— —

Instructor:

Homework 11 (70 points)

Due: 11:00AM on Tuesday, March 10 + Thursday, March 12

Danny Tamkin and Jack Weber

You can work alone or with a partner of your choice. You do not have to work with the same partner for both parts.

1 Block Stacking Documentation

1. describe the algorithm that you implemented, argue that it is correct, and argue its expected running time.

   The algorithm that we implemented solves the block stacking problem through dynamic programming. We start with a set of n block types each with three dimensions: width, length, and height (w, l, h). We then thought about how many possible blocks could be created from each of the block types where a block is defined as a unique orientation. One way that we could have gone about it would have been to say that there are 3! blocks for ever block type. Instead, we decided that a block is not unique if just the width and length are switched. That's to say for any block type (a, b, c), there are three unique block types: (a, b, c), (b, c, a), and (c, a, b). The block (b, a, c) is identical to (a, b, c).

   So, for n block types, there are $3n$ possible blocks, called the set of B blocks. Generating this set occurs in $\theta(n)$ time since the algorithm simply needs to generate three blocks (which occurs in constant time) for each of the n block types.

   The next step is for each block b in B to consider all of the possible blocks that *could* be stacked on top of b. For a single b, finding a set of blocks that could be stacked on top occurs in $\theta(n)$ time. That is because we need to go through all of the $3n - 1$ blocks to check if they can fit on top of b. This process is repeated for all $3n$ blocks giving it a total runtime of $\theta(n^2)$.

   At this point we have a set B of all of the blocks $b_0, b_1, \ldots b_{3n}$, each associated with a set of blocks that can fit on top of it. So b_i is associated with s_i where s_i is a set of blocks that fit on top of b_i. The algorithm then sorts B by the number of blocks that can fit on top of it. So any b_i with an associated s_i containing 0 elements will end up at the beginning of the sorted list. We use python's built-in timsort algorithm which can sort in $O(nlogn)$.

   We can now finally begin generating our dynamic table. We build a hashtable (dictionary) where the keys are the $3n$ blocks that could be the base of the stack. For a key

b_i, representing a potential base of the stack, the values are the maximum possible height with b_i as a base and the optimal block (if any) to put on top of the base to achieve this maximum height. Iterating through the sorted list of blocks, we use the following algorithm to fill the hashtable H:

For a block b_i with an associated set of blocks that fit on top s_i:

If s_i is empty, $H[b_i].height = b_i.height$ and $H[b_i].onTop = \{\}$

Else, $H[b_i].height = b_i.height + maxHeight(s_i)$ and $H[b_i].onTop = blockWithMaxHeight(s_i)$

Where $maxHeight(s)$ and $blockWithMaxHeight(s)$ find the maximum height and block with max height (respectively) from a set of no more than $3n - 1$ blocks by performing constant time lookups in H. Thus in the worst case, filling the hash table for a given b can take $\theta(n)$ time. For all $3n$ possible bases, filling the full hashtable has a runtime of $\theta(n^2)$.

We then find the base in the hashtable with the maximum height and trace the block on top associated with the block into we get to a block with no block on top, storing each of the blocks along the way. Finding the max height takes $\theta(n)$. Tracing takes $\theta(n)$ as well.

In total, we can ignore the $\theta(n)$ and $O(nlogn)$ run times and say that the algorithm's total run time is $\theta(n^2)$.

2. describe an interesting design decision that you made (i.e. an alternative that you considered for your algorithm and why you decided against it).
One design decision we made was deciding to implement the dynamic table as a hash table. We originally considered just using the indexes of the blocks which could be stored in an array. The height and block on top could have the same indexes as the blocks that they were associated with. We decided to use the hash table instead to avoid potential issues that could have arised when we shifted the ordering of the blocks during sorting. Since hashtables allow us to look up in constant time anyways, we figured the potential additional space complexity would be justified in order to reduce the risk of inconsistent indexing across multiple arrays.a

3. 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 . . .)
We described how the algorithm works in 1 so we will not repeat here. However, we will walk through the different functions that implement the algorithm.
The infile is read in and converted to a list of tuples in the **main** function where each tuple represents a block. **levelOneTwo** then takes this list of tuples and creates a list of dictionaries. Each dictionary has a block as well as the blocks that fit on top of them. This list of dictionaries is sorted by the number of blocks that fits on top and then **dpTableGenerator** creates the dynamic table implemented with a hash table.

The tallest stack is generated using **optimalStack** and then the outfile is created and written to by the **main** function.

4. how you tested your code and the results of sample tests

   We tested our code by creating infiles and expected outfiles associated with the infile. We then ran the main program from the terminal. The expected and actual outfiles were compared using a compareOutFiles program. This program compares each tuple, ignoring the ordering of width and length, and prints out the line where there is an inconsistency. Testing the example on the homework assignment as well as the test cases posted on Piazza, all of our tests came back reporting that the two outfiles matched.