A1 W2016

1. [Covered in notes]
Our algorithm works as follows. Given the array, we split the array in half. We find the number of inversions in the left half, then the number of inversions in the right half. Now we want to find the number of transverse inversions. We first sort the left half, and sort the right half of the array. Then we use mergesort to count the number of transverse inversions, as demonstrated in the following pseudocode:

```
mergeAndCount(A) {
    result = 0
    S = A // Copy A into new array S
    i = 0, j = n/2
    for (k = 0, k < n, k++)
        if (i > n/2) A[k] = S[j++]
        else if (j > n) A[k] = S[i++]; result += n/2
        else if (S[i] < S[j]) A[j] = S[i++]; c += j - n/2
        else A[k] = S[j++]
}
```

2. [A1 problem]
A) We have that T (n) = T (2n/3) + T (n/3) + n^2 ≥ n^2 for all n ≥ 0. Hence we have that T(n) ∈ Ω(n^2).

We show that T (n) ∈ O(n^2). Consider the recurrence S(n) = T(2n/3)+T (2n/3)+n^2 = 2T(2n/3)+n^2. Using the Master Theorem, we have a = 2, b = 3/2 , and c = 2. We have that log_b(a) = log_3/2(2) ≈ 1.71 < c. Hence by the Master Theorem, S(n) ∈ Θ(n^2) and thus S(n) ∈ O(n2). Since for all n ≥ 0 we have that T(n) ≤ S(n), we have that T(n) ∈ O(S(n)). Hence we have that T(n) ∈ O(n^2).

Since we have T(n) ∈ Ω(n^2) and T(n) ∈ O(n^2), we have that T(n) ∈ Θ(n2).

B) At level k of the recursion tree, there are sqrt(n) * n^1/4 *…*n^1/2k = n^(1-2^-k) nodes with each node having n^(1/2k) work. Hence the total work on level k is n.

When k = loglogn, the subproblem size is n^1/2k = 2. Hence the total work done is at most n*k = nloglogn, so we have that T(n) ∈ $O(\text{nloglogn})$

3. Our algorithm works similar to counting inversions. We first sort the pairs by x-coordinate. Then we use a modified merge sort algorithm on the y-coordinates. We merge sort the left and right halves recursively. We have that all the points in the left half of the array have a smaller x-coordinate than all the points in the right half. Then during merge, for each point in the left half of the array, we can count the number of points in the right half that have a smaller y-coordinate. The pseudocode for this merge and count is given below.

```
mergeAndCount(A) {
    result = 0
    S = A // Copy A into new array S
    i = 0, j = n/2
    for (k = 0, k < n, k++)
```

```
        if (i > n/2) A[k] = S[j++]
        else if (j > n) A[k] = S[i++]; result += n/2
        else if (S[i] < S[j]) A[j] = S[i++]; c += j - n/2
        else A[k] = S[j++]
}
```

Runtime: It takes O(nlogn) time to sort the pairs by x-coordinate. Our modified mergesort takes O(nlogn) time. Hence our algorithm is O(nlogn) time.

4. [A1 problem]

A) We use a divide and conquer algorithm to find the maximum rectangular area. If there is only one building, we return the height of the building. For a range of buildings, the maximum area must be either
(1) Contained in the left half of the buildings.
(2) Contained in the right half of the buildings.
(3) The maximum area spans the 2 middle buildings.
Hence at each recursive step, we recurse in the left half of the buildings, then recurse in the right half of the buildings, then find the maximum area containing the 2 middle buildings. Then we return the maximum of the 3.

To find the maximum area containing the 2 middle buildings, we start with the maximum area of a poster contained in these 2 buildings. At each iteration, we either add the taller building from the left or the right side. The current area then becomes the height of the currently shortest building multiplied by the width of the range of buildings. We iterate until our range now spans all the buildings, keeping track of the maximum area as we iterate. Since this process iterates through each building exactly once, it takes $\Theta(n)$ time, where n is the number of buildings.

Runtime: Recursing in the left and right halves take $T(n/2)$ time each (sloppy). Hence the recurrence relation for this algorithm is $T(n) = 2T(n/2) + \Theta(n)$. Using the Master Theorem, we have $a = 2$, $b = 2$, and $c = 1$. We have that $\log_b(a) = \log_2(2) = 1 = 1\ c$. Hence by the Master Theorem,

$T(n) \in \Theta(n \log n)$

Correctness: There are only 3 possibilities for where the largest rectangle is: we compute the maximum of all 3. We prove correctness for the algorithm to compute the area of the largest rectangle spanning the middle two buildings. The height of the rectangle has to be less than or equal to the height of the two middle buildings. The algorithm calculates the maximum area of the rectangle with height less than the height of the midpoint. For every height $h(i) \leq h(mid)$ the algorithm finds the smallest index to the right of the midpoint with a height less than h(i) and the largest index to the left of the midpoint with a height less than h(i).

Bonus

We find the maximum area as follows. For every building k, we calculate the maximum area where some i is the shortest building. Then we take the maximum of all of these. The maximum area must have the height of the shortest building in its range. Hence this method calculates the maximum area.

For each building k, we must know the index of the first shorter building to its left, i, and the index of the first shorter building to its right, j. Then the maximum area with building k as the shortest building must be the height of k times the width $j - i$.

To do this, iterate through the buildings from left to right, maintaining an increasing stack of buildings (the height of buildings in the stack is monotonically increasing). At each iteration, we pop off the stack any buildings that are taller than the current one. For each popped item k, we calculate the maximum area where k is the shortest building. The first shorter building to its left must be the current top of the stack, since we maintained our stack to be increasing. The first shorter building to its right must be the current building. We can calculate the area using the difference between indices for the width and the height of the popped building. Then we push the current building to the stack. We iterate until we have reached all the buildings.

Runtime: Every building is pushed to the stack once, and every building is popped from the stack at most once. Hence the runtime of this algorithm is $T(n) \in O(n)$.

B)
We iterate through each row from top to bottom. For each row, we find the maximum area where the rectangle's bottom edge is along our current row. This is synonymous with part(a): the height of each "building" is the height of the column until we reach an occupied unit, and we want to find the maximum area of the buildings.

For each row, we must calculate the height of each "building" with the current row as the ground. This is straightforward for the first row: it is simply 0 or 1 depending on whether that unit is occupied. We keep track of these heights. For the next rows, we simply iterate through all the units in the current row. If the unit is unoccupied, we increment the height of the building by 1. If it is occupied, the height is 0.

Runtime: For each row, this operation takes $\Theta(n)$ time, since we iterate each unit once. For each row, we also run the algorithm in part (a) once, which also takes $O(n)$ time. Hence the amount of work on each row is in $O(n)$ time. There are n rows, so this algorithm takes $n \cdot O(n)$ time which is in $O(n^2)$ time.

Correctness: The maximum rectangle must have its bottom edge along some row. Since we iterate through every row, and the algorithm from part (a) is correct, our algorithm will find the maximum area.