Homework 3

Liwen Ouyang

March 15th 2019

Problem 1

To begin with we want to show that it is impossible to assign drivers if a path from starting node$s$ to destination t has n edges where $n \equiv 0 \mod 3$, provided that s and t are distinct.

Without loss of generality let's assume there are n segments assigned to A. By the constraints, there can be $n-1, n, n+1$ Bs and Cs. Apparently, the only combinations where the total number of segments is divisible by 3 are $(n, n, n); (n, n-1, n+1)$. We'll show neither of them is possible. For the case of (n, n, n), i.e. $n \geq 1$ segments for A,B and C. It is impossible because we know A must have the first and last segment, therefore in the middle $3n-2$ segments, there are $n-2$ A, n B and C, which violates the constraint because there are 2 more B and C than A in the segments excluding the first and last segment. For the case of $(n-1, n, n+1)$, it is also impossible because $n+1-(n-1) = 2 > 1$, which means there is one person that drives 2 more segments than another for the whole trip. Hence we can conclude any path that has n edges where $n \equiv 0 \mod 3$ cannot be valid. Besides, also note that a path that has 2 edges is also not possible trivially.

We will be using a modified Dijkstra's algorithm where we use a counter c to keep track of the number of edges of the path we discover. We will also be keeping track of the shortest paths to every node with the number of edges $n \equiv 0, 1, 2 \mod 3$, so that we will not be adding nodes arbitrarily number of times to the priority queue we use for our algorithm.

procedure SHORTESTPATH(M, s, t) ▷ Given a map M and distinct nodes s, t
  Initialize Q as a priority queue
  Initialize an Array A_i for each node i to store the shortest discovered path
  $A_i[x]$ for $x = 0, 1, 2$ stores the shortest weight of the path whose number of edges is x modulo 3
  set all three entries of A_i to be positive infinity
  put $(s, 0, 0)$ onto Q ▷ Note the first 0 is weight as priority, the second 0 is a counter for edges
  while Q is not empty **do**
    $(i, w, c) \leftarrow Q.extractMin()$
    if i is $t \wedge c \neq 2 \wedge c$ is not divisible by 3 **then**
      declare victory with a shortest path of weight w, assign corresponding letters.
      assign first and last segment to A
      from second to second last segment assign driver following sequence B, C, A
    end if
    for all neighbors x of i, with a edge of cost l **do**
      if $w + l \leq A_i[c\%3]$ **then**
        put $(i, w + l, c + 1)$ into Q
        $A_i[c\%3] \leftarrow w + l$
      end if
    end for
  end while
  declare it is impossible
end procedure

The algorithm is modified Dijkstra, and we will only put a node onto queue a finite number of times, which means it has a running time of $O(m \log n)$. It is correct because Dijkstra's algorithm is going to return a

shortest path from starting node to every node, and we modified it so that it won't declare victory until it finds a shortest path that satisfies our constraints. Therefore our algorithm gives the shortest valid path.

Problem 2

(a) To begin with we want to prove a cut property of maximum spanning tree: let e be the maximum-weight edge crossing cut $(S, V/S)$ in G. Then e belongs to every minimum spanning tree of G.

Let $e = (v, w)$ be the maximum-weight edge across cut $(S, V/S)$ and suppose for contradiction that T is maximum spanning tree but does not include e. There is a path from v to w in T. Let $e' = (v', w')$ be an edge on this path that crosses the cut. Let $T' = T + \{e\} - \{e'\}$. T' is still a spanning tree because any path in T that needs e' can now be routed via e and there will be no cycle as adding e creates one cycle, removing e' destroys it. Since e is the maximum-weight edge from S to V/S,

$$w(T') = w(T) - w(e') + w(e) > w(T)$$

which is a contradiction, because we assumed T is the maximum spanning tree.

Now seeking for contradiction let's assume there is a path from arbitrary node s to another node t of maximum capacity c that is not in the maximum spanning tree T. Now let the edge of lowest cost $|e|$ in the path in T be e. Let there be a cut $S, V/S$ such that $s \in S$ and $t \in V/S$, and e is the maximum cost edge that crosses that cut. This cut is guaranteed to exist because of the cut property we proved. Now if there's a path of capacity c from s to t, then there must be an edge e' in the path. However, we know $c > |e| > |e'|$. This means that path cannot possibly have a capacity of c, which is a contradiction.

(b) We will be using a modified BFS in this problem. We will only discover a node a from a node b if the cost of the edge (a, b) is strictly larger than C. Our algorithm will return True if we can't reach t from s at the end of the search and True otherwise.

```
procedure AtMostC(C, s, t, G)        ▷ given a graph G and nodes s and t and constant C
    Initialize a Queue Q for BFS
    put s into Q
    while Q is not empty do
        n ← Q.dequeue()
        if n is t then
            return False                                          ▷ Find t from s
        end if
        for all neighbors i of n do
            if |(n, i)| > C then
                put i into Q
            end if
        end for
    end while
    Return True                                                   ▷ Cannot find t from s
end procedure
```

The algorithm is essentially a modified BFS, which means it has run time of $O(m+n)$. Our algorithm is correct because we can only find t from s if there is a path from s to t such that every edge in that path has a cost that is larger than C. This means the capacity of that path is greater than C, which tells us the bottleneck capacity cannot possibly be less than C. Hence we will return false.

(c) We will construct the maximum spanning tree of the graph using a modified prim's algorithm first, and then traverse the tree to get the bottleneck capacity.

```
procedure MakeTree(G, s)        ▷ given a graph G and a starting node, return the maximum
spanning tree T of it
```

```
    set A as a priority queue and put all nodes into it
    set a(n) = −∞ for all nodes
    set a(s) = 0
    set edgeTo(s) = null
    while A is not empty do
        v ← A.dequeque()
        set T = T ∪ edgeTo(v)
        for all edges (v, w), where w ∈ A do
            if c(v, w) > a(w) then
                a(v) = c(v, w)
                edgeTo(w) = (v, w)
            end if
        end for
    end while
    Return T
end procedure
```

This algorithm has a run time of $O(m \log n)$ as shown in lectures. It is going to give maximum spanning tree based on the cut property we proved in (a). Now that we have the maximum spanning tree, according to (a), we only need to traverse it to get the bottleneck capacity:

```
procedure BOTTLENECK(G, s, t)          ▷ Given a graph G and two nodes s, t
    T ← MakeTree(G, s)
    E ← BFS(T, s, t)          ▷ Use bfs to find the unique path from s to t in T: O(n + n − 1)
    return the edge that has the highest cost in E          ▷ at most n − 1 edges: O(n − 1)
end procedure
```

The algorithm has a running time of $O(m \log n + n)$ in total. Based on (a), the algorithm is correct because we proved T, as the maximum spanning tree of G, contains a path from s to t that has the bottleneck capacity.

Problem 3

(a) For a $O(n^2)$ algorithm, we will just take a brute-force approach by searching for the targets for every entry of the array.

```
procedure FINDTARGET(H)          ▷ Given an array representing people of different height
    for every index i of H do
        i1, i2 ← i                                          ▷ Initializing two counters
        while i1 >= 0 do                                    ▷ Searching for target before i
            if H[i1] > H[i] then
                declare H[i1] is the target of i
                break
            end if
            i1 − −
        end while
        while i2 < H.length do                              ▷ Searching for target after i
            if H[i2] > H[i] then
                declare i2 is the target of i
                break
            end if
            i2 + +
        end while
    end for
end procedure
```

This algorithm has a running time of $O(n^2)$ because for each element of the array we will check $O(n)$

element, which yields a running time of $O(n^2)$. It is trivially correct because we are just searching the targets of every person through the entire array.

(b) Using divide and concur we will divide the array into two arrays of half of its size, and assuming we know all the shootings happened within two sub-arrays, we will establish shootings that are from one sub-array to another. In order to do this, we will define our input as an array of objects.Each object representing a person has three fields: h: an int as height, l: a person object representing the left target; r: a person object representing right target. We will set h to the corresponding numbers and r and l as null in the beginning. Besides, we will also define a helper function merge, which takes in two array of person object and merge them in to one while updating the l and r fields of every element, i.e. establish shootings that cross two arrays.

```
procedure MERGE(L,R)                    ▷ L, R are two arrays that have objects with updated fields
    i ← L.length − 1
    j ← 0
    while i ≥ 0 ∧ j ≤ L.length + R.length − 1 do
        if L[i].h > R[j].h then
            if R[j].l ≠ null then
                R[j].l ← L[i]
            end if
            j + +
        end if
        if L[i].h < R[j].h then
            if L[j].r ≠ null then
                L[i].r ← R[j]
            end if
            i − −
        end if
    end while
    return S ← L ∪ R                    ▷ preserve order: all elements of L come before R
end procedure
```

Note that the running times of this algorithm is $\Theta(n)$ because we are just iterating over two arrays of similar size one time. Now that we have the helper function, we an simply define a recursive function to apply divide and conquer to solve the problem:

```
procedure FINDTARGETS(A)                ▷ Given an array of person objects
    if A.length = 1 then                ▷ Base Case
        return A
    end if
    L ← first half of A                 ▷ Divide A into half
    R ← remaining half of A
    return merge(FindTargets(L),FindTarget(R))
end procedure
```

The algorithm is correct because we assume the shootings that happen within L and R are already established, and we establish all the shootings that cross L and R while merging. By induction, the algorithm is going to establish all the shootings in A. The running time of the algorithm can be described by the recurrence: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$

(c) Note that a person is not shot if and only if there are two (one for the first and last person) people that are both higher that that person right next to that person. That being said, there can be at most $\lceil \frac{n}{2} \rceil$ people who are not shot, because the maximum case will be an alternating sequence of high and short people. That is, every odd person is shorter than the person right before and after. Hence, there can be at least $\lfloor \frac{n}{2} \rfloor$ people get shot in one round.

(d) To begin with it's useful for us to define a function that tells us who will not be shot in a given round assuming there are more than 2 people:

```
procedure NOTSHOT(A)                                          ▷ Assuming |A| ≥ 3
    initialize an empty array A_s to store result
    if A[0] < A[1] then
        put A[0] into A_s
    end if
    for 1 ≤ i ≤ |A| − 2 do
        if A[i] < A[i − 1] ∧ A[i] < A[i + 1] then
            put A[i] into A_s
        end if
    end for
    if A[|A| − 1] < A[|A| − 2] then
        put A[|A| − 1] into A_s
    end if
    return A_s          ▷ Note that A_s preserves the relative order of the people who are not shot
end procedure
```

This algorithm is correct because we have already established that a person is not shot in a round if and only if that person is shorter than the people or person right next to that person in (c). The running time is $\Theta(n)$, since we are just iterating over the whole array. Now we can define a recursive algorithm to calculate the rounds it takes to end a game:

```
procedure ROUNDS(A)                                   ▷ Given array representing people's heights
    if |A| ≤ 1 then
        return 0
    end if                          ▷ two base cases because of our assumption on A in notShot
    if |A| = 2 then
        return 1
    else
        A_s ← notShot(A)
        return Rounds(A_s) + 1
    end if
end procedure
```

The algorithm is correct because we are simply just recursively eliminating people who are shot in every round and increment a counter for the number of rounds. Therefore it is going to give us the correct number of rounds by induction, given the correct base case where $|A| = 2$, which takes 1 round. The running time of the algorithm is going to be the solution to the recurrence: $T(|A|) = T(|A_s|) + \Theta(n)$. Note that in (c) we showed that $|A_s| \leq \frac{|A|}{2}$. Therefore the recurrence becomes: $T(n) \leq T(\frac{n}{2}) + \Theta(n)$, which by Master theorem solves to $T(n) = \Theta(n)$

Problem 4

(a) We want to prove $T(n) = \Theta(n)$. For the lower bound let's assume $\forall n_0 : n_0 < n \rightarrow T(n_0) \geq cn_0$. Then by our inductive hypothesis we have:

$$T(n) \geq \frac{2cn}{3} + \frac{2cn}{9} + n = (\frac{8}{9}c + 1)n$$

Therefore we want a constant c such that $(\frac{8}{9}c + 1)n \geq cn$. Apparently for $c \leq 9$, the inductive step holds. Since we know when $n = 1, 2, 3$, T is bounded by some constant, we can choose c accordingly to prove the base case for the inductive proof.

Similarly we want to show that $\forall n_0 : n_0 < n \rightarrow T(n_0) \leq c'n_0$ for an upper bound, following same arguments above, we know the inductive proof holds when $c' \geq 9$. Hence we proved $T(n) = \Theta(n)$

(b) We want to show that $T(n) = \Theta(n^2)$. For the lower bound let's assuume $\forall n_0 : n_0 < n \rightarrow T(n_0) \geq xn_0^2$.

Then by our inductive hypothesis we have:

$$T(n) \geq \frac{1}{3}(x(n-1)^2 + x(n-2)^2 + x(n-3)^2) + cn = xn^2 + (c-4a)n + \frac{14}{3}x$$

Now it is trivial to see that if we choose x such that $c-4x > 0$, we have $T(n) \geq xn^2 + (c-4x)n + \frac{14}{3}x \geq xn^2$, which proves the inductive step. The base case is also satisfiable by choosing a, as we know T is bounded by some constant when $n \leq 3$.

For the upper bound, let's assume $\forall n_0 : n_0 < n \rightarrow T(n_0) \leq an_0^2 + bn_0$, by our hypothesis we have:

$$T(n) \leq \frac{1}{3}(a(n-1)^2+a(n-2)^2+a(n-3)^2+b(n-1)+b(n-2)+b(n-3))+cn = an^2+(c-4a+b)n-2b+\frac{14}{3}a$$

Now note that if we choose a, b such that $c - 4a + b \leq b$ and $-2b + \frac{14}{3}a \leq 0$, the inductive step holds:

$$T(n) \leq an^2 + (c - 4a + b)n - 2b + \frac{14}{3}a \leq an^2 + bn$$

And we can choose a, b to get a base case. Hence we proved that $T(n) = \Theta(n^2)$

(c) We want to show that $T(n) = \Theta(n \log n)$. For the lower bound let's assume $\forall n_0 : n_0 < n \rightarrow T(n_0) \geq cn_0 \log n_0$. Then by our inductive hypothesis:

$$T(n) \geq \frac{cn}{2}\log\frac{n}{2} + \frac{cn}{3}\log\frac{n}{3} + \frac{cn}{6}\log\frac{n}{6} + n = cnlogn - n(1 - c(\frac{\log 2}{2} + \frac{\log 3}{3} + \frac{\log 6}{6})$$

It is obvious that if we choose c such that:

$$1 - c(\frac{\log 2}{2} + \frac{\log 3}{3} + \frac{\log 6}{6}) \geq 0$$

The inductive step holds. Similarly, for the upper bound if we assume $\forall n_0 : n_0 < n \rightarrow T(n_0) \leq an_0 \log n_0$, we can choose a constant a such that:

$$1 - a(\frac{\log 2}{2} + \frac{\log 3}{3} + \frac{\log 6}{6}) \leq 0$$

so that our inductive step holds. For the base case we can choose corresponding constants. Therefore we have the bounds:$T(n) = \Theta(n \log n)$.

(d) We want to sow $T(n) = \Theta(n \log n)$ First let's assume $\forall n_0 : n_0 < n \rightarrow T(n_0) \leq an_0 \log n_0 + bn_0$. Then we want to prove that:

$$T \leq \sqrt{2n}(a\sqrt{2n}\log\sqrt{2n} + b\sqrt{2n}) + \sqrt{n} \leq an\log n + bn$$

which is equivalently:

$$(a\log 2 + b)n + \sqrt{n} \leq 0$$

apparently, if we choose $a > 0$ and b such that:

$$(a\log 2 + b) = -\epsilon < 0$$

We have:

$$-\epsilon n + \sqrt{n} \leq 0 \iff \sqrt{n} \leq \epsilon n$$

which is trivially true for sufficiently large n. We can choose a and b for the base case. Therefore we have $T(n) = O(n \log n)$

For the lower bound let's assume: $\forall n_0 : n_0 < n \rightarrow T(n_0) \geq cn_0 \log n_0$, then we have:

$$T(n) \geq \sqrt{2n}c\sqrt{2n}\log\sqrt{2n} + \sqrt{n} = cn\log 2n + \sqrt{n} \geq cn\log n$$

which proves the inductive step. We can choose c to find a base case. Therefore we have $T(n) = \Theta(n \log n)$

Problem 5

Adopting a divide and conquer approach, we will calculate the largest perfect tree T_1 in the left and right subtree, the largest perfect tree T_2 rooted at the root given and compare them.

To begin with we will define an algorithm to find the depth of the largest perfect tree rooted at a given node and the largest perfect tree in the subtrees of that node. The algorithm will return a tuple (n_m, d_m, n_s, d_s) where n_m, d_m are the root and depth of the largest perfect tree rooted at the given node. And n_s, m_s are the root and depth of the larges perfect tree obtainable from the proper subtrees of the tree.

procedure TREES(n) ▷ given the root n of a proper tree
  if n has no children **then**
    return $(n, 0, n, 0)$
  else
    $n_m^L, d_m^L, n_s^L, d_s^L \leftarrow Trees(n.left)$ ▷ Get result from left and right subtrees
    $n_m^R, d_m^R, n_s^R, d_s^R \leftarrow Trees(n.right)$
    $n_m \leftarrow n$
    $d_m \leftarrow \min(d_m^L, d_m^R) + 1$ ▷ see proof in (1) below
    $d_s \leftarrow \max(d_s^L, d_s^R, d_m^L, d_m^R)$ ▷ see (2)
    if $d_s = d_s^L$ **then**
      $n_s \leftarrow n_s^L$
    end if
    if $d_s = d_s^R$ **then**
      $n_s \leftarrow n_s^R$
    end if
    if $d_s = d_m^L$ **then**
      $n_s \leftarrow n_m^L$
    end if
    if $d_s = d_m^R$ **then**
      $n_s \leftarrow n_m^R$
    end if
    return (n_m, d_m, n_s, d_s)
  end if
end procedure

This algorithm will have a running time of $O(n)$ because essentially we are just traversing the tree and assign a tuple to every node. In order to show it's correct we need to show the following two properties:

(1) Denote the depth of maximum perfect tree rooted at n as $D(n)$, then $D(n) = \min(D(n.left), D(n.right)) + 1$. This can be shown using induction. The base case will be a node with two children, and it's trivially true because we have $1 = \min(0, 0) + 1$. Now assuming the proposition is true for the left and right subtrees of a tree rooted at n, we want to show $D(n) = \min(D(n.left), D(n.right)) + 1$. Without loss of generality we lets say $D(n.left) \geq D(n.right)$. Then we can conclude that we can get a perfect tree of depth $D(n.right)$rooted at $n.left$ and merge it with the maximum perfect tree rooted at $n.right$ to get the largest tree rooted at n with a depth of $D(n.right) + 1$. This completes our proof.

(2) The largest perfect tree obtainable from left and right subtrees is the maximum among the largest perfect trees rooted at $n.left$ and $n.right$, and the largest perfect trees in the subtrees of $n.left$ and $n.right$. This is trivial to see.

Now we can proceed to give an algorithm for the problem using a wrapper function:

procedure MAXTREE(n) ▷ given the root n of a proper tree
  $n_m, d_m, n_s, d_s \leftarrow Trees(n)$
  if $d_m \geq d_s$ **then**
    return (n_m, d_m)
  else
    return (n_s, d_s)

end if
end procedure

This algorithm gives the correct result, since we are just returning the larger perfect tree between the largest perfect tree rooted at the node and the largest perfect tree obtainable from its subtrees, and it has a running time $O(n)$ as shown for the helper function.