21S-COMSCI180-1 Final

SANCHIT AGARWAL

TOTAL POINTS

95 / 100

QUESTION 1

1 Q1 **20 / 20**

✓ **- 0 pts** Correct

- 4 pts No runtime analysis.

- 20 pts Wrong Algo.

- 12 pts Do not satisfy the runtime constraint.

- 3 pts Tiny mistakes.

- 2 pts Do not return the edges should be removed.

QUESTION 2

Q2 25 pts

2.1 (a) **10 / 10**

✓ **- 0 pts** Correct

- 2 pts Algorithm slightly off

- 2 pts Missing complexity, proof of correctness

- 5 pts Click here to replace this description.

- 10 pts Missing tag to question, please check with TA

2.2 (b) **15 / 15**

✓ **- 0 pts** Correct

- 5 pts Suboptimal / incorrect algorithm

- 6 pts Incorrect complexity with incorrect algorithm

- 7 pts Click here to replace this description.

- 10 pts Click here to replace this description.

QUESTION 3

3 Q3 **25 / 25**

✓ **- 0 pts** Correct

- 10 pts Not using 2D dynamic programming.

- 25 pts No answer

- 5 pts Minor mistake with dynamic programming.

QUESTION 4

Q4 30 pts

4.1 (a) **7 / 7**

✓ **- 0 pts** Correct

- 7 pts Missing

4.2 (b) **6 / 7**

- 0 pts Correct

- 0.5 pts very minor error (see comment)

✓ **- 1 pts** **special case missing: negation operator**

- 1 pts Value assignment is wrong: +1 is true and -1 is false.

- 2 pts Clauses won't contain AND operator

- 3 pts Incomplete answer: after reading your answer, I still don't know how exactly to create such a tree.

- 3 pts A clause is a three-tuple connected with *or* operators, not "and".

- 5 pts Mostly wrong / only answered "Yes" with no correct explanation.

- 7 pts Empty / completely wrong.

- 7 pts Handwriting extremely hard to read (regrade request is welcomed).

4.3 (c) **12 / 16**

- 0 pts Correct

- 8 pts Reduction is not polynomial. The ForestVerify problem must be polynomial size. A single tree needs 3x more subtrees per layer (one per +1), giving exponential size. Also not allowed to just do pointers, as the actual ForestVerify problem is still the full exponential tree. (Otherwise you can solve 3SAT in polytime by following a +1 node backwards)

- 12 pts Calling ForestVerify on each tree (clause) only tells you that each tree has some assignment that makes that tree true. Trees representing a single clause are always satisfiable trivially. This has no guarantee that the SAME assignment makes ALL trees true. "Fixing this" takes exponential time, as it would be a solver for 3SAT, since ForestVerify is trivial on clause trees.

- 12 pts Not allowed to modify the evaluation function or node values. Nodes must be +1/-1. Evaluation is always "True if sum >= 0". You can only check if it's in ForestVerify, aka if some assignment makes sum >= 0 or not. Changing any of this makes it not a valid ForestVerify problem. This was stated in the FAQ. (For comparison, you can't for example change what 3SAT being satisfied means, so you can't change what ForestVerify being True means)

- 12 pts Did not account for ForestVerify accepting if half of the trees can be satisfied, while 3SAT requires all to be satisfied. "sum of [+1 or -1] > 0" only needs half to be +1, the other half can be -1. E.g. just stated to use part (b) for every clause, and no other construction.

Close to correct (trying to add extra trees that are always false), but minor issue

- 4 pts Tried to add "always false" clauses then turn them into trees, which can't be done as seen in HW4 Approx. Close enough to the right answer, you can go straight to making "always -1" trees.

✓ **- 4 pts** **Can't have "always false" literals, since the decision problem considers all possible assignments (similar to HW4). Close enough to the right answer, as it's trying to make "always -1" trees.**

- 4 pts Can't have trees that are true when all clauses true and false when any clause is false, as that just solves 3SAT.

- 12 pts Defined "half-3SAT", reduction from 3SAT to half-3SAT is missing/wrong. This is not trivial. You can't just repeat Approx from Hw as that adds 7 T and 1F and can't reach 1/2, and you can't add "always false" clauses as seen with the HW.

- 0 pts Other issue (see comment)

- 16 pts Reduction in the wrong direction: Showed Forest < 3SAT, problem asked for 3SAT < Forest

- 16 pts No answer, no reduction/construction provided from multi-clause 3SAT, or extremely incorrect

CS 180 FINAL EXAM

Sanchit Agarwal
UID: 105389151

Question 1

Given:

- A Graph G with the following properties:
  - Undirected
  - Connected
  - Positive weight m edges
  - n nodes

To Find:

- A set of edges with the following properties:
  - Removing the edges will make the graph acyclic
  - The edge set will have the smallest total weight

Lemma 1:

A undirected graph with n nodes cannot be acyclic if the number of edges are more than n – 1.

Proof(By Contradiction):

Assumption: Let there exist a graph G with n nodes and e > n – 1 edges and no cycles

As proved in lecture, a connected graph without a cycle is a tree => All the connected components of G are trees

Now, we also proved in lecture that a tree with n nodes has exactly n – 1 edges – (1)

Now, let us assume that in graph G there are k connected components each with $\{n_1, n_2, \dots n_k\}$ nodes respectively such that $n_1 + n_2 + \dots. n_k = n$

Now since each connected component is a tree, the number of edges in each connected component would be n – 1 where n is the total number of nodes.

No. of edges = $(n_1 - 1) + (n_2 - 1) + \dots. (n_k - 1) = (n_1 + n_2 + \dots. n_k) - k = n - k$

- ⇨ No. of edges = $n - k \leq n - 1 < e$ {For the graph to be defined $k \geq 1$}
- ⇨ Therefore our assumption is wrong
- ⇨ Lemma 1 proved.

Lemma 2:

A undirected acyclic graph with n nodes and n – 1 edges must be connected

Proof(By Contradiction):

Assumption: Let there exist a undirected acyclic graph with n nodes and n – 1 edges that is not connected

Let the graph have k connected components:

Since there is no cycle, each connected component is a tree with $n_i – 1$ edges where n_i is the number of nodes in the component (as shown in the proof of Lemma 1)

⇨ Total number of edges = $(n_1 – 1) + (n_2 – 1) + …. (n_k – 1) = (n_1 + n_2 + …. n_k) – k = n – k$

Since the graph is not connected the $k > 1$

⇨ $n – k < n – 1$

⇨ Therefore, our assumption is wrong

⇨ A undirected acyclic graph with n nodes can only have n -1 edges if it has 1 connected component => The graph is connected

⇨ An undirected acyclic graph with n nodes and n – 1 edges must be connected

⇨ A directed acyclic graph with n nodes and n -1 edges must be a tree.

Construction of Algorithm:

Now, according to Lemma 1, the maximum number of edges in a undirected graph such that it has no cycle is n – 1. **-- (A)**

According to Lemma 2 such an undirected acyclic graph is connected. **– (B)**

Now to find the set of edges to be removed of minimum weight such that the remaining graph is acyclic we have to atleast remove (m – (n – 1)) edges (By (A)). Removing more than (m – (n – 1)) edges will increase the cost of the edge set and so we should atmost remove (m – (n – 1)) edges. Now the remaining n – 1 edges must form a tree (as proved in Lemma 2). To minimize the weight of our edge set , the edges selected in the graph must have the maximum weight possible

⇨ We need to find a maximum spanning tree in the given graph and the set of all the edges remaining must comprise the set of edges with smallest total weight such that the graph is acyclic.

Preamble:
There were several algorithms discussed in the lecture to find spanning trees. I will be using the Kruskal's Algorithm to find the maximum spanning tree. The approach for finding a minimum spanning tree using Kruskal's Algorithm was discussed in lecture and finding maximum spanning tree is exactly the same process, just here instead of starting with a array of edges sorted in ascending order I start with an array of edges sorted in the descending order.

Algorithm:

```
start
  m <- set of all edges
  n <- set of all nodes
  m_new <- set of edges that form connected acyclic graph
  m_result -> minimum weight edge set desired

  sort m in descending order
  Apply Kruskals algorithm on the set m:
      Add every edge that does not form a cycle in m_new
      Add every other edge in m_result
return m_result
end
```

Correctness: By construction of the Algorithm
Complexity Analysis: O(m*log(n)) = O((m + n) * log(n))
The algorithm majorly does two things

1) Sort the set of m edges which would take O(m*log(m)) time
2) Kruskal's Algorithm for maximum spanning tree which would take O(m*log(m)) time (as proved in lecture)

Total Complexity = 2 * O(m*log(m)) = O(m*log(m))
Now since m <= n * n for any graph => log(m) < 2*log(n)
⇨ O(m*log(m)) = O(2*m*log(n)) = O(m*log(n)) = O((m + n) * log(n))

Hence Proved.

1 Q1 20 / 20

✓ **- 0 pts** Correct

- 4 pts No runtime analysis.

- 20 pts Wrong Algo.

- 12 pts Do not satisfy the runtime constraint.

- 3 pts Tiny mistakes.

- 2 pts Do not return the edges should be removed.

QUESTION 2

a)

Given:

- A one dimensional array with n unique elements

Definitions:

A Hill in a 1D array of size n is an element such that both of its neighbours are smaller than it.

To Find:

- A Hill in the given one dimensional array

Algorithm:

```
Algorithm 2a : Binary Search

start
  given array -> a
  start <- 0
  end <- n - 1
  while start and end are not the same:
    mid <- start + (end - start)/2

    # examine if the middle pivot is the hill
    if (mid = 0 or a[mid] > a[mid - 1]) and (mid = n - 1 or a[mid] < a[mid + 1]):
        break;

    # if left neighbour is greater then left half must contain peak
    else if mid > 0 and a[mid - 1] > a[mid]:
        end = mid - 1

    # If right neighbour is greater then right half must contain peak
    else:
        start = mid + 1
  return a[mid]
end
```

Correctness:

It is mathematically and logically obvious that given n distinct elements, for any subgroup there must exist one such element which is larger than all the other elements in the subgroup.

Now in the above algorithm, while at a pivot if the left element or right element is greater than this implies that there exists an element at the respective side that is bigger than its neighbours. This is true as there will be an element which is greater than all the other elements in that respective side, since this element is larger than all the other elements it must be larger than its neighbours. If that element is the corner element of the respective side than it itself will be the pivot as

- Left side leftmost: greater than all elements in the left side => greater than its right neighbour
- Left side rightmost: left side rightmost is the element left to the pivot => It is greater than all the elements at the left side and greater than the pivot as according to the algorithm left side is only chosen if its rightmost element (left neighbour of the pivot) is greater than the pivot
- Right side leftmost: right side leftmost is the element right to the pivot => It is greater than all the elements at the right side and greater than the pivot as according to the algorithm right side is only chose if its leftmost element (right neighbour of the pivot) is greater than the pivot
- Right side rightmost: greater than all the elements at the right side => greater than its left neighbour

Therefore the above algorithm is just attempting to localize an element that is greater than its neighbour via binary search and since there has to be an element that is larger than all its neighbours at a particular side, this algorithm is guaranteed to complete according to the arguments given above.

Time Complexity:

In every iteration of the loop above, the search space is reduced to half. Since there are a total of n elements, in the worst case exactly after logn iteration we will converge to a single element. Since everything else in the loop (element comparisons etc) has a complexity O(1) => Time complexity of the above algorithm is O(logN)

2.1 (a) **10 / 10**

✓ **- 0 pts** Correct

- 2 pts Algorithm slightly off

- 2 pts Missing complexity, proof of correctness

- 5 pts Click here to replace this description.

- 10 pts Missing tag to question, please check with TA

b)

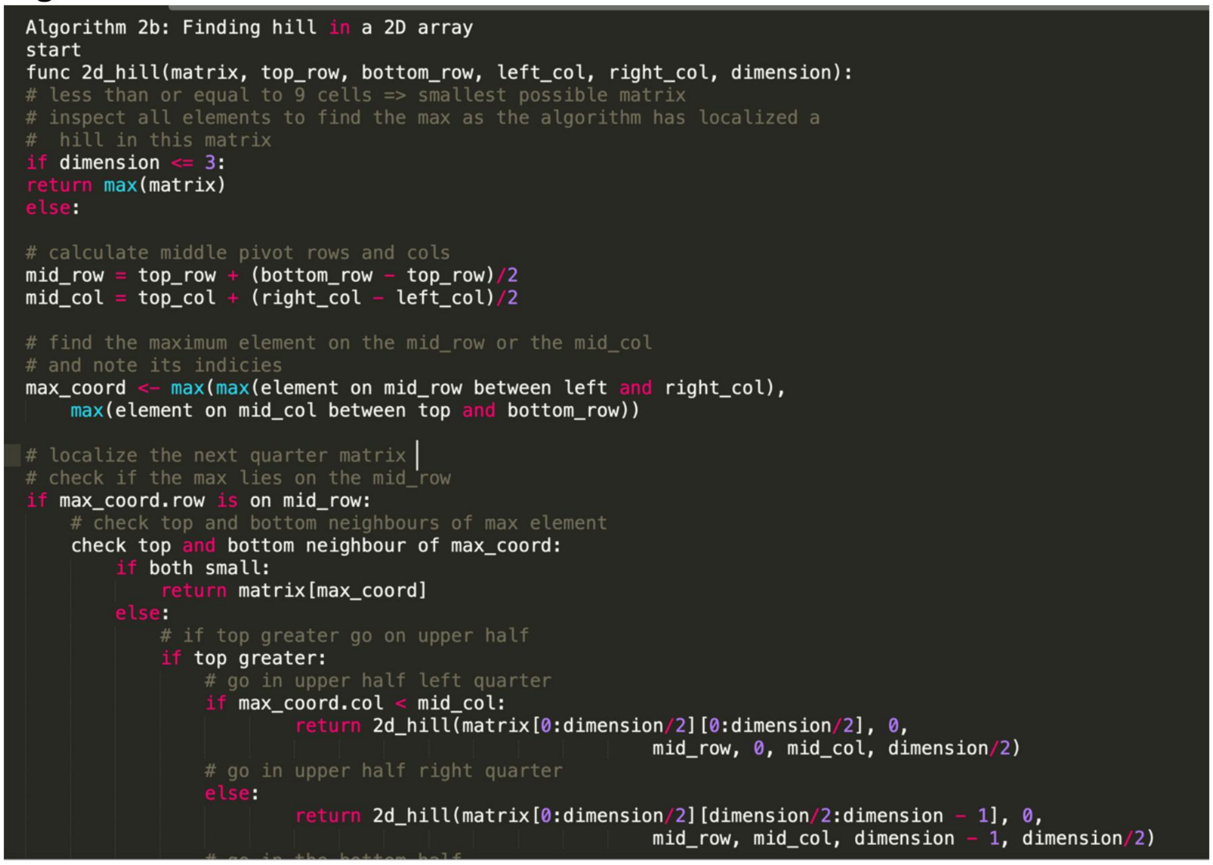
Given:

- A 2D array of size n*n

Definitions:

A hill in a 2D array is an element A[i][j] such that it is greater than A[i – 1][j], A[i + 1, j], A[i, j – 1], A[i, j + 1]

Algorithm:

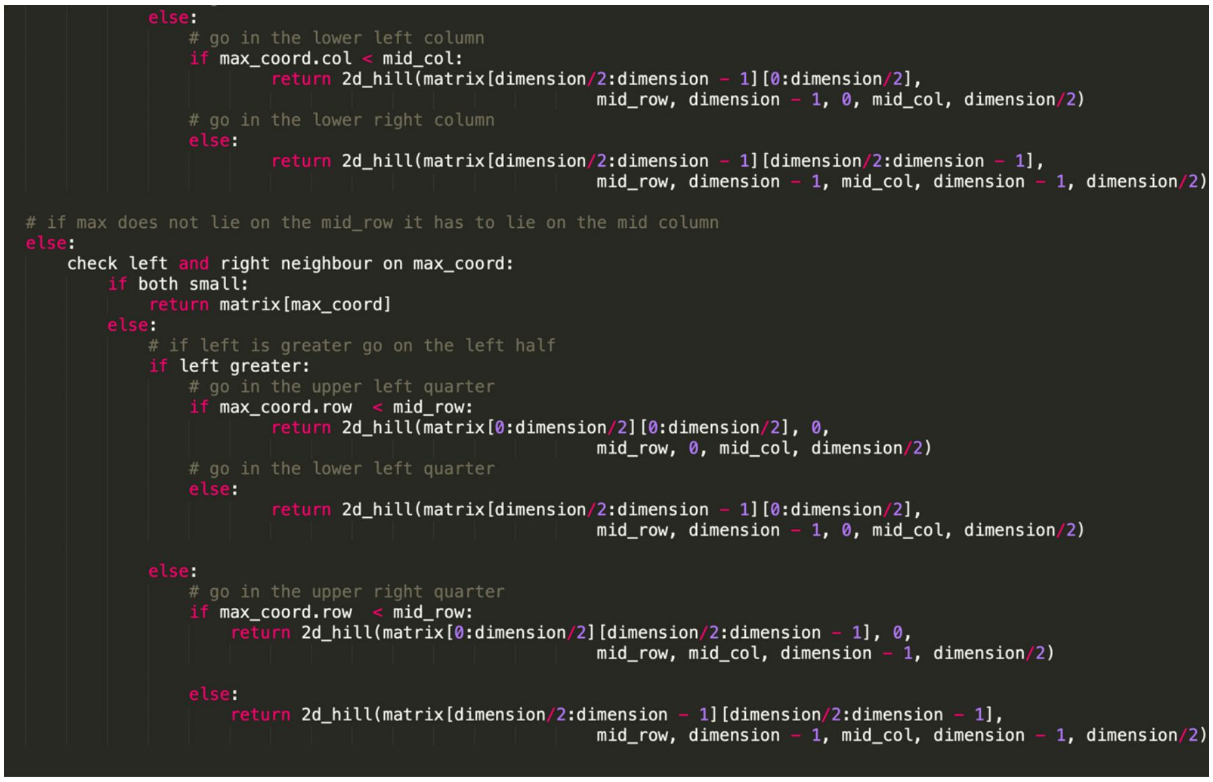


```
Algorithm 2b: Finding hill in a 2D array
start
func 2d_hill(matrix, top_row, bottom_row, left_col, right_col, dimension):
# less than or equal to 9 cells => smallest possible matrix
# inspect all elements to find the max as the algorithm has localized a
#  hill in this matrix
if dimension <= 3:
return max(matrix)
else:

# calculate middle pivot rows and cols
mid_row = top_row + (bottom_row - top_row)/2
mid_col = top_col + (right_col - left_col)/2

# find the maximum element on the mid_row or the mid_col
# and note its indicies
max_coord <- max(max(element on mid_row between left and right_col),
    max(element on mid_col between top and bottom_row))

# localize the next quarter matrix
# check if the max lies on the mid_row
if max_coord.row is on mid_row:
    # check top and bottom neighbours of max element
    check top and bottom neighbour of max_coord:
        if both small:
            return matrix[max_coord]
        else:
            # if top greater go on upper half
            if top greater:
                # go in upper half left quarter
                if max_coord.col < mid_col:
                        return 2d_hill(matrix[0:dimension/2][0:dimension/2], 0,
                                                        mid_row, 0, mid_col, dimension/2)
                # go in upper half right quarter
                else:
                        return 2d_hill(matrix[0:dimension/2][dimension/2:dimension - 1], 0,
                                                        mid_row, mid_col, dimension - 1, dimension/2)
                # go in the bottom half
```


```
            else:
                # go in the lower left column
                if max_coord.col < mid_col:
                        return 2d_hill(matrix[dimension/2:dimension - 1][0:dimension/2],
                                                        mid_row, dimension - 1, 0, mid_col, dimension/2)
                # go in the lower right column
                else:
                        return 2d_hill(matrix[dimension/2:dimension - 1][dimension/2:dimension - 1],
                                                        mid_row, dimension - 1, mid_col, dimension - 1, dimension/2)

# if max does not lie on the mid_row it has to lie on the mid column
else:
    check left and right neighbour on max_coord:
        if both small:
            return matrix[max_coord]
        else:
            # if left is greater go on the left half
            if left greater:
                # go in the upper left quarter
                if max_coord.row  < mid_row:
                        return 2d_hill(matrix[0:dimension/2][0:dimension/2], 0,
                                                        mid_row, 0, mid_col, dimension/2)
                # go in the lower left quarter
                else:
                        return 2d_hill(matrix[dimension/2:dimension - 1][0:dimension/2],
                                                        mid_row, dimension - 1, 0, mid_col, dimension/2)

            else:
                # go in the upper right quarter
                if max_coord.row  < mid_row:
                    return 2d_hill(matrix[0:dimension/2][dimension/2:dimension - 1], 0,
                                                        mid_row, mid_col, dimension - 1, dimension/2)

                else:
                    return 2d_hill(matrix[dimension/2:dimension - 1][dimension/2:dimension - 1],
                                                        mid_row, dimension - 1, mid_col, dimension - 1, dimension/2)
```

Algorithm on a top level basis:

- Find the max on the mid_row and mid_col
- If it lies on the mid_row then check for top and bottom neighbours:
    - If top is greater choose the upper half
        - If element is before mid column
            - Choose upper left matrix as next search space
        - else:
            - Choose upper right matric as the next search space
    - Else:
        - If element is before mid column
            - Choose lower left matrix as next search space
        - else:
            - Choose lower right matrix as the next search space
- If it lies on the mid_col then check for left and right neighbours:
    - If the left neighbour is greater
        - If the element is before mid_row
            - Choose upper left quarter
        - If the element is after mid_row
            - Choose the lower left quarter
    - else:
        - If the element is before mid_row
            - Choose upper right quarter
        - If the element is after mid_row

- Choose the lower right quarter

Correctness:
This algorithm is using the same intuition as in the 1d array. With every iteration we are localizing the potential “hill” by choosing a quarter such that there exist some element in it which is greater than both the dividing boundaries.
Now since in every group of elements, there has to be a largest element. Therefore lets consider the following cases:

- if the element that led to the selection of that particular quarter is the largest than it will be a hill as it is greater than the max element on the dividing boundary and is greater than all the elements in its quarter and so it must be greater than its neighbours.
- If some other element at the dividing boundary is the greatest than it again must be greater than the boundary as some element tin that quarter (one that led to the selection of the quarter) is greater than the maximum of the boundary but is still less than the greatest element in the quarter. Also since it is the greatest in that quarter, it will be greater than its neighbours and hence will be the greatest
- Any element lying somewhere in the middle of a quarter or on a non-dividing boundary has to be the greater than its neighbours as it is greater than all the elements in that quarter

In the above algorithm, calculating the max is necessary to avoid case number 2 above and so as shown above this algorithm guarantees to find a hill (even with all the corner cases) => The above algorithm is correct.

Time Complexity:
In every iteration the algorithm is looking over 2 * dimension elements to find the max on mid_row and mid_col. Rest all the computations like, comparisons and constant loop iterations in the base case are O(1).

Since with every iteration, the dimension is getting halfed

- ⇨ Complexity = 2 *n + 2 * n/2 + 2 * n/4 …… 2*n/logn
- ⇨ Complexity = 2 * n (1 + ½ + ¼ …. 1/logn)
- ⇨ Complexity = 2 * n * 2 * (1 – (1/2)^(log n)) ~ O(n)
- ⇨ Hence the above algorithm has a time complexity of O(n)

2.2 (b) **15 / 15**

✓ **- 0 pts** Correct

- 5 pts Suboptimal / incorrect algorithm

- 6 pts Incorrect complexity with incorrect algorithm

- 7 pts Click here to replace this description.

- 10 pts Click here to replace this description.

QUESTION 3

GIVEN:

- N cities
- K fire stations

To Find:

- An O(n^2 * k) algorithm which gives an optimal way to place k fire stations such that average distance from each city to the closest fire station is minimized.

Algorithm Construction:

Minimizing the average distance from every city to the closest fire station implies minimizing the sum of total distances from every city to the closest fire station – (1)

Now if the minimum cost to place j stations in i cities is considered to be OPT(i, j), then we need to find OPT(n, k).

Now for the nth city, either a fire station is placed on it or it isn't

⇨ OPT(n, k) = min(placement_cost(n) + OPT(n – 1, k – 1) , no_placement_cost(n) + OPT(n – 1, k))

Here placement_cost is the revision in the OPT(n – 1, k – 1) cost as all the cities to the right of the rightmost station in the n – 1 cities have the potential of being served by n if the cost is lower.

⇨ Lets say the rightmost station in n – 1 cities is m, then for any cities i between m and n, if dist(i, n) < dist(i, m) then placement_cost(n) += dist(i, n) - dist(i, m).

Here no_placement_cost(n) is the distance between the rightmost station in the n -1 cities and the nth city

If anytime there is a tie while calculating OPT(n, k) then we always go with placement of the station on n as a tie breaker.

Algorithm:

```
start
  input <- set of n coordinates for n cities, k
  coords <- set of n coorrdinates
  num_stations = k
  # we first sort the n coordinates
  s_coords <- sort(coords)
  num_cities <- length of s_coords
  dp_array <- n * k 2D array of (int, list) pairs (empty)
  distance_array <- n * n 2D array (empty)

  all the elements at the diagnol of distance array (i == j) = 0
  for i in 1,2...n:
    for j in i + 1,2...n:
      # distance between jth and ith city in s_coords
      # dist(i, j) == dist(j, i)
      distance_array[i][j] = s_coords[j] - s_coords[i]
      distance_array[j][i] = s_coords[i] - s_coords[j]

  # bottom's up approach Dynamic Programming
  for i in 1,2...n:
    for j in 0,1...k:
      if j > n:
        store nothing at dp_array[i][j]
      elif j == n:
        dp_array[i][j] = pair(0, list of all cities till i)
      else:
        # if I place jth petrol pump on city i
        # placement_cost(n) + OPT(n − 1, k − 1) case
        if j == 1:
          p_cost = sum of all indicies uptil col i in row i of distance_array

        else:
          p_cost = int in dp_array[i - 1][j - 1]
          old_rightmost_city = last element of list in dp_array[i - 1][j - 1]
          # calculating placement cost
          for city in old_rightmost_city to i:
            if distance_array[city][i] < distance_array[city][old_rightmost_city]:
              placement_cost += distance_array[city][i]
                              - distance_array[city][old_rightmost_city]
          p_cost == placement_cost
        # if I do not place the petrol pump on city i
        # no_placement_cost(n) + OPT(n − 1, k) case
        np_cost = int in dp_array[i - 1][j]
        old_rightmost_city = last element of list in dp_array[i - 1][j]
        no_placement_cost = distance_array[i][old_rightmost_city]
        np_cost = np_cost + no_placement_cost

        # calculating minimum cost
        # min(placement_cost(n) + OPT(n − 1, k − 1) ,
        #     no_placement_cost(n) + OPT(n − 1, k))
        if p_cost < np_cost:
          dp_array[i][j] = (p_cost, j appended to list in dp_array[i - 1][j - 1])
        else:
          dp_array[i][j] = (np_cost, list in dp_array[i - 1][j])
    return list in dp_array[n][k]
    end
```

Correctness:
The algorithm is correct by the construction of algorithm

Time Complexity:
In the algorithm I do many things. Lets evaluate each:

- Sorting s_coords = O(NlogN)
- Distance_array computation = O(N^2)
- Main DP Algorithm: I visit and update every cell of a n * k 2D array. Inside each iteration, I reference other 2D arrays and do some mathematical computations all of which is constant except calculating the placement cost which in worst case scenario can take upto O(N) Steps
    - Therefore, total iterations = O(N*logN)
    - Maximum cost in each iteration = O(N)
- Total Cost = O(N*logN * N) = O(N^2 * logN) -> desired

3 Q3 **25 / 25**

✓ **- 0 pts** **Correct**

- 10 pts Not using 2D dynamic programming.

- 25 pts No answer

- 5 pts Minor mistake with dynamic programming.

Question 4

a)
To prove:
Forest verify problem is NP Complete

Proof:
For forest verify problem to belong to NP there must exists an evidence t such that we can find a certifier B(s, t) such that B(s, t) checks if the result of forest verify problem is true under the assignment t.

In t, every ith element represents the truth assignment of x_i if the given input binary string $x_1 x_{2, \ldots\ldots} x_{d-1} x_d$
Therefore given a decision forest, I need a truth assignment of the d dimensional string such that it returns true.

Assumption:
There are k trees in the forest

Algorithm:

```
start (assuming 1 indexed arrays)
  a <- k * d 2D Array all elements initialized to -1
  for i in range 1,2...k:
    select the ith decision tree
    apply bfs from the root to a "+1" node in this tree
    assign a[i][m] the truth assignment observed on the path
  for j in range 1,2....d:
    ones <- count the number of 1s in a[][j]
    zeoes <- count the number of 0s in a[][j]
    if ones == zeroes:
      assign t[j] arbitarily
    elif ones > zeroes:
      t[j] = 1
    else:
      t[j] = 0
```

Explanation:
In the above algorithm I am basically finding t from the graph. Since I am applying BFS on each decision tree from root node to “+1” => each tree will lead to a result of +1. However some of the truth assignments will conflict

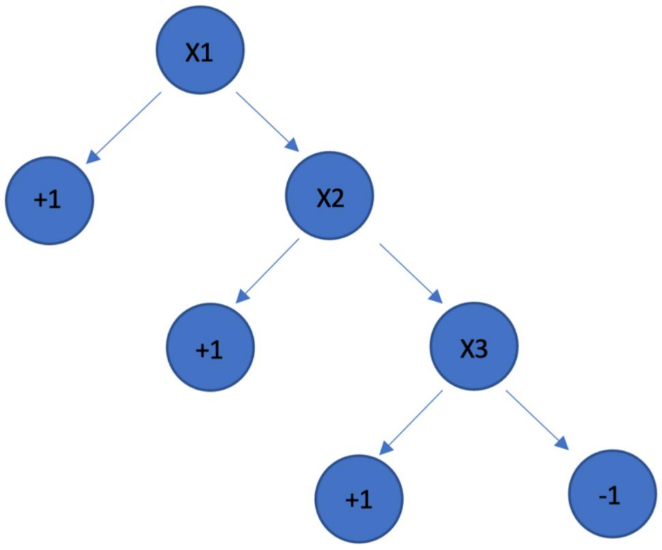
across various trees and so I calculated which what assignment of t_i leads to more number of positive result. The value that is more is assigned to t_i . This way I will have more +1s than potential -1s and the overall result will be true.

Time Complexity:
K rows of d elements -> O(k * d)
Count ones and count zeroes -> O(k * d)
Total Complexity - >O(k * d) = Polynomial
Therefore Forest Verify belong to NP.

b)
A 3-SAT clause is made up of 3 literals say x1, x2 and x3.
So a clause C can be represented x1 or x2 or x3
Now to build a decision tree, first note the value of each literal when they are set to a) 0 , b) 1.
Now starting from the first literal, make a node labelled x1 and make two outgoing edges one with the label 0 and other with the label 1. Make each of the edges to end at a node and then label the node +1 if x1(edge value) = 1 and label the other node x2.

Repeat the above process with x2 and then with x3. What we will have as a result would be a decision tree than encodes the 3-SAT clause

The decision tree will the structured as shown below and the edges will be marked according to the way described above.



4.1 (a) **7 / 7**

✓ **- 0 pts** Correct

- 7 pts Missing

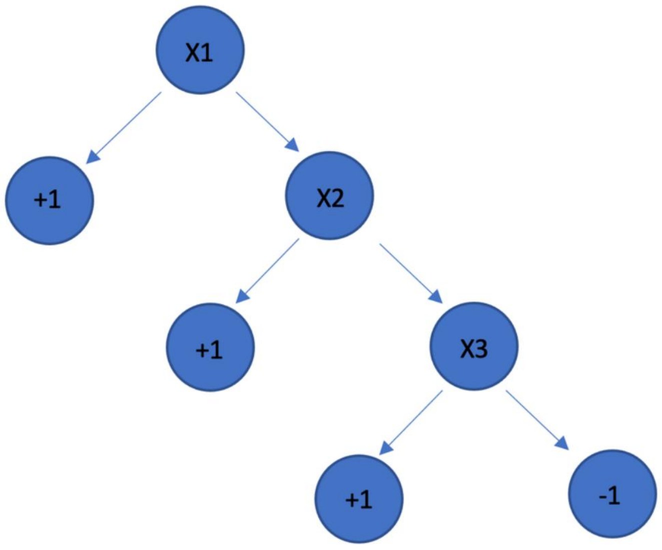
across various trees and so I calculated which what assignment of t_i leads to more number of positive result. The value that is more is assigned to t_i . This way I will have more +1s than potential -1s and the overall result will be true.

Time Complexity:
K rows of d elements -> O(k * d)
Count ones and count zeroes -> O(k * d)
Total Complexity - >O(k * d) = Polynomial
Therefore Forest Verify belong to NP.

b)
A 3-SAT clause is made up of 3 literals say x1, x2 and x3.
So a clause C can be represented x1 or x2 or x3
Now to build a decision tree, first note the value of each literal when they are set to a) 0 , b) 1.
Now starting from the first literal, make a node labelled x1 and make two outgoing edges one with the label 0 and other with the label 1. Make each of the edges to end at a node and then label the node +1 if x1(edge value) = 1 and label the other node x2.

Repeat the above process with x2 and then with x3. What we will have as a result would be a decision tree than encodes the 3-SAT clause

The decision tree will the structured as shown below and the edges will be marked according to the way described above.



4.2 (b) **6 / 7**

- **- 0 pts** Correct
- **- 0.5 pts** very minor error (see comment)
- ✓ **- 1 pts** special case missing: negation operator
- **- 1 pts** Value assignment is wrong: +1 is true and -1 is false.
- **- 2 pts** Clauses won't contain AND operator
- **- 3 pts** Incomplete answer: after reading your answer, I still don't know how exactly to create such a tree.
- **- 3 pts** A clause is a three-tuple connected with *or* operators, not "and".
- **- 5 pts** Mostly wrong / only answered "Yes" with no correct explanation.
- **- 7 pts** Empty / completely wrong.
- **- 7 pts** Handwriting extremely hard to read (regrade request is welcomed).

c)

Definitions:

Polynomial Reduction: *Reducing a problem A in polynomial time to a problem B means that I can use the solver of B to solve A in polynomial time*

Assumption:

- Verify-forest can handle duplicate decision trees (assumption without the loss of generality)

Algorithm:

```
start
  input <- set of n truth assignment of x1,x2....xn
  binary_string = x1,x2....xn
  forest <- set of all the decision trees (initially empty)
  for each clause Ci in C1,C2....Cn:
    # build a decision tree as described in part (b)
    add tree of Ci to forest

  # construct a dummy clause with 3 dummy literals
  now construct a clause Cd with literals x_d1, x_d2 and x_d3


  # add n copies of Cd in the
  for i in range 1,2...n:
    add Cd to the forest

  # append the dummy literals to reference binary string
  append x_d1 to binary string
  append x_d2 to binary string
  append x_d2 to binary string

  # append 0s to the input
  # assigning 0 to each dummy variable
  # this will assure that Cd is always -1
  append 3 zeroes to the input

  return forest-verify(binary_string, input, forest)
end
```

Evaluation and Correctness:
In the algorithm defined above, I am adding N dummy clauses that are always evaluated to false. Now according to how decision trees are described in the question, if a clause is False (i.e. not satisfiable) then the decision tree gives a -1. Since I have N dummy clauses that are guaranteed to be false, in my final sum calculated in Forest verify, these trees will contribute -N. Now if all the other N clauses are satisfies and hence are true I will get +1 for each of them from the decision tree and hence my overall value would be +N. Thus if all the N clauses are satisfies, in my final sum I will have N – N = 0. Forest-Verify according to definition will output true which is consistent with the behaviour or 3SAT. However if even 1 of the N clauses is not satisfies and its decision tree returns -1 then the total contribution from the N clauses will be < N. This will lead to the total sum to be less than 0 and the result of forest-verify to be false. Thus consistent with the behaviour of 3-SAT if all the clauses are satisfied I get True and rest in all other cases I get false.
Therefore the Algorithm is correct.

Time Complexity:
Creating Forest -> c * N = O(N)

  Each decision tree just have 3 literals and so can be made in constant time and there are n such literals

Creating Dummies -> O(1)
Forest-Verify -> O(P)

  Forest Verify is assumed to be polynomial with respect to its input

Total Complexity = O(P + N) = Polynomial

Hence Proved

4.3 (c) **12 / 16**

- **- 0 pts** Correct
- **- 8 pts** Reduction is not polynomial. The ForestVerify problem must be polynomial size. A single tree needs 3x more subtrees per layer (one per +1), giving exponential size. Also not allowed to just do pointers, as the actual ForestVerify problem is still the full exponential tree. (Otherwise you can solve 3SAT in polytime by following a +1 node backwards)
- **- 12 pts** Calling ForestVerify on each tree (clause) only tells you that each tree has some assignment that makes that tree true. Trees representing a single clause are always satisfiable trivially. This has no guarantee that the SAME assignment makes ALL trees true. "Fixing this" takes exponential time, as it would be a solver for 3SAT, since ForestVerify is trivial on clause trees.
- **- 12 pts** Not allowed to modify the evaluation function or node values. Nodes must be +1/-1. Evaluation is always "True if sum >= 0". You can only check if it's in ForestVerify, aka if some assignment makes sum >= 0 or not. Changing any of this makes it not a valid ForestVerify problem. This was stated in the FAQ. (For comparison, you can't for example change what 3SAT being satisfied means, so you can't change what ForestVerify being True means)
- **- 12 pts** Did not account for ForestVerify accepting if half of the trees can be satisfied, while 3SAT requires all to be satisfied. "sum of [+1 or -1] > 0" only needs half to be +1, the other half can be -1. E.g. just stated to use part (b) for every clause, and no other construction.

Close to correct (trying to add extra trees that are always false), but minor issue

- **- 4 pts** Tried to add "always false" clauses then turn them into trees, which can't be done as seen in HW4 Approx. Close enough to the right answer, you can go straight to making "always -1" trees.
- ✓ **- 4 pts** **Can't have "always false" literals, since the decision problem considers all possible assignments (similar to HW4). Close enough to the right answer, as it's trying to make "always -1" trees.**
- **- 4 pts** Can't have trees that are true when all clauses true and false when any clause is false, as that just solves 3SAT.

- **- 12 pts** Defined "half-3SAT", reduction from 3SAT to half-3SAT is missing/wrong. This is not trivial. You can't just repeat Approx from Hw as that adds 7 T and 1F and can't reach 1/2, and you can't add "always false" clauses as seen with the HW.
- **- 0 pts** Other issue (see comment)
- **- 16 pts** Reduction in the wrong direction: Showed Forest < 3SAT, problem asked for 3SAT < Forest
- **- 16 pts** No answer, no reduction/construction provided from multi-clause 3SAT, or extremely incorrect