5. (25%)

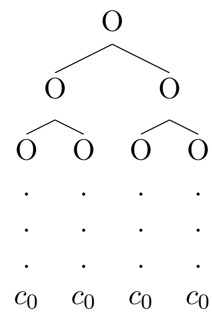
(a) Write a recurrence for the running time of the modified closest-pair algorith, in terms of the number of points n.

Because the Sorting Algorithm we are using is $\Theta(n \log n)$, the recurrence relation is:

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 2 \\ 2T(\frac{n}{2}) + bn + cn\log n & \text{if } n > 2 \end{cases}$$

(b) Sketch the recursion tree for this recurrence and derive a non-recursive (but not necessarily closed-form) exact expression for its solution.

The recursion tree looks like:



where the work per node is given by $\frac{b}{2^k} + \frac{cn}{2^k}\log(\frac{n}{2^k})$ for levels $0, ..., \log n - 2$. The work per level can be generalized to be $2^k(\frac{b}{2^k}\frac{cn}{2^k}\log(\frac{n}{2^k})) = bn + cn\log(\frac{n}{2^k})$. Summing the total work, this looks like:

$$\begin{aligned}
T(n) &= c_0\frac{n}{2} + \sum_{k=0}^{\log n - 2}(bn + cn\log(\frac{n}{2^k})) \\
&= c_0\frac{n}{2} + b\frac{(\log n - 1)(\log n - 2)}{2} + cn\sum_{k=0}^{\log n - 2}\log(\frac{n}{2^k}) \\
&= c_0\frac{n}{2} + b\frac{(\log n - 1)(\log n - 2)}{2} + cn\sum_{k=0}^{\log n - 2}(\log n - \log 2^k) \\
&= c_0\frac{n}{2} + b\frac{(\log n - 1)(\log n - 2)}{2} + cn(\log n)(\log n - 1) - cn\log 2\sum_{k=0}^{\log n - 2} k \\
&= c_0\frac{n}{2} + b\frac{(\log n - 1)(\log n - 2)}{2} + cn(\log n)(\log n - 1) - cn\frac{(\log n - 2)(\log n - 1)}{2} \\
&= c_0\frac{n}{2} + (\log n - 1)(\log n - 2)(\frac{b}{2} + \frac{cn}{2}) \\
&= c_0\frac{n}{2} + b'((\log n)^2 - 3\log n + 2) + c'n((\log n)^2 - 3\log n + 2) \\
T(n) &= c_0\frac{n}{2} + b'((\log n)^2 - 3\log n) + b'' + c'n(\log n)^2 - c_1'n\log n + c''
\end{aligned}$$

(c) Show that the expression you got in part **(b)** is $\Theta(n(\log n)^2$.
Taking the limit of $\frac{T(n)}{n(\log n)^2}$ it is clear that:

$$\lim_{n\to\infty}\left(\frac{T(n)}{n(\log n)^2}\right) = c < \infty$$

where c is an arbitrary constant not pulled from the previous relations. Therefore:

$$T(n) = \Theta(n(\log n)^2)$$

(d) Professor Strammermax claims that the running time of the new algorithm can be reduced to $\Theta(n \log n)$. The key idea is to *reconstruct* the sorted `ptsByY` array dynamically inside the algorithm.
Suppose that the two recursive calls in the algorithm are modified to return both the closest pairs on left and right *and* two arrays containing all the left and right points, respectively, each sorted by y-coordinate. (Clearly, we can compute such a sorted array in constant time when $n \leq 2$.) Describe in pseudocode how to combine these two arrays in time $\Theta(n)$ to produce an array of *all* input points sorted by y. Justify the correctness and running time of your solution.

Two arrays, each sorted by y value, can be combined with the following algorithm where `yLeft` is the y-sorted algorithm for the left side and `yRight` is the y-sorted algorithm for the right side.:

```
CombineArrays(yLeft,yRight,n)                        ▷ n = yLeft.length + yRight.length
  leftCounter← 0
  rightCounter← 0
  j ← 0
  while j < n do                                                        ▷ n+1 times
    if leftCounter<yLeft.length-1 and rightCounter-1<yRight.length      ▷ n times
      if yLeft[leftCounter].y < yRight[rightCounter].y            ▷ at most n/2 times
        combinedArray[j]=yLeft[leftCounter]                       ▷ at most n/2 times
        leftCounter++                                             ▷ at most n/2 times
      else
        if yLeft[leftCounter].y > yRight[rightCounter].y          ▷ at most n/2 times
          combinedArray[j]=yRight[rightCounter]                   ▷ at most n/2 times
          rightCounter++                                          ▷ at most n/2 times
        else                     ▷ always choose the point from yLeft if points equal
          combinedArray[j]=yLeft[leftCounter]
          leftCounter++
    else      ▷ These statements run n/2 times if yLeft and yRight have the same length
      if leftCounter=yLeft.length-1
        combinedArray[j]=yLeft[leftCounter]
        leftCounter++
      else                                      ▷ i.e. rightCounter=yRight.length-1
        combinedArray[j]=yRight[rightCounter]
        rightCounter++
    j++                                                                  ▷ n times
  return combinedArray
```

Note that each statement in the algorithm above runs either a constant or linear, because each operation is a single step, and we only run a single iteration loop. Therefore, each step runs at most $n+1$ times, and `combinedArrays`$=\Theta(n)$.
Correctness follows from inspection. Note that all the points in `yLeft` will always have lower x coordinate values than the points in `yRight`. The algorithm will always put the lower $y-$value from

the next available index in either `yLeft` or `yRight`. If all of values from either `yLeft` or `yRight` have already been used, this algorithm will simply fill the remaining slots in `combinedArrays` with the remaining values from the other array.