Samuel Sicklick

Algorithm and Key Insights

This implementation of *StockYourBookshelf* employs dynamic programming in an almost parallel way to *SubsetSum*, in that a data structure is constructed to contain the highest possible amount that can be spent for each theoretical maximum to spend until the *budget* (@param *M* in the comments to *maxAmountThatCanBeSpent*) is reached. Through object oriented programming, the algorithm checks at each introduction of a Sefer *class* the most expensive possible *type* which can be added given the constraints of incrementing amounts of money to be spent and that there remains in the *Shelf* one *type* from all prior *classes*. By seeing which *types* of which previous *classes* were bought when the amount which could be spent was *M* - *t*, where *t* is the value of a particular *type*, it can be determined if *t* can be bought as the *type* for the *class* at hand.

Optimal Substructure with Overlapping Subproblems

The overlapping subproblems in *StockYourBookshelf* are the *types* of which previous *classes* were bought when the amount which could be spent was *M* - *t*, where *t* is the value of a particular *type* of the *class* at hand. This provides an optimal substructure since the addition of *t* will now maximize the money spent from *M* at this *class* while maintaining that there are still some *type* of each prior *class* being purchased.

Recurrence

Let *max*(*i*) be the highest value T_j of C_i which is viable to be purchased.

$$OPT(C_i,\ M) = \left\{ \begin{array}{ll} 0 & if\ OPT(i-1, M - max(C_i) = 0 \\ max(C_i) & if\ i = 1 \\ max\ (C_i) + OPT(C_i - 1, M - max(C_i)) & if\ i > 1\ and\ OPT(i-1, M - max(C_i))\ != 0 \end{array} \right\}$$

Performance

There are CM cells to fill and filling each cell takes $\sum_{i=1}^{M} C_i.size()$ time, as each member of C_i must be examined to determine the highest value T_j which can be purchased at stage $C_i M$. Let $t = \sum_{i=1}^{M} C_i.size()$, as it represents the sum of the T_j values over *i* instances of *C*. Determining *t* only requires $O(1)$ lookups for each of the values in the summation. Thus, the runtime of *maxAmountThatCanBeSpent* is $O(CMt)$. Assuming there is a possible solution, the optimal *Shelf* will be stored with T_j values in the order by which they were observed by *maxAmountThatCanBeSpent*. They are sorted via *Collections.sort*, which implements *Mergesort* and runs in $O(nlogn)$ at worst, to be immediately returned. Thus, *solution* runs in $O(nlogn)$ time, where *n* is equivalent to *C* since the *Shelf* contains one *type* of each *class*.