Oct 29, 2019
Alex Franklin
Xander Koo

CS140 Problem Set 9 Question 3 Documentation

Algorithm

The blocks are put into a list containing all of their possible orientations. This list is then sorted by the length, then by width if length is equal, from largest to smallest. This ensures that no block can be stacked onto a block that comes after it.

Our recursive algorithm is defined as $A(i) =$ the tallest stack possible with blocks 0, 1, 2, …, i where i is at the top of the stack. For $A(0)$, our base case, this is simply the height of the first block on the list. Otherwise, we check all solutions $A(k),\ 0 \leq k < i$ where block i fits on top of block k and get the max of these, i.e. we get the maximum height across all stacks that have block $k,\ 0 \leq k < i$, on top. We can express the recursive structure as follows:

A(i) = { h_0 if i =0
{ h_j + max A(i), where 0 ≤ k < i and block i fits on block k otherwise

This assumes that no block $i-1$ can fit on top of block i.

Then the DP table is filled out from A(0), then A(1), …, A(i). The final answer is the maximum value in the DP table.

Correctness

Let us first prove the optimal substructure of the problem.

Proof by contradiction. Suppose you have an optimal stack of blocks $\{b_0,\ b_1,\ \ldots\ b_k\}$ that is the solution for k blocks, such that block k is at the top of the stack, any block i fits on top of block $i-1$, and the sum of heights of all blocks is the greatest possible. Then suppose that the subset $\{b_0,\ b_1,\ \ldots\ b_{k-1}\}$ is not the optimal height for a stack of blocks that is topped by block $k-1$. Then we could replace this solution with the optimal solution for the block $k-1$-topped solution into our k-block solution, which would give us a taller height than our optimal solution, which contradicts our supposition of optimality for our main problem. Therefore, by contradiction, the problem exhibits optimal substructure.

Since the problem exhibits optimal substructure, we can then use a recursive algorithm (and thus a dynamic programming solution) to solve the problem.

The recursive case considers all possible blocks that block i can fit on top of (since it cannot fit on top of a block after it in the list because the list is sorted) and finds the max of all of these cases. This is an exhaustive list of options so this is correct assuming that the recursive calls before it are correct. The base case A(0) is correct (trivially), so all following recursive calls are also correct (principle of induction).

Running time

For each block, there are at most 6 unique orientations it can take, so our list that we are running our algorithm on has at most 6n items for n blocks.
To sort this list of 6n items is O(nlogn) using Collections.sort.
To fill in each entry of the DP table, at most 6n other table values must be accessed. Since there are n entries to fill, the total time to fill the DP table is 6n*n, which is $O(n^2)$.
To find the max entry in the table, at most 6n table values must be accessed, which is O(n).

So the total runtime complexity is $O(n^2)$.

Interesting Design Decision

We were thinking at first to have 2 dimensions in our DP table, where we find an optimal solution given a range of blocks from i to j, and assume that j is the block at the top. For the recursive case, we were checking A(i, j-1), A(i, j-2), …, A(i, i) and getting the max if block j can fit on top in the same way as our final algorithm. It was then pointed out to us by Professor Chen that we never used the first term in this function so there was no point in putting it in the function. So we changed to have a 1 dimensional DP table using the same condition. This greatly improved the running time of our algorithm since we could eliminate a bunch of useless operations.

Overview

We use a `BlockDimensions` class to represent a block that has a length, width, and height. The class contains a method to compare two blocks (for ordering in `Collections.sort`), a method to see if one block fits on top of another, and a method to print the dimensions of the block in string representation.

The `BlockStacking` class is where the actual algorithm is run. It has a method `getBlockArray()` which returns an array of all possible orientations of the blocks given in the file, and then sorts these in descending length order. This is called at the beginning of the main method before the algorithm as it creates the list the algorithm is run on.

The actual algorithm is run in the rest of the main method of the `BlockStacking` class.

We use arrays to store our DP table and table pointers for finding the max height and reconstructing the optimal solution.

We use an ArrayList of BlockDimensions to store the actual optimal solution once we have reconstructed it and write them to the output file in order using the toString method in BlockDimensions.

Testing

We ran our algorithm on the test examples given and compared our output to the given solution. For 3 blocks we got:
3
6 8 2
2 6 8
1 4 10
This was the same as the solution given to us.

For 10 blocks we got:
12
915 793 335
862 782 530
782 530 862
649 492 386
567 429 368
540 426 926
492 386 649
429 368 567
421 362 27
211 172 736
123 135 67
67 123 135
This was slightly different to the solution given, but was only different in the ordering of the lengths and widths – the heights were the same – and our solution was still valid. This means our algorithm simply returned a different optimal solution.

We also tested our output against the 100 block example, and the solution was also valid in the same way as above. Actual output is omitted for brevity.

Acknowledgements

We would like to thank Professor Chen for helping us refine our algorithm in office hours today, and Marcel for giving us pointers on how to finish up the project.