Problem Set 7

7.1. Seam Carving

In a recent paper, "Seam Carving for Content-Aware Image Resizing", Shai Avidan and Ariel Shamir describe a novel method of resizing images.
You are given an image, and your task is to calculate the best vertical seam to remove. A 'vertical seam' is a connected path of pixels, one pixel in each row. We call two pixels 'connected' if they are vertically or diagonally adjacent. The 'best' vertical seam is the one that minimizes the total "energy" of pixels in the seam.
Here's the dynamic programming algorithm:

Subproblems: for each pixel (i,j), what is the lower-energy seam that starts at the top row of the image, but ends at (i,j)?

Relation: Let 'dp[i,j]' be the solution to subproblem (i,j). Then
'dp[i,j] = min(dp[i,j-1], dp[i-1, j-1], dp[i+1, j-1]) + energy(i,j)'

Analysis: Solving each subproblem takes O(1) time: there are three smaller subproblems to look up, and one call to energy(), which all take O(1) time. There is one subproblem for each pixel, so the running time is O(A), where A is the number of pixels, i.e, the area of the image.

In 'resizable_image.py', write a function 'best_seam(self)' that returns a list of coordinates corresponding to the cheapest vertical seam to remove. You should implement the dynamic program described above in a bottom-up manner.
Test your code using 'test_resizable_image.py', and submit 'ResizableImage.py'.
You can view your code in action by running 'gui.py'.

See 'best_seam(self)' in 'resizable_image.py' tests complete in <4sec
Also, I included a top-down approach 'remove_best_seam(self)', which runs significantly slower, but produces the same results.

7.2. HG Fargo

You have been given an internship at the extremely profitable and secretive bank HG Fargo. Your immediate supervisor tells you that higher-ups in the bank are interested in learning from the past. In particular, they want to know how much money they 'could' have made if they had invested optimally.

Your supervisor gives you the following data on the prices of select stocks in 1991 and in 2011:

Company	Price in 1991	Price in 2011
Dale Inc	$12	$39
JCN Corp	$10	$13
Macrowave	$18	$47
Pear Inc.	$15	$45

a. If you had $20 available to purchase stocks in 1991, how much of each stock should you buy to maximize profits when you sell everything in 2011? Note that you do not need to invest all of your money.

1 share of pear $20 → $5 (cash) + $45 (pear)

b. If you had $30 available to purchase stocks in 1991, how much of each stock should you have bought?

Reasonable possibilities: 2 Dale | $84, 1 Dale 1 Pear | $87, 2 Pear | $90

2 Pear

c. If you had $120 available to purchase stocks in 1991, how much of each should you have bought?

10 shares of Dale

Dale stock yielded the highest return and 12 | 120, so we don't have to worry about remainders.

Your supervisor asks you to write an algorithm for computing the best way to purchase stocks, given the initial money total, the number count of companies with stock available, an array start containing the prices of each stock in 1991, and an array end containing the prices of each stock in 2011. All prices are assumed to be positive integers.

There is a strong relationship between this problem and the knapsack problem. The knapsack problem takes four inputs: the number of different items, the item sizes size, the item values value, and the size capacity of the knapsack. The goal is to pick a subset of the items that fits inside the knapsack and maximizes the total value.

d. Which input to the knapsack problem corresponds to the input total in the stock purchasing problem?

capacity, since total is the initial 'space' (in money) we are given to fill with items (via investing)

e. .. count ..

items. These are our distinct options for investing. Our integers in our linear combinations

f. .. start ..

size. This is what it "costs" to fill our bag or buy stocks

g. .. end ..

value. This is what we recieve when we cash out/ empty our bag.

h. Unfortunately, the algorithm for the knapsacks problem cannot be directly applied to the stock purchasing problem. For each of the following potential reasons, state whether it's a valid reason not to use the knapsacks algorithm.

1. In the stock purchase problem, there is a time delay between the selection and the reward.

Not valid

2. All of the numbers in the stock purchasing problem are integers. The value array in the knapsacks problem is not.

Not valid.

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

This reason is valid.

4. In the knapsack problem there are some variables representing the sizes of objects. There are no such variables in the stock purchasing problem.

Not valid.

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

This is a valid reason.

6. In the stock purchasing problem, you sell all the items at the end. In the knapsack problem, you don't do anything with the items.

Not valid.

Despite these differences, you decide that the knapsack algorithm is a good starting place for the problem you are trying to solve. So you dig up some pseudocode for the knapsack problem, relabel the variables to suit the stock purchasing problem, and start modifying things. After a long night of work, you end up with a couple feasible solutions. Unfortunately, there is a bit of a hard-drive error the next morning, and the files are mixed up. You have recovered 6 different functions, from various states in your development process. The first is the following:

```
STOCK(total, count, start, end)
return STOCK-RESULT(total, count, end, purchase)
```

This is the function that you ran to get your results. The STOCK-TABLE function generates the table of subproblem solutions. The STOCK-RESULT function uses that to figure out which stocks to purchase, and in what quantities. Unfortunately, you have two copies of the STOCK-TABLE function and three copies of the STOCK-RESULT function. You know that there's a way to take one of each function to get the pseudocode for the original knapsack problem (with the names changed). You also know that there's a way to take one of each function to get the pseudocode for the stock purchases problem. You just don't know which functions do what.

Analyze each of the other five procedures, and select the correct running time. Recall that 'total' and 'count' are positive integers, as are each of the values 'start[stock]' and 'end[stock]'. To make the code simpler, the arrays 'start', 'end', and 'result' are assumed to be indexed starting at 1, while the tables 'profit' and 'purchase' are assumed to be indexed starting at (0,0). You may assume that entries in a table can be accessed and modified in $\Theta(1)$ time.

i. What is the worst-case asymptotic running time of STOCK-TABLE-A in terms of 'count' and 'total'?

```
STOCK-TABLE-A(total, count, start, end)
create a table profit
create a table purchase
for cash=0 to total
    profit[cash, 0] = cash
    purchase[cash, 0] = FALSE
    for stock=1 to count
        profit[cash, stock] = profit[cash, stock-1]
        purchase[cash, stock] = FALSE
        if start[stock] ≤ cash
            leftover = cash - start[stock]
            current = end[stock] + profit[leftover, stock]
            if profit[cash, stock] < current
                profit[cash, stock] = current
                purchase[cash, stock] = TRUE
return purchase
```

Θ(total) Θ(count)

Θ(count · total)

j. What is the worst-case asymptotic running time of STOCK-TABLE-B in term of count and total?

```
STOCK-TABLE-B(total, count, start, end)
create a table pofit
create a table purchase
for cash=0 to total
    profit[cash, 0] = 0
    purchase[cash, 0] = FALSE
    for stock=1 to count
        profit[cash, stock] = profit[cash, stock-1]
        purchase[cash, stock] = FALSE
        if start[stock] ≤ cash
            leftover = cash - start[stock]
            current = end[stock] + profit[leftover, stock-1]
            if profit[cash, stock] < current
                profit[cash, stock] = current
                purchase[cash, stock] = TRUE
return purchase
```

Θ(total)

Θ(count)

Θ(count · total)

K. What is the worst-case asymptotic running of STOCK-RESULT-A in terms of count and total?

```
STOCK-RESULT-A(total, count, start, end, purchase)
create a table result
for stock = 1 to count                    } Θ(count)
    result[stock] = 0
cash = total
stock = count
while stock > 0                           }
    quantity = purchase[cash, stock]      } Θ(count)
    result[stock] = quantity              }
    cash = cash - quantity · start[stock] }
    stock = stock - 1                     }
return result
```

Θ(count)

1. What is the worst-case asymptotic running time of STOCK-RESULT-B in terms of count and total?

```
STOCK-RESULT-B (total, count, start, end, purchase)
create a table result
for stock = 1 to count                    } Θ(count)
    result[stock] = FALSE                 }
cash = total
stock = count
while stock > 0                           }
    if purchase[cash, stock]              } Θ(count)
        result[stock] = TRUE              }
        cash = cash - start[stock]        }
    stock = stock - 1                     }
return result
```

Θ(count)

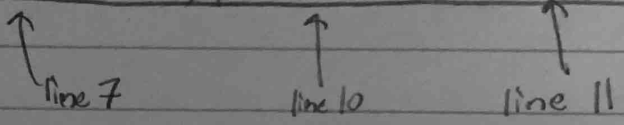
m. What is the worst-case asymptotic running time of STOCK-RESULT-C in terms of count and total?

```
STOCK-RESULT-C(total, count, start, end, purchase)
create a table result
for stock=1 to count                    } Θ(count)
    result[stock] = 0
cash = total
stock = count
while stock>0                           } Θ(total+count)
    if purchase[cash, stock]
        result[stock] = result[stock] + 1
        cash = cash - start[stock]
    else
        stock = stock - 1
return result
```

Θ(total + count)

m. The recurrence relation computed by STOCK-TABLE-A is:

5. $profit[c,s] = max\{profit[c,s-1], profit[c-start[s],s] + end[s]\}$



Q. The recurrence relation computed by STOCK-TABLE-B is:

it chooses the max of:

profit[cash, stock-1]

and end[s] + profit[cash - start[stock], stock-1]

this corresponds to option **Z.**

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

p. Which two methods, when combined, let you compute the answer to the knapsack problem?

STOCK-TABLE-B & STOCK-RESULT-B

doesn't allow you to buy multiple shares "current = end[stock] + profit[leftover, stock-1]"

Uses booleans to identify which stocks were bought (what table B returns)

q. Which two methods, when combined, let you compute the answer to the stock purchases problem?

STOCK-TABLE-A & STOCK-RESULT-C

The stock table A recurrence relation allows us to buy multiple shares of stock in a company.
STOCK-RESULT-C contains booleans, and works on integer results.

With all that sorted out, you submit the code to your supervisor and pat yourself on the back for a job well done. Unfortunately, your supervisor comes back a few days later with a complaint from the higher-ups. They've been playing with your program, and were very upset to discover that when they ask what to do with $1B in 1991, it tells them to buy tens of millions of shares in Dale, Inc. According to them, there weren't that many shares of Dale available to purchase. They want a new feature: the ability to pass in limits on the number of stocks purchasable.

You begin, as always, with a small example:

Company	Price in 1991	Price in 2011	Limit
Dole Inc.	$12	$39	3
JCN Corp	$10	$13	∞
Macrowave	$18	$47	2
Pear, Inc	$15	$45	1

r. If you had $30 available to purchase stocks in 1991, how much of each stock should you have bought, given the limits imposed above?

1 share of Dale & 1 share of Pear $12+15+3 \rightarrow 39+45+3 = 187$

S... $120 ...?

JCN is not very profitable, so first we buy all available shares of the others for $87. The leftovers are used to buy 3 shares of JCN and $3 in cash.

D3J3MZP1C3

4. Give pseudocode for an algorithm STOCK-LIMITED that computes the maximum profit achievable given a starting amount 'total', a number 'count' of companies with stock available, an array of initial prices 'start', an array of final prices 'end', and an array of quantities 'limit'. The value stored at limit[stock] will be equal to ∞ in cases where there is no known limit on the number of stocks. The algorithm need only output the resulting quantity of money, not the purchases necessary to get that quantity. Remember to analyze the runtime of your pseudocode, and provide a brief justification for its correctness. It is sufficient to give the recurrence relation your algorithm implements and talk about why the recurrence relation solves the problem at hand.

STOCK-LIMITED(total, count, start, end, limit) ⊕(count · total²)
create a table maximum, maximum[0] = 0
for stock = 1 to count
    maximum[stock] = min(limit[stock], total / (start[stock]) // so all have finite maxes
create a table profit
for cash = 0 to total // go through funds ⊕(total)
    profit[cash, 0, 0] = cash
    for stock = 1 to count // go through stocks ⊗(count)
        for numBought = 0 to maximum[stock] // go through available stocks ⊗(stock)
            profit[cash, stock, numBought] = profit[cash, stock-1, maximum[stock-1]]
            if numBought > 0 && start[stock] ≤ cash // try buying another
                current = end[stock] + profit[cash - start[stock], stock, numBought - 1]
                profit[cash, stock, numBought] = max{profit[cash, stock, numBought], current}

This works because we simply add a 3rd subproblem constraint to the stock picking problem.
The new recurrence relation is:

$$profit[cash, stock, limit] := \begin{cases} cash & \text{if } stock = 0 \\ profit[cash, stock-1, max[s-1]] & \text{if } limit = 0 \text{ or } start[s] > c \\ \max\{profit[cash - start[stock], stock, limit-1] + end[stock], & \text{else} \\ \quad profit[cash, stock-1, max[s-1]] \end{cases}$$