Describe the algorithm that you implemented, argue that it is correct, and argue its expected running time

We start by taking every possible orientation of our blocks and storing it in some data structure. We can say, without loss of generality, that we will always orient the blocks such that length > width, since rotations just in the horizontal plane do not affect our ability to stack. So, if we have n blocks, we have 3n orientations. Then, we sort this data structure in decreasing order by area of the base (width x length). We do this to help ourselves in the future – we know that we can only stack a block on top of another if the base of the bottom block is bigger than the base of the top block.

From there, we can construct two one-dimensional arrays of length 3n. One array, which we'll call T, will track the maximum height we can currently stack. That array is initialized with all 0's. The other array will track which block has been stacked on what other block. We'll call this array P. It is initialized such that P[i] = i.

Let's call our algorithm stack. We'll call the function that gets the height of the block at index i h(i). stack(i) is defined as follows:

stack(i) = { - infinity if i > 3n or i < 0
max (max $_{1 <= j < i,\ w(i) < w(j),\ l(i) < l(j)}$ (h(i) + stack(j), h(i)))

This function only provides the max height of the stack, and will not point out what block is actually stacked where. So, this function only works with array T and does not fully represent what the code does. The code will also track the order of stacking in array P.

Our stack function for index i checks to see if the block at index i could be stacked on any of the blocks between indices 0 and i – 1. It then sees if adding block i to the stack is better than any other stack block i is involved in by taking the max of the heights of all of those potential stacks. The outer max function comes into play when block i cannot be stacked on any of the blocks at a lower index, or if the height of that index hasn't been filled in yet.

When this function is done, the height of our stack could be anywhere in the array because we know essentially nothing about what stacking actually occurred. To retrieve it, we must find the max of the array.

This is correct because the sub problem always has the max possible height. If we remove a block off the top of our stack and don't consider that block or any smaller blocks an option, we still have the max possible height of our stack. This is because we're considering the blocks in sorted order, meaning we are looking at no blocks with smaller areas than the block we just threw out, and also because we are always looking at all possible stacks of the blocks we do have.

We also rely on the optimal substructure of the problem. Our proof of that property is as follows: Consider stack(i). Assume that the optimal solution to stack(i) is some series of blocks b_1, b_2, …, b_{k-1}, b_k. Now assume that block b_{k-1} can be reached in some way that provides a higher stack than stack(i). Let's call this series of blocks c_1, c_2, …, c_r, b_{k-1}. We can then stack b_k back on top of b_{k-1}, since we know both the length and width of b_k must be smaller than those of b_{k-1}, given that the stacking of b_k on b_{k-1} happened in stack(i). So, since b_k cannot have a height of 0, this new stack must have a greater height than stack(i) and thus stack(i) did not actually have the optimal solution. Therefore, we have reached a contradiction.

Now we'll talk about array P. Any time the algorithm determines it has made the optimal block stacking, it will put the index of the block being stacked on at the index of the block that is being stacked. Essentially, if the block at index i is being stacked on the block at index j, P[i] becomes j. This creates a way to track what stacks occur. At the end of the main function, when the height has been computed, we can then trace the stacks back through array P. Let's say the height of our stack can be found at index x in array T. Then, we can start at P[x]. If P[x] != x, the block has been stacked so we must save the index in some way to keep track of the order of blocks. If P[x] = x, the block has not been stacked since the array was initialized such

that P[i] = i. Each time P[x] != x, visit index P[x] and repeat this process until the index does not equal the value. When this happens, we will have a list built up of the order of blocks used. The final list of blocks used can then be pulled from our initial list of blocks by checking each index.

Run time:
The expected run time of this algorithm is $\Theta(n^2)$. At every index i in T, our algorithm will look at every index between 1 and i, performing a constant time lookup in our table and potentially an addition. There is also a linear time max value comparison once all of the lookups have been made, but this does not affect the asymptotics of the algorithm.

Describe an interesting design decision that you made (i.e. an alternative that you considered for your algorithm and why you decided against it).

We originally were thinking about how to use a two-dimensional table where the axes are width and height, but we determined that using a table like that would be wasteful because there are a limited number of blocks, so only part of the table would be filled in.

We also originally had six possible orientations for each block, but we realized that was unnecessary because just swapping the length and the width doesn't provide a helpful new block.

An overview of how the code you submit implements the algorithm you describe (e.g. highlights of central data structures/classes/methods/functions/procedures/etc . . .)

The main dynamic programming algorithm is implemented in the `stack_blocks` method, which calls the `normalize`, `fill_table` (calling the recursive function `stack`), and `get_soln` methods. The remaining methods, `can_add`, `surface_area`, and `stringify` are helper methods, with `main` managing IO.

Blocks are represented as `tuple`s(immutable arrays) of their dimensions. The first represents the width, the second the length and the last the height. The table is represented as a `list`, as is the sequence we use to recover the solution. Below Array and `list` (formally Dynamic Array) are used interchangeably (as opposed to Linked List).

`stack_blocks`
: This function takes in a `list` of `tuple`s, each of which represents the dimensions of the types of blocks. Then we call `normalize`, which precomputes the valid permutations of block types, and sorts them according to `surface_area`. Then we initialize the DP table and the solution `list` and call `fill_table`, which fills in the table and solution array. We then call `get_soln` which does fixpoint iteration to recover the best solution.

`normalize`
: Find all unique permutations for the input `list` of `blocks`. Filter those such that the width is strictly smaller than the length. Then return the `list` sorted by the `surface_area`.

`fill_table`
: Takes in the DP `table`, the `list` of `blocks`, and the zeroed `soln` array and calls `stack` on each index of `blocks`.

`stack`
: Takes in the DP table, the `list` of `blocks`, the `soln` array and the current index `i`, and executes the previously described function `stack` on it using a Greatest Of All Time maximization method. It stores the results in `table` and `soln`.

`get_soln`
: The recursive algorithm to retrieve the solution from the `table`, `blocks` and `soln` `list`s. It does this by finding the index of the maximum of the table. This is the best solution. Then it maintains a list of the `blocks_used` as it performs a fixpoint traversal of the solution array. As it traverses these indices, it adds the block at each traversed index to the `blocks_used` list. It then reverses the list and returns it along with the height.

`can_add`
: Takes two blocks, and tests if we can add the second on top of the first.

`surface_area`
: calculates the surface area of an input block (width times length).

`stringify`
: turns a block into a space-separated list.

How you tested your code and the results of sample tests

To test the functionality of the algorithm, we loaded the file into the python2 REPL and gave the function stack_blocks certain lists of dimensions. Then we came up with a couple of interesting corner cases for testing that we ran through the IO interface.

No Stacking

input:

```
4
2 2 3
2 2 4
2 2 5
2 2 6
```

Stdout:

```
The tallest tower has 1 blocks and a height of 6
```

output:

```
1
2 2 6
```

All Same

input:

```
5
1 2 3
1 2 3
1 2 3
1 2 3
1 2 3
```

Stdout:

```
The tallest tower has 2 blocks and a height of 4
```

output

```
2
3 2 1
2 1 3
```

Uniformly sampled input:

```
8
18 8 5
14 11 21
21 3 7
8 21 27
5 13 2
25 5 13
11 3 21
25 24 11
```

StdOut:

```
The tallest tower has 7 blocks and a height of 124
```

Output:

```
7
27 21 8
25 13 5
24 11 25
21 8 27
13 5 25
11 3 21
5 2 13
```