Block Stacking Problem

Jay Rodolitz

October 2019

1 The Problem

You are given some number of types of blocks, each with specified height, width, and length (all positive integers), and the ability to orient each block as desired. You can use as many of each type of block as you would like, and a block can be placed in any of the three stable orientations. A block can be stacked on top of another block if and only if the two dimensions on the base of the top block are smaller than the two dimensions on the base of the lower block. What is the height of the tallest valid tower you can create, and what blocks does it contain?

2 Proof of Correctness

We can see that the problem has optimal substructure, as each tower contains some top block, and a tower below. We prove by contradiction, as follows. Imagine a tower with blocks from top to bottom $a_0, a_1, a_2, ...a_{n-1}, a_n$, of height H. Now, we will look at all towers which can contain an on top of them. Our solution constructs $a_0, a_1, a_2, ...a_{n-1}$, which has height $H - h(a_n)$. Assume there is some other, optimal tower which we can put an on, $s_0, s_1, s_2, ...s_n$, with height $>= H - h(_a n)$. Then, we can construct $s_0, s_1, s_2, ...s_n, a_n$, with height $>= H$, which means we have optimal substructure.

3 The Algorithm

Using this intuition I created an algorithm stack(k): return the height of the highest VALID stack of the first k blocks in a sorted list, which has k at the top of the stack. In a valid stack, no block is on top of a block smaller than it in width or height.

We require some preprocessing to get the provided input to the form where we can use the algorithm: In read_arr(), we take each block and expand it to all valid orientations, putting all the blocks in a list. Then we sort that list in order of decreasing block length (with width breaking ties), and make index 0 contain the floor (infinity x infinity x 0). From here, we can run the algorithm, stack(k, table), which is stack(k) as described, but puts values in a dynamic programming table. We can then call reconstruct() to reconstruct the solution.

The algorithm has two cases. In the base case, k = 0, and we return 0 - if you have no blocks, you have no height.

In the recursive case, you look at every block that the kth block could possibly go on top of, from 0 to k-1, and figure out which block you could choose to make the tallest stack, found by computing the max(stack(each valid index)). Add the height of k to that and you have the optimal height for k.

The optimal height for some whole array is the maximum value of stack(k) from k=0 to k = len(array) - the maximum possible height for any possible top

4 Complexity

The algorithm creates a 1 dimensional Dynamic Programming table of the max height of a tower topped by each possible block from the floor to the smallest block, where the floor is at index 0, and the smallest block is at the end of the array. It uses $\Theta(k)$ space to do so. The answer isn't stored in any particular place, but is the max value stored in the table.

Timewise, the algorithm operates in $\Theta(n^2)$: for each of the n array elements it looks back at each previous element, n operations.

5 Process

My code implements the described recurrence relation with the aforementioned preprocessing. It treats blocks of tuples of (l, w, h), and stores in the table a tuple of (max height, block directly below). It fills the table with calls from stack(0) to stack(k). To reconstruct the solution, the algorithm loops from the index of the block at the top of the largest stack, i= (indexOf(max(table))) to 0, saving that block and moving to the saved index.

A design decision I made here was to use a 1d table. My initial table, under the intuition that this problem was similar to the 0-1 knapsack (single use elements, keeping track of bases here, like weight there), was actually a 3 dimensional, k x len x width table, which would've run in... $O(n^9)$? Which I thought was, obviously, ridiculous. But I didn't understand how to do the 1d table on k for *days*, not till Xander and Aden explained that you actually were allowed to look at other elements of your list of blocks while operating on one of them without violating the "rules" of dynamic programming. I tested the code method by method on a small personal data set and it behaved as expected, and then I ran the provided 100 dataset and got no diffs.