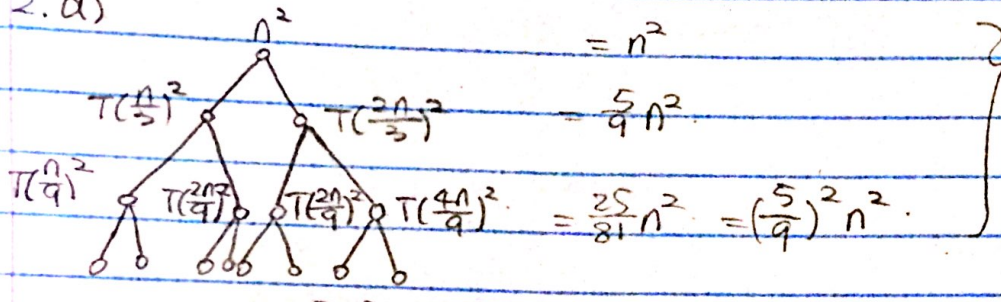
Zefang Zhu
20510897
CS341 A1.

$T(n) = T(\frac{2n}{5}) + T(\frac{n}{5}) + n^2$

2. a)



$T(n) = n^2 + \frac{5}{9}n^2 + (\frac{5}{9})^2 n^2 + \cdots$

$= n^2(1 + \frac{5}{9} + (\frac{5}{9})^2 + \cdots)$

$= n^2 \sum_{i=0}^{\log_3 n} (\frac{5}{9})^i$

$= n^2 \frac{-1}{\frac{5}{9} - 1} \leq dn^2$, for some constant d.

$\Rightarrow T(n) = O(n^2)$

b) $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$

let $n = 2^k$, then $k = \log n$

$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$

$\Rightarrow T(2^k) = 2^{\frac{k}{2}} \cdot T(2^{\frac{k}{2}}) + 2^k$ (divide by 2^k)

$\Rightarrow \frac{T(2^k)}{2^k} = \frac{2^{\frac{k}{2}} \cdot T(2^{\frac{k}{2}})}{2^k} + \frac{2^k}{2^k}$

$\Rightarrow \frac{T(2^k)}{2^k} = \frac{T(2^{\frac{k}{2}})}{2^{\frac{k}{2}}} + 1$

let $y(k) = \frac{T(2^k)}{2^k}$, $T(2^k) = y(k) \cdot 2^k$

$y(k) = y(\frac{k}{2}) + 1$

using master theorem, $a=1, b=2, c=0$

$\Rightarrow y(k) = O(k^0 \log k) = O(\log k)$

$T(n) = T(2^k) = O(\log k \cdot 2^k) = O(n \log \log n)$.

$\Rightarrow T(n) = O(n \log \log n)$

3. MergeCount(P, temp, left, mid, right)

```
count ← 0
i ← left
j ← mid
k ← left
while i <= mid-1 and j <= right then
    if P[i].y <= P[j].y or P[i].x == P[j].x then
        temp[k] = P[i]
        k++; i++;
    else
        temp[k] = P[j]
        count = count + mid - i ; k++; j++;
while i <= mid-1 then
    temp[k] = P[i]
    k++; i++;
while j <= right then
    temp[k] = P[j]
    k++; j++;
for m ← left to right do
    P[m] = temp[m]
return count
```

```
MergeSort(P, temp, left, right)
count ← 0
if right > left then
    mid ← (right + left) / 2
    count ← MergeSort(P, temp, left, mid)
    count ← count + MergeSort(P, temp, mid+1, right)
    count ← count + MergeCount(P, temp, left, mid+1, right)
return count
```

```
Incompable (set, size)
for i←0 to size do
    P[i] = set[i]
sort x field in P in ascending order
return MergeSort (P, temp, 0, size-1)
```

complexity = MergeCount = $O(n)+O(n)+O(n)+O(n)=O(n)$
MergeSort = $O(n\log n)$
Incompable = $O(n\log n)$

Explain: This algorithm is based on inversions in Q1. Firstly, we sort all points by x field in ascending order. Then we divide into small units by merge sort (divide into half and half). We compare two pieces by MergeCount to count how many inversions in those sorted points, expect those have the same x value. This idea comes that all comparable points are $x_i \geq x_j$ and $y_i \geq y_j$ and visa versa. Then for incomparable points, since we have sorted x, $y_i < y_j$ each time count one incomparable. After we got those special inversions, we merge together and count larger unit and so on. Therefore, the maximum running time based on mergesort, that is $O(n\log n)$.

4. a) MaxSpace (arr, size)

```
create a stack named s
i←0
while i<size do
    if s is empty or arr[s.top] <= arr[i] then
        push i into s
        i++
    else
        temp ← s.top
        pop the top from s
        if s is empty then
            area ← arr[temp] * i
        else
            area ← arr[temp] * (i - s.top - 1)
        if MaxA < area
            maxA ← area
while s is not empty
    temp ← s.top
    pop the top from s
    if s is empty then
        area ← arr[temp] * i
    else
        area ← arr[temp] * (i - s.top - 1)
    if MaxA < area
        MaxA ← area
```

complexity: Only one while loop, count O(n)

Explain: Firstly, we create a stack to store data, keep reading new incoming value. If the stack is empty or the incoming value is greater than current top of stack, we push the value into the stack. If the incoming value is smaller than current top of stack, means we should stop and clear the stack until top of stack is smaller than the incoming value. This ensures that merged height is a whole universe. In addition, when we are cleaning the stack, since poped value always <= top of the stack, and the incoming value always smaller than poped value, the area calculating must be the max area between poped values to top values. Therefore, we can get the maximum space when loop from i to the end.

4.b)

```
MaxSpace2(map, n)
 arr[n] ← 0
 max ← 0
 for i ← n-1 to 1 do
     for j ← 0 to n-1 do
         if Unit occupied then
             arr[j] = 0
         else if i == n-1 then
             arr[j] = 1
         else
             arr[j] += 1
     MaxArea ← MaxSpace(arr, n).
     if MaxArea > Max then
         Max ← MaxArea
return Max
```

Complexity: $\sum_{i=1}^{n}(\sum_{i=1}^{n} O(1) + O(n)) = O(n^2)$

Explain = We get occupied units block our continuous count, start from the top, compute the maximum space one horizontal line have. Then compare for the max from all maximum space. When we meet occupied, the height be assigned to be 0. when not, the height increases. This is the main idea.