CS341 HW1

2. a) $T(n) = T(2n/3) + T(n/3) + n^2$

induction hypothesis:

$T(n) \le c \cdot n^2$

base case:

$n=1 : T(1)=1$

induction step:

$$T(m) \le c \cdot \frac{4}{9}m^2 + c \cdot \frac{1}{9}m^2 + m^2$$

$$\le \frac{5}{9} \cdot cm^2 + m^2$$

$$\le cm^2 \quad \text{for all} \quad c > 9$$

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

b) $T(n) = n^{\frac{1}{2}} T(n^{\frac{1}{2}}) + n$

$$= n^{\frac{1}{2}}\left(n^{\frac{1}{4}} T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}\right) + n$$

$$= n^{\frac{1}{2}} n^{\frac{1}{4}} T(n^{\frac{1}{4}}) + 2n$$

$$= n^{\frac{1}{2}} n^{\frac{1}{4}} n^{\frac{1}{8}} T(n^{\frac{1}{8}}) + n^{\frac{1}{2}} n^{\frac{1}{4}} n^{\frac{1}{4}} + 2n$$

$$= n^{\frac{1}{2}} n^{\frac{1}{4}} n^{\frac{1}{8}} T(n^{\frac{1}{8}}) + 3n$$

$$\vdots$$

$$= n^{\frac{1}{2} + \frac{1}{2^2} + \dots + \frac{1}{2^i}} T(n^{\frac{1}{2^i}}) + i \cdot n$$

$$\vdots$$

$$= nT(1) + \log\log n \cdot n$$

$$= n + n\log\log n$$

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

3. Count pairs of comparables of A[1,n]

Algorithm:

```
Ax[1,n] = sortByX(A[1,n]); //O(nlogn)

countAndSortByY(Ax[1,n]) {
    if (n==1) return 0;
    leftCount = countAndSortByY(Ax[1,n/2]);
    rightCount = countAndSortByY(Ax[n/2+1,n]);
    crossCount = mergeSortByYAndCountAcross(Ax[1,n/2], Ax[n/2+1,n]);
    return leftCount + rightCount + crossCount;
}

mergeSortByYAndCountAcross(left[1,n/2], right[1,n/2]) {
    // right side's x is guaranteed to be ≥ left
    i=0, j=0, k=0, count=0;
    while (i≤n/2 && j≤n/2) { // merge and count
        if (left[i].y ≤ right[j].y) {
            Ax[k] = left[i];
            count += (n/2 - j);
            i++; //increment left counter
        } else {
            Ax[k] = right[j];
            j++; // increment right counter
        }
        k++;
    }
    //clean up rest of left or right
    for(i -> n/2) { A[k] = left[i]; k++; }
    for(j -> n/2) { A[k] = right[j]; k++; }
    return count;
}
```

Correctness:

Since the Array is sorted by x-values at the beginning of the algorithm, therefore when it is divided in the middle, it is guaranteed that the right side's x-values are greater than all of the left side's x-values.

By recursion, we can assume that the function "countAndSortByY" returns the number of pairs of comparables and sorts the array by y-values.

Therefore, when we apply merge-sort on left and right, if the left side's y-value is $\leq$ right side's y-value, then it is guaranteed that both the x and y values of left side is $\lesssim$ the x and y values of the right side. So we know that this left half's coordinate is comparable with every coordinate on the right hand side of the value we compared.

ex:

y-values:

2	3	4	9

1	6	7	8

Since 3 is less than 6, it is comparable to everything to the right of 6.

This is similar to the merge in the inversion counting algorithm.

Therefore we can count the number of pairs of comparables between left and right in $O(n)$ time.

By recursion, we will count the total number of inversions time complexity:

Initial time-complexity: $O(n\log n)$ from sorting

$$T(n) = \underset{\substack{\uparrow \\ \text{divide}}}{2T(\tfrac{n}{2})} + \underset{\substack{\uparrow \\ \text{merge}}}{cn}$$

By Master theorem, $a=2, b=2, c=1$

$1 = \log_2 2$

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

4. a)

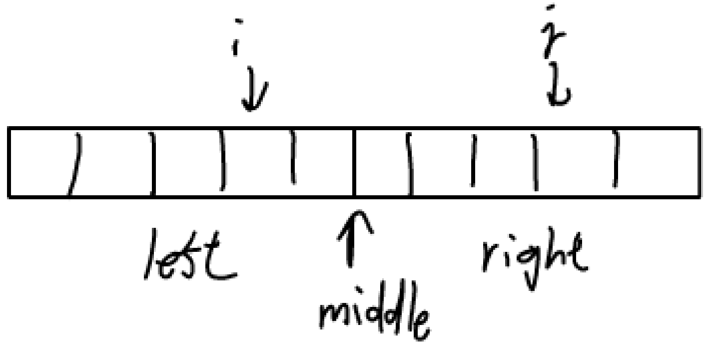
```
maxArea(A[1,n]) {
    if(n==1) return A[0];
    leftArea = maxArea(A[1, n/2]);
    rightArea = maxArea(A[n/2+1, n]);
    height = min(A[n/2], A[n/2+1]); // need the smaller height
    crossArea = height * 2;          // to get across the centre
    i = n/2 - 1; j = n/2 + 2;
    while(i ≥ 0 || j < n) {
        while(A[i] ≥ height) {
            i--;
        }
        while(A[j] ≥ height) {
            j++;
        }
        crossArea = max(crossArea, height * (j-i));
        height = max(A[i], A[j]); // new max height
    }
    return max(leftArea, rightArea, crossArea);
}
```

Correctness:

Similar to the maximum subarray problem, we divide the list into 2. The optimum solution must be contained within the first half, the second half, or across the middle.

The maximum area within the divided halves can be found recursively. We just need to find the maximum area across the mid-point.

This can be done by "expanding" out from the center and keep two pointers to the expanded indices:



We also keep track of the biggest area that the interval $i \to j$ can hold. As we expand i and j, we calculate a new area based on the maximum height of $A[i]$, $A[j]$. If the new Area is larger, we keep the new area. This ensures the largest possible area across the middle. Therefore The max area is simply the maximum of leftside, rightside, and across.

Time Complexity:

$$T(n) = \underbrace{2T(\tfrac{n}{2})}_{\text{dividing}} + \underbrace{O(n)}_{\substack{\text{finding} \\ \text{max area} \\ \text{across}}}$$

The finding max area across the middle is $O(n)$ because we traverse through each index exactly once.

By Master's Theorem: $a=2$, $b=2$, $c=1$

$1 = \log_2 2$

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

4b)
```
maxRectangle(G[1,n][1,n]) {
    Histogram[1,n][1,n]; //empty matrix
    for (int i=0; i<n; i++) {
        if (G[i][0] == Occupied) {
            Histogram[i][0] = 0;
        } else {
            Histogram[i][0] = 1;
        }
        for (int j=1; j<n; j++) {
            if (G[i][j] == Occupied) {
                Histogram[i][j] = Histogram[i][j-1]+1;
            } else {
                Histogram[i][j] = 0;
            }
        }
    }
    maxArea = maxArea(H[0]); // 4a's function O(n)
    for (int i=1; i<n; i++) {
        area = maxArea(H[i]); // 4a O(n)
        if (area > maxArea) {
            maxArea = area;
        }
    }
    return maxArea;
}
```

Correctness:

Notice that the grid can be viewed as n lists of histograms. Then we can simply apply 4a's $O(n)$ algorithm for each list, and then determine the maximum value of these values. The maximum value of these areas is guaranteed to be the maximum rectangle.

The n lists of Histograms can be constructed by going through the grid from top down, and calculate the height of a histogram at each coordinate.

Time Analysis:

To construct the histogram list, we have to go through every grid value, which is $N \times N = O(n^2)$.

To apply 4a's algorithm on n histograms, it takes $n \times O(n) = O(n^2)$

Therefore $T(n) = O(n^2) + O(n^2)$
$= O(n^2)$