The running time of the algorithm models a recurrence of the form $T(n) = 2 \cdot T(\frac{n}{2}) + O(n \cdot log(n))$. The reason being that we have two subproblems of size $\frac{n}{2}$. The combining portion is $O(n \cdot log(n))$ because the bottleneck is in doing two quicksorts, one for the left and another for the right half of the subarrays. Setting up a table and solving the system.

level	size of a problem	cost of a node	# nodes	row sum
0	n	$n \cdot log(n)$	1	$n \cdot log(n)$
1	$\frac{n}{2}$	$\frac{n}{2} \cdot log(\frac{n}{2})$	2	$n \cdot log(\frac{n}{2})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$\frac{n}{2^i}$	$\frac{n}{2^i} \cdot log(\frac{n}{2^i})$	2^i	$n \cdot log(\frac{n}{2^i})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	1	1	2^k	2^k

$$\begin{aligned} \frac{n}{2^k} &= 1 \\ k &= log_2(n) \end{aligned}$$

$$\begin{aligned}
\sum_{i=0}^{k-1} n \cdot log(\frac{n}{2^i}) + 2^k &= n \cdot \sum_{i=0}^{k-1} log(\frac{n}{2^i}) + 2^k \\
&= n \cdot \sum_{i=0}^{k-1} (log(n) - log(2^i)) + 2^{log_2(n)} \\
&= n \cdot \sum_{i=0}^{k-1} log(n) - n \cdot \sum_{i=0}^{k-1} log(2^i) + n \\
&= n \cdot log(n) \cdot (k - 1 - 0 + 1) - n \cdot \sum_{i=0}^{k-1} i \cdot log(2) + n \\
&= n \cdot log(n) \cdot log(n) - n \cdot \frac{(k-1) \cdot (k)}{2} + n \\
&= n \cdot log^2(n) - n \cdot \frac{(log(n) - 1) \cdot (log(n)}{2} + n \\
&= \Theta(n \cdot log^2(n))
\end{aligned}$$

Thus the algorithm runtime is $\Theta(n \cdot log^2(n))$.

The reason the MPSS middles works is similar to the analysis used for the divide and conquer version of the maximum contigious subarray problem.

We assume that $MPSS(A[low \dots mid])$ and $MPSS(A[mid+1 \dots high])$ have been found based on the divide and conquer nature of the algorithm but it still does not take into account the crossing section of the two subproblems. that is find the $MPSS(A[low \dots mid \dots hi])$. We first find all subsequences of the form $A[i \dots mid]$ where $low \leq i \leq mid$, we denote this S_L and sequences $A[mid+1 \dots j]$ where $mid+1 \leq j \leq hi$, denote this S_R. We find and store this subsequences sum because any crossing would be comprised of two of the sums from S_L and S_R respectively. Now because it does not matter whether a subsequence is contigious, we sort S_L in ascending order and S_R in descending order. The reason being it now becomes easy to find the smallest positive subsequence sum out of S_L and S_R. concretely, if $S_L[i]+S_R[j] \leq 0$ then since S_L is in sorted order we can increase the index i to possibly get a valid candidate for smallest positive subsequence sum. if $S_L[i] + S_R[j] > 0$ then we check if S_{min} if larger than the one we have currently found and if so we update it. We then increment j since S_R is sorted in descending order we could find a possibly smaller sum. This continues until i or j become out of bounds which the case where we have ran out of valid candidate for the minimum positive subsequence sum and we return the candidate we found if any.