Student Name: Yinsheng Dong
Student Number: 11148648
NSID: yid164
CMPT360 Assignment 2

Q1. Kleinberg and Tardos p.190 #5

Solution:

Algorithm:

```
'//': is the comment

// created by yid164
Algorithm:

    // list houses H
    a list name H contains all houses h

    // list stations S
    a list name S contains all station s that we constructed

    // traversal point p
    a point p

    // road r
    a road r which initial length = the road length

        // Algorithm function
        function():

            // start p, it can be either east to west or west to east
            p start west of r move to east
            while H is not empty:

                // inspect h to s
                if distance between p to any h in H == 4 miles:

                    //construction
                    construct s which add it to S

                    // remove h and all the covered houses
                    remove all h from H which distance between p to h <= 4 mile

                    // make the road length become to current
                    r = r - (p traversal that is from the end of west to current location)

            // return until there H is empty
            return function
```

Explanation of why it is correct:

Proof by induction:

From the case in this question, the worst case in the problem is that for covering n houses, we need to construct k stations which k = n

For the general case in this question, we might need to construct k station to cover n houses, k < n,

Base case:

We assume that in situation: the worst case: in the algorithm I implemented, there are k elements in S, and n elements in H, k = n

Inductive Step:

case 1: we add one more element h to H, and the distance between h an any element of S <=4: the algorithm will remind the k elements in S, so we have k < n

case 2: we add one more element h to H, and the distance between h and any element of S > 4 the algorithm will add one more s to S, so we have k = n

Conclusion:

The algorithm is correct since k <= n that I proved

Q2. Kleinberg and Tardo p.197 #17

Solution:

Algorithm:

In next page

```
Q2. Kleinberg and Tardo p.197 #17
    '//': is the comment

    // created by yid164
    Algorithm:

        // the variables I1, I2, I3 ... In for storing the interval info
        Assume every interval by using I1, I2, ... In

        // the solution store the itervals which Ii's no overlap interval
        Assume the solution Si which must contains an interval Ii


        // the solution is to find the solution for Ii which i is max_size
        find_solution (max_size) :

            // Si include 0 ,1, 2, 3, ... i
            initial Si include all intervals include Ii


            // in loop i find the all oeverlap intervals of Ii and then delete them
            for max_size of Si which i is fixed

                x be a point contained in Ii

                delete Ii and all overlap intervals of Ii from Si

            // finally, return the Si which is the every non-overlap of Ii
            Return Si
```

This function is using to find the potential solution for an interval, and it will return the Si which contains all the non-overlap of Ii

```
// the pick_solution for finding the best one (largest one)
pick_solution():

    // compute the find_solution(i), try the all Ii
    compute find_solution(i) which i = 1, 2, ,3, 4 ...n

    // largest_s set store the best solution
    an interval set largest_s will store the best solution

    // by comparing all the solutions, find the best one
    compare these solutions that we computed:

        if find_solution(i) < find_solution(j):
            largest_Si = find_solution(j)

        else if find_solution(i) > find_soluton(j):
            largest_Si = find_solution(i)

        else
            largest_Si could be both or either

    return the largest_Si
```

This function is looking for the best (largest) solution that intervals contained.

Explanation of why it is correct:

By this question, we know that the number of intervals (tasks) will not be 0 or less than 0**,** because if it is 0, CPU takes nothing, or if it is less than 0, it would be an error.

Then we assume that the problem has n intervals(tasks) which n > 0, and a set of S which contains all intervals.

Then pick_solution() will compare all the potential solutions which find_solution() produced .

Then largest_Si will be produced by the pick_solution()

Conclusion:

Since find_solution() produces the potential Si and pick_solution() gives the largest Si, so it should be correct.
Time-Complexity: find_solution(): O(n) and pick_solution(): O(n^2)
Total: O(n^2)

Q3.

Algorithm:

```
'//': is the comment
// created by yid164

Algorithm:

// T is the instance of sport team, which has s, f, and Q
Team T:
        start_time s, finish_time f, members Q

// list L contains all the teams
The list L of teams: [T1, T2, ... Tn]

// QiQj is the rumble of magitude, initially it is null
The QiQj

// Use the merge sort to divide the L and call the the merge()
merge_sort (L): merge sort all element in L

    // L can not be divided if it is empty or only one element
    if L has one element or empty
        return L

    // Divide the left, right side recursively, and call the merge() function
    Divide the L to Left_side and Right_side
    Left_side <- merge_sort(Left_side)
    Right_side <- merge_sort(Right_side)
    L <- merge(Left_side, Right_side)
```

This merge_sort() function is using to divide the list in 2 part (left and right) recursively.

```
// conquer function for merging
merge(Left_side, Right_side):

    // comparing there start time
    // if the left start time is earlier or equal to right
    if left_side.s is earlier or equal than right_side.s:

        // then compare if left one's end time and right one's start time
        // if it does overlap, then return the QlQr comparing to QiQj
        if left_side.f later or equal than right_side.s:
            if QlQr > QiQj or QiQj == NULL:
                QiQj == QlQr

        // else, continue merging
        else:
            continue merging

    // if left_side team 's start time is later the right side
    // then exchange the left and right, then return merging
    else:
        left_side = temp
        right_side = left_side_t
        right_side = temp
        return merge(left_side, right_side)
```

The merge() function is to merge all left and right teams to compare their start time and end time to check if their overlap or not, then to compare all the rumble of moganite, to get the best one.

The running time of the algorithm:

T(n) = 2T(n/2) + cn which c is the constant depends on how many teams in list

Time complexity is O(nlog(n))

Q4.

Algorithm:

In next page

```
Q4.

        '//': is the comment

        // created by yid164
        Algorithm:

        // Type point p which contains x and y
        Point p:
            x-axis: x, y-axis: y

        // the list L which contains of all point initially
        // goal is L will contains the rectangularly visible points at last
        The list L of points:
            L = [p1, p2, p3, p4... pn]

        //merge sort all element in L and L will contains all the rectangularly visible
        merge_sort(L):

            // if only one element or empty, then just return list
            if L has one element or empty:
                return L

            // just like the q3, divide in 2 part, left and right, then do the merge
            Divide the L to Left_side and Right_side

            // Merge sort A which A is left
            Left_side <- merge_sort(Left_side)

            // Merge sort B
            Right_side <- merge_sort(Right_side)

            // call the function merge()
            L <- merge(Left_side, Right_side)
```

The merge_sort() function is using to divide the given list L which L is contains all the points {p1, p2, p3, … pn} in 2 part (left and right) recursively, and calling merge() to calculate

```
// merge function to make sure if x is a RV or not
merge(Left_side, Right_side)

    // using these 2 loops to check left and right point's x, y
    for left_point in Left_side:
        for right_point in Right_side:

            // if left point and right point are in same point
            if left_point.x = right_point.x and left_point.y = right_point.y:

                // they are both not RV, then remove them
                remove left_point and right_point

            // if the left point's x is larger than right's x
            else if left_point.x > right_point.x:

                // exchange them and return merge
                exchange left_point and right_point
                return merge(Left_side, Right_side)

            // When left_point.x smaller the right one
            else if left_point.x < right_point.x:

                // and the left y is smaller or equal to right y,
                // the right must not be a RV, so remove it
                if left_point.y <= right_point.y:
                    remove(right_point)
```

The merge() function is using 2 loops to find eligibility by checking left and right's x, y.
If right's x is smaller than left's, exchange them to guarantee the left x is smaller than right's x.
If we got the left.x is still smaller than right.x, if the right.y is larger than left.y, the right must not be the RV, then we delete it recursively, until L only contains the RV.

Recurrence for the time:

merge_sort() = log(n),
merge() = n^2,
T(n) = log(n) * n^2
Time complexity: O(n^2)