and-conquer. We want to split the matrix into quadrants by analyzing the middle row and the middle column. (If there is an even number of rows/columns just choose either of the rows and choose either of the columns). Find the maximum value in the middle row and the maximum value in the middle column. Once those maximum values are found, compare them to each other. Then take the larger value and compare it to its neighbors. If it is the largest value out of its neighbors, then it is the hill. If it is not the largest value, use recursion and repeat the same steps with that portion of the matrix to which that larger value belongs to. This will be repeated until a maximum value that is greater than all of its neighbors is found, and it will be a hill.
Note: I am assuming that all values in the matrix are distinct, just like in part 2a.

The algorithm is as follows:

```
def find2DHill(Matrix):
    Matrix = 2D array of numbers we are investigating (given as input to algorithm)
    rowMax = max value in the middle row of Matrix
    colMax = max value in the middle column of Matrix

    # hill found at the intersection of middle row and middle column
    if (rowMax = colMax):
      return rowMaxIdx

    # local hill of middle column and middle row was found in the middle column
    elif (colMax > rowMax):
      if (colMax > right neighbor) and (colMax > left neighbor):
         return colMax      # colMax itself is already a hill
      elif (right neighbor of colMax > left neighbor of colMax):
         return find2DHill(Matrix on right side of middle column)
      elif (right neighbor of colMax < left neighbor of colMax):
         return find2DHill(Matrix on left side of middle column)

     # local hill of middle column and middle row was found in the middle row
     elif (colMax < rowMax):
       if (rowMax > top neighbor) and (rowMax > bottom neighbor):
         return rowMax      # rowMax itself is already a hill
       elif (top neighbor of rowMax > bottom neighbor of rowMax):
         return find2DHill(Matrix on the top side of middle row)
       elif (top neighbor of rowMax < bottom neighbor of rowMax):
          return find2DHill(Matrix on the bottom side of middle row)
```

Since this is a divide and conquer algorithm, it makes sense to analyze it with the Master Theorem. There is 1 subproblem of size of n/2, and an additional cost of 2n to find the maximum of the middle row and to find the maximum of the middle column. When applying these values to the Master Theorem, it is an O(n) algorithm.

3. The best way to approach this problem would be to use dynamic programming. We start with “n” cities along the highway and are trying to place “K” fire stations in the cities such that the average distance from each city to the fire station is minimized. There will be a lot of computations for the distances, so it would be best to store the data in an array. We should first start with a simple case of K=1 (one fire station) and place the fire station at the last city on the highway. As we use higher values of K, we can place the fire stations between the first and last city on the highway. Basically, we start with the simple case and progress from there.

```
def fireStations(n, K):
  dist = [ ][ ]
  # dist[a][b] is the min. distance between the prev. fire station and the i cities
  tab[K][n] = empty 2D array to keep track of calculations
  # for the sake of this algorithm, pretend that arrays are indexed from 1 instead of 0

  # fill up the first row of tab (based on the initial placement of 1st fire station)
  for (x = 1; x < n; x++):
    tab[1][x] =  distance between 1st  city to the fire station at the xth city

  # fill up the remaining rows of the tab table for other fire stations (distances)
  for (i = 2; i≤K; i++):
    for (x = i+1; x≤n; x++):
      for (y = x-1; y < i; y++):
        tab[i][j] = min(tab[y][x-1] + dist[y+i][x], tab[i][x])

  # check if the position of the first fire station was correct
  for (int a = 1; a ≤ n; a++):
    tab[K][a] = min(tab[K][a] + sum of distances from the ath city)

  # look in tab to find the minimum distances
  look in tab to find cities that correspond with the minimum distances
  return those cities as locations for the K fire stations
```

The algorithm itself looks intimidating, but the idea behind it is not too complex. For example, imagine that we have to place 2 fire stations in n cities. To start with, we would tentatively place a fire station in the n^{th} city. Now there is 1 more fire station to place. To figure out where to place the remaining station, we would look at the row in the table where there is a single fire station. While looking at that row, we can see the distance from the first distance to the next city on the highway. Next, we check if the position of the initial fire station in the last city could be improved. If it can be improved, then we do so. Finally, we check the table to find the minimum distances and the cities that correspond to them — the fire stations will be placed there.

The time complexity of this algorithm is $O(n^2K)$. This is because we “remember” the data in a 2D array. The computations that go into the array take O(n) time because of the min() function and the array accesses. Filling the array with the computations takes O(Kn) time because those are the dimensions of the array. Therefore, the overall time complexity of the algorithm is $O(n^2K)$.

4.

A) A problem belongs in NP as long as a solution to the problem can be verified in polynomial time. To verify the solution of a Forest-Verify problem, all that needs to be done is a simple tree traversal based on the input binary string. If the traversal yields the same result as the solution, it will be NP. The worst case scenario would be that you have to traverse all T trees to their very bottom node. The very bottom node of a tree could be d+1 at most, since there can be up to d nodes based on the input string and the final “+1” or “-1” node. Therefore, in the worst case scenario the time to check a solution would be the traversal of T trees that have depth d+1. Traversal is a polynomial time operation, and doing it T times would still be polynomial. Therefore, the Forest-Verify problem belongs to NP.

B) A single 3-SAT clause can indeed be turned into a single decision tree such that the prediction of the Decision tree corresponds to the value of the clause. This can be done quite simply:

- Choose the first literal in the clause to be the root of the decision tree. If the first literal is true, set the right child node to “+1” and the left child node to one of the other unused literals in the clause. If the input value for that literal is “1” then it will traverse to the right. If the input value for that literal is ”0”, then it will traverse to the left. Note that if the literal is a negative, then the opposite will happen — if the input value for that literal is 1 it will traverse to the left, and if the input value for that literal is 1 it will traverse to the right.
- Do the same thing as above but with the node that contains the unused literal, from the above step.
- For the last literal, set the right node to “+1” and the left node to “-1”. The left node would be the case where all the literals evaluated to “0” based on the input string and the right node would be any case where one or more literals evaluated to ”1” based on the input string.

C) A polynomial-time reduction from 3-SAT to Forest-Verify can be accomplished. To begin with, we create a single decision tree from the first clause of the CNF (just like how we did in part B). If there are more clauses in the CNF that do not have trees created for them, then we must create decision trees for them as well. They must also be processed so that they link up properly with the original decision tree for the first clause. This essentially creates a massive tree that is a polynomial-time reduction from 3-SAT to Forest-Verify. A generalized algorithm for this process is shown below.

```
def ForestSAT(CNF):
    initTree = single decision tree created from the clause of the CNF (like in part B)

    while (another clause exists in the CNF that has not been “processed”):
        unprocClause = single decision tree created from unprocessed clause of the CNF
        for (litNode in unprocClause):
            while (litNode’s children are not both “+1” and “-1”):   # not the leaf nodes
                if (litNode’s left child is “+1”):
                    if (litNode’s left child is not negated):
                        set its right node to lead into initTree
                        set its left node to “+1”
                    elif (litNode’s left child is negated):
                        set its left node to lead into initTree
                        set its right node to “-1”
                elif (litNode’s right child is “+1”):
                    if (litNode’s right child is not negated):
                        set its right node to lead into initTree
                        set its left node to “+1”
                    elif (litNode’s right child is negated):
                        set its left node to lead into initTree
                        set its right node to “-1”
        mark this clause as “processed” so it doesn’t get repeated
        initTree = unprocClause
```

This algorithm is just a compact way of explaining how to create a tree that is a polynomial-time reduction from 3-SAT to Forest-Verify. This newly constructed tree will correspond to the 3-SAT. The tree will operate in the same way as the 3-SAT — if any of the given clauses are true, then the tree (and therefore 3-SAT) will return true. If all of the clauses are false, then the tree (and therefore 3-SAT) will return false. This algorithm shows that there is a polynomial time reduction from 3-SAT to Forest-Verify.