PS7

July 9, 2025

Problem 7-1

The implementation of method best_seam(self) appears in resizeable_image.py. The function evaluates the best seam using a bottom up dynamical programming procedure. This is equivalent to a simple iteration of a directed acyclic graph (DAG) in the order of a topological sort. The code can be tested with test_resizable_image.py. The test file has been modified to run on Python 3.

Problem 7-2

(a)

If I had 20$ available to purchase stocks in 1991, I would buy one stock of Pear, Inc. which give a profit of $45\$ + 5\$ = 50\$$ in 2011.

(b)

If I had 30$ available to purchase stocks in 1991, I would buy two stocks of Pear, Inc. which would give a profit of $2 \cdot 45\$ = 90\$$.

(c)

If I had 120$ available to purchase stocks in 1991, I would buy 10 stocks of Dale, Inc. which would give a profit of $10 \cdot 39\$ = 390\$$.

(d)

The stock purchasing problem has four inputs: *total* - initial money, *count* - number of companies with stock available, *start* - array containing the prices of each stock at 1991, and *end* - array containing the prices of each stock in 2011. Prices are assumed to be in $\mathbb{N}$.

The knapsack problem takes inputs: *items* - the number of different items, *size* - the item sizes ($\in \mathbb{N}$), *value* - the item value, and the *capacity* - total size of the knapsack.

There is aparent correspondence between the inputs of the two problems.

total corresponds to *capacity*.

(e)

count corresponds to *items*.

(f)

start corresponds to *size*.

(g)

end corresponds to *value*.

(h)

The algorithm for the knapsack problem cannot be directly applied to the stock purchasing problem. The valid reasons why the the knapsack alogrithm can't be directly applied are reasons 3 and 5 (reason 2 is only a difference):

3. In the stock purchasing problem, the money left over from the purchases is kept as cash, which contributes to the utlimate profit. The knapsack problem has no equivalent concept.

5. In the stock purchasing problem, you can buy more than one share in each stock.

(i)

In STOCK-TABLE-A from Fig. 1 there is a nested loop iterating over the cash (0 to *total*) and stock (from 1 to *count*). Therefore the running time is $O(total \cdot count)$.

(j)

Similarly to the last question, in STOCK-TABLE-B from Fig. 2 there is a nested loop iterating over the *cash* (0 to *total*) and *stock* (from 1 to *count*). A single iteration of the loop is $O(1)$. Therefore the running time is $O(total \cdot count)$.

(k)

In STOCK-RESULT-A from Fig. 3 there are two seperate loops, one for loop over the *stock* from 1 to *count* and a while loop for *stock* which starts at *count* and decreases by 1 in each iteration. A single iteration of the loop is $O(1)$. Therefore the running time is $O(count)$. The algorithm basically, goes over the elements of the purchase table and buys a specified amount of shares if there is enough money left.

(l)

Similarly to the last question. In STOCK-RESULT-B from Fig. 4 there are two seperate loops, one for loop over the *stock* from 1 to *count* and a while loop for *stock* which starts at *count* and decreases by 1 in each iteration. Therefore the running time is $O(count)$.

The algorithm basically, goves over the elements of the purchase table (in a decreasing order of stocks, starting from *count*) and buys one share of a certain stock if there is enough money left.

(m)

Im STOCK-RESULT-C from Fig. 5 there are two seperate loops, one for loop over the *stock* from 1 to *count* with contributes $O(count)$ and a while loop for *stock* which iterates from *count* to 0 and decreases by 1 only when there is not enough case to buy more stocks. In the worst case where the price of the first stock (iterated in the while loop) is 1\$ and this takes $O(total)$ time. Note, that if no $purchase[cash, stock]$ is $False$ for all *stock* this leads to an infinite loop.

The algorithm, goes over the elements of the purchase table if specified in the purchase table (True), it buys a much of a certain stock as it can and goes to the "previous" indexed stock type in the predetermined order.

(n)

The recurrence relation computed by the STOCK-TABLE-A function is:

$$profit[c, s] = \max profit[c, s-1], profit[c - start[s], s] + end[s]$$

The method is useful for the stock purchasing problem, since you can buy multiple number of shares of a certain stock type.

(o)

The recurrence relation computed by the STOCK-TABLE-B function is:

$$profit[c, s] = \max profit[c, s-1], profit[c - start[s], s-1] + end[s]$$

The method is useful for the knapsack problem, since once you made a choice regarding a certain item you go on to the next item. The difference between the two recurrenc relations (in this question and the previous one) is the s, $s-1$ and the second agument of $profit$ (on the right).

(p)

Methods STOCK-TABLE-B and STOCK-RESULT-B when combined, let you compute the answer to the knapsack problem.

(q)

Methods STOCK-TABLE-A and STOCK-RESULT-C when combined, let you compute the answer to the stock purchase problem. STOCK-TABLE-A produces a purchase table with values boolean values for a defined state of $[cash, stock]$, which leads to STOCK-RESULT-C "buying" as much shares as it can (for a defined quantity of $cash$) of the specified stock. It then returns the result table, with stores how many stocks to buy for each stock type.

(r)

If I had 30$ in 1991. There are tree leading stock purchase strategies:

1. One of Pear, Inc. and one of Dale, Inc, giving a profit of 87$
2. One of Macroware, Inc. and one of Dale, Inc., giving a profit of 86$
3. Two of Dale, giving a total profit of 84$.

Therefore the first strategy is the best.

(s)

If I had 120$ in 1991. The best strategy is to buy the most profitable shares, Dale, Inc., Macroware, Inc., Pear, Inc. until the allowed limit, this purchase costs 87$. Then I would buy 3 shares of Macroware, Inc., leading to a total income of 297$.

(t)

The limits allows adding subproblems to the recursion relation

$$profit[c, s] = \max \min_{m \in \{limit[s], c/start[s]\}} (profit\left[c, s-1\right], profit\left[c - m \cdot start\left[s\right], s\right] + m \cdot end\left[s\right])$$

m is either bounded by $limit\left[s\right]$ are by the amount of cash c.

We modify STOCK-TABLE-A to consider the possible choices of m, and modify the elements of $purchase$, which instead of a boolean value are now the number of shares to buy. The final part is similar to STOCK-RESULT-C where the stocks are iterated in reverse order and outputing the total amount of money

The pseudo-code taking into acount the limits on the number of shares is given bellow.

STOCKLIMITED($total$, $limit$,$count$, $start$, end):
_create a table $profit$
_create a table $purchase$
_**for** $cash = 0$ to $total$:
__$profit\left[cash, 0\right] = cash$
__$purchase\left[cash, 0\right]$ =False
__**for** $stock = 1$ to $count$:
___$profit\left[cash, stock\right] = profit\left[cash, stock - 1\right]$
___$purchase\left[cash, stock\right] = 0$
___$stock_limit = \min\left(limit\left[s\right], cash/start\left[s\right]\right)$

___**for** $m = 1$ to $stock\_limit$:
____$leftover = cash - m \cdot start\,[stock]$
____$current = m \cdot end\,[stock] + profit\,[leftover, stock]$
____**if** $profit\,[cash, stock] < current$**:**
_____$profit\,[cash, stock] = current$
_____$pruchase\,[cash, stock] = m$
_$cash = total$
_$total_money = 0$
_**for** $stock = count$ to 1:
__$q = purchase\,[cash, stock]$
__$total_money = total_money = q \cdot end\,[s]$
__$cash = cash - q \cdot start\,[s]$
_return $total\_money$

Running time analysis: if there is no limit, the additional for loop is bounded by $total/start\,[s]$ which worst case value is $O\,(total)$. Therefore we add an additional factor $total$ to the original running time $O\left(count \cdot total^2\right)$. This can also be seen by noticing that there are three nested loops, the first and the third iterate $total$ times in the worst case and the second iterates $count$ number of times in the worst case.

Correnctness: The algorithm runs over all possible combinations of $(cash, stock, stock\_limit)$ and stores the the best ones in $purchase$, the final part reverses the original order of iteration over the stocks, insuring that the optimal result is obtained.