This write-up illustrates what is expected by way of a solution for a problem involving the design and analysis of an algorithm (cf: Instructions for Hw1). You may wish to model your answers along these lines. This is meant to be illustrative only, and is not set in stone, so feel free to modify this for your purposes. The key is to generate an answer that communicates the main ideas effectively and is easy for others to understand.

**Problem:** Let $S$ be a set of $n$ distinct points, $p_i = (x_i, y_i), 1 \leq i \leq n$, in the plane. A point $p_j \in S$ is a maximal point of $S$ if there is no other point $p_k \in S$ such that $x_k \geq x_j$ and $y_k \geq y_j$. See the figure; note that the maximal points form a “staircase”.

![Maximal and Non-maximal Points](image)

Give an efficient divide-and-conquer algorithm to determine the maximal points of $S$.

Your answer should include (i) a clear description of the main ideas and the data structures used, which makes the correctness self-evident, (ii) pseudocode for the algorithm (at the level of detail seen in the text), and (iii) an analysis of the running time and space used.

**Solution:**

**Key ideas for the algorithm:** We will compute the maximal points recursively, using divide-and-conquer. We maintain the invariant that the maximal points returned for any subset of $S$ are sorted by decreasing $y$, hence, by definition of maximal points, also sorted by increasing $x$. (Note that, by definition, no two maximal points can have the same $x$- or the same $y$-coordinate.)

Pre-sort the points of $S$ by non-decreasing $x$; if two points have the same $x$-coordinate, then break the tie in favor of the one with the smaller $y$-coordinate. (Since the points of $S$ are distinct, if two points have the same $x$-coordinate, then their $y$-coordinates must be different.) If $S$ contains only one point, then return that as the sole maximal point of $S$. Otherwise, partition $S$, by $x$, into two equal-sized halves and recursively compute the maximal points of each half. Scan the maximal points of the left half by increasing $y$ and discard all points that are not higher than the highest maximal point of the right half; for such points, both coordinates are, respectively, no larger than those of the highest point of the right half, so they cannot be maximal in $S$. Return, as the maximal points of $S$, any remaining maximal points of the left half followed by all the maximal points of the right half. Note that the returned set satisfies the invariant.

There are a couple of ways of implementing the algorithm efficiently (e.g., using linked lists or arrays). We choose to store $S$ in a global array $S[1 : n]$, by non-decreasing $x$. ($S[1 : n]$ denotes the
locations $S[1]$ through $S[n]$. We also maintain a global array $M[1 : n]$ which stores maximal points as follows: When we compute the maximal points of $S[i : j]$, where $1 \leq i \leq j \leq n$, we store these points in $M[i : k]$ by decreasing $y$ (hence increasing $x$) where $k \leq j$ and $k-i+1$ is the number of maximal points of $S[i : j]$. (In the process, previously-computed maximal points of any subsets of $S[i : j]$ are destroyed as they are no longer needed.)

---

Pre-sort the input points by non-decreasing $x$-coordinates into a global array $S[1:n]$ using, say, Mergesort. If two points have the same $x$-coordinate, then break the tie in favor of the point with the smaller $y$-coordinate.

$k \leftarrow$ MAXIMAL-POINTS(1,n) // The maximal points of $S[1:n]$ are returned in $M[1:k]$ in decreasing $y$ order

MAXIMAL-POINTS(i,j)
// Computes the maximal points of $S[i:j]$ and stores them in a global array $M[i:k]$, by decreasing $y$, where $k <= j$ and $k-i+1$ is the number of maximal points. Returns $k$.
if (i=j) then store $S[i]$ in $M[i]$ and return(i)

$m \leftarrow$ floor((i+j)/2)

$s \leftarrow$ MAXIMAL-POINTS(i,m) // $M[i:s]$ stores maximal points of $S[i:m]$

t \leftarrow$ MAXIMAL-POINTS(m+1,j) // $M[m+1:t]$ stores maximal points of $S[m+1:j]$

MERGE(i,s,m+1,t)
end // MAXIMAL-POINTS

MERGE(a,b,c,d)
// Combines the maximal points in $M[a:b]$ and $M[c:d]$ by scanning $M[a:b]$ in increasing $y$ order from the lowest point, i.e., $M[b]$, and discarding points that are not above the highest point of $M[c:d]$, i.e., $M[c]$.

$u \leftarrow b$
while((a <= u) & (M[u].y <= M[c].y)) $u \leftarrow u-1$
// Points in $M[a:u]$ remain maximal after the merge. (Note that $M[a:u]$ is empty if the condition "a <= u" becomes false, i.e., if u = a-1.)
// Copy the points from $M[c:d]$ to $M[u+1:a-d-c+1]$ and return $u+d-c+1$
for i \leftarrow 1 to d-c+1 do

$M[u+i] \leftarrow M[c+i-1]$

return(u+d-c+1)
end // MERGE

---

Analysis: The pre-sorting takes time $\Theta(n \log n)$. The time to divide $S$ into two halves is $\Theta(1)$. The time to merge the maximal points of the two halves is upper-bounded by the sum of their sizes, which is $\Theta(n)$ in the worst case. Let $T(n)$ denote the time taken by MAXIMAL-POINTS(1,n). Then, $T(n) = 2T(n/2) + \Theta(n)$, if $n > 1$, and $T(n) = \Theta(1)$ else. By the Master Theorem, $T(n) = \Theta(n \log n)$, and the total time, inclusive of the pre-sorting, is of the same order.

The storage used by the algorithm is $\Theta(n)$, since the arrays $S$ and $M$ are of size $n$. (The storage used by the rest of the algorithm is $\Theta(1)$.)