Mercy Bickell and Kristine Cho
Problem Set 11 - Problem 2

Algorithm Description:
This algorithm uses a dynamic programming approach to solving the problem. It fills in the dynamic programming table from the block with the largest base area to the smallest. It checks whether the current block being checked can be stacked on each of the blocks before it and keeps track of the maximum height for the tower that has to contain the current block. The tower is the set of blocks with the maximum height overall.

Proof of correctness:
To prove its correctness, let us first prove that this problem has an optimal substructure.

Proof by contradiction:
Let $b_1, b_2, b_3, \ldots, b_m$ be all possible block types, sorted from largest base area to smallest base area.
Let $\{i_1, i_2, i_3, \ldots, i_m\}$ be the indices in the array.
Assume that $\{i_1, i_2, i_3, \ldots, i_m\}$ is a solution for blocks $b_1 \ldots b_m$ but that $\{i_2, i_3, \ldots, i_m\}$ is not a solution for blocks $b_{i2} \ldots b_m$.
Then there exists some solution to $b_{i2} \ldots b_m$, $\{i'_2, i'_3, \ldots, i'_n\}$ such that the sum of the heights of $b'_{i2}, b'_{i3}, \ldots, b'_i n$ is greater than the sum of the heights of $b_{i1}, b_{i2}, b_{i3}, \ldots, b_{im}$.
If that were true, we could create a tower $\{i_1, i'_2, i'_3, \ldots, i'_n\}$ to the original solution that is a better solution. Contradiction.

The recursive solution is:

$$T(i) = height(b_i) + \max_{length(b_j)<length(b_i) \ \& \ width(b_j)<width(b_i)} T(j) \tag{1}$$

The dynamic programming solution can be found in the submitted code.

Analysis:
The expected running time is $\Theta(n^2)$, where n is the length of the array containing all possible blocks and their configurations. This is because the algorithm goes through each block and, for each block, can possibly check all the blocks before it.

Design choices:
We decided to create a new data structure called a BlockPair to save the pointers. We chose this as opposed to turning the array into a matrix because it simplified the code and made it more intuitive.
We created a class called Block that took a block type (length, width, and height) and

created all possible permutations for it. It implemented the Comparable interface and was used to sort all possible block types.
The BlockPair class was used to trace back the tower so that we could find out which blocks were used for the maximum height tower. It stored the maximum height of the tower containing that block and a pointer to the index of the block right below it.
The method $maxHeightDP$ uses a dynamic programming approach to solving the problem. Our overall algorithm is as follows. First, we created all possible permutations of the blocks. We then sorted these blocks from the largest base area to the smallest base area, because only strictly smaller blocks can be stacked on larger blocks. We then filled in the DP table (an array of size n) from largest block to smallest block. For each block, we calculated the maximum height of the tower that had to contain that block. I.e., we checked whether it could be stacked on each of the blocks before it and, if it could, we added the height of the current block to the height of the tower that included the other block.

Testing:
We tested our algorithm using the test input given in the problem set. Our results can be found in the Github repository in the file called "test1output.text".

Acknowledgements
Thank you to all the mentors for all of your help!