Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — November 22, 2011
Professors Erik Demaine and Srini Devadas — Problem Set 7

Problem Set 7

Both theory and programming questions are due on **Tuesday, December 6** at **11:59PM**. Please download the .zip archive for this problem set. Instructions for submitting your answers will be posted to the course website by Tuesday, November 29.

We will provide the solutions to the problem set 10 hours after the problem set is due. You will have to read the solutions, and write a brief **grading explanation** to help your grader understand your write-up. You will need to submit the grading guide by **Thursday, December 8, 11:59PM**. Your grade will be based on both your solutions and the grading explanation.

Problem 7-1. [30 points] **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 welcome to read the paper, but we recommend starting with the YouTube video:

`http://www.youtube.com/watch?v=vIFCV2spKtg`

Both are linked from the Problem Sets page on the class website. After you've watched the video, the terminology in the rest of this problem will make sense.

If you were paying attention around time 1:50 of the video, then you can probably guess what you're going to have to do. 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.

The video didn't spend much time on dynamic programming, so here's the 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 $\Theta(A)$, where A is the number of pixels, i.e., the area of the image.

Download `ps7_code.zip` and unpack it. To solve this problem set, you will need the Python Imaging Library (PIL), which you should have installed for Problem Set 4. If you wish to view your results, you will additionally need the Tkinter library.

In `resizeable_image.py`, write a function `best_seam(self)` that returns a list of coordinates corresponding to the cheapest vertical seam to remove, e.g., $[(5, 0), (5, 1), (4, 2), (5, 3), (6, 4)]$. You should implement the dynamic program described above in a bottom-up manner.

The class `ResizeableImage` inherits from `ImageMatrix`. You should use the following components of `ImageMatrix` in your dynamic program:

- `self.energy(i,j)` returns the energy of a pixel. This takes $O(1)$ time, but the constant factor is sizeable. If you call it more than once, you might want to cache the results.
- `self.width` and `self.height` are the width and height of the image, respectively.

Test your code using `test_resizable_image.py`, and submit `ResizeableImage.py` to the class website. You can also view your code in action by running `gui.py`. Included with the problem set are two differently sized versions of the same sunset image. If you remove enough seams from the sunset image, it should center the sun.

Also, please try out your own pictures (most file formats should work), and send us any interesting before/after shots.

Solution: Code is available on the course website.

Problem 7-2. [70 points] **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 very 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[1] of select stocks in 1991 and in 2011:

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

As a first step, you decide to examine what the optimal decision is for a couple of small examples:

(a) [5 points] If you had \$20 available to purchase stocks in 1991, how much of each stock should you have bought to maximize profits when you sell everything in 2011? Note that you do not need to invest all of your money — if it is more profitable to keep some as cash, you do not need to invest it.

Solution: There are six distinct possible purchasing plans. We can buy no stocks at all, ending with $\$20$ in 2011. There are four different ways we can buy one share: one for each company. For Dale, the money we would end up with is $\$39 + \$20 - \$12 = \$47$. For JCN, the money we would end up with is $\$13 + \$20 - \$10 = \$23$. For Macroware, the money we would end up with is $\$47 + \$20 - \$18 = \$49$. For Pear, the money we would end up with is $\$45 + \$20 - \$15 = \$50$. The best of these is clearly Pear. The only other option would be to buy two shares in JCN, leaving us with a total of $2 \cdot \$13 = \$26$. Hence, the best option is to buy a single share in Pear.

(b) [5 points] If you had \$30 available to purchase stocks in 1991, how much of each stock should you have bought?

Solution: To make this simpler, note that it is never cost-effective to purchase Macroware. Suppose that we choose to purchase k shares in Macroware. Now consider how our money would be affected if instead we choose to purchase k shares in Pear with the same amount of money. The amount of money that we make by selling those shares is $k \cdot \$45$, as opposed to $k \cdot \$47$, but when purchasing Pear instead of Macroware, we save $k \cdot \$3$ on the initial purchases. That money can be kept as cash, so that we end up with $k \cdot (\$45 + \$3) = k \cdot \$48 > k \cdot \$47$ in 2011. Hence, it's never cost-effective to purchase Macroware.

[1]Note that for the purposes of this problem, you should ignore some of the intricacies of the real stock market. The only income you can make is from purchasing stocks in 1991, then selling those same stocks at market value in 2011.

A similar argument can be made about JCN. Because JCN is cheaper than all of the other stocks, we cannot always replace it with a cheaper stock. But purchasing two shares in JCN is less cost-effective than purchasing a share in Pear, so we will never want to purchase more than one share in JCN. Furthermore, if we purchase one share in JCN and leave $\geq \$2$ as cash, we could have made more money by instead purchasing a share in Dale.

This limits the set of possibilities. If we purchase one share, the best option is still Pear. Consider the possible ways to purchase two shares. We need not consider any combinations involving Macroware. Furthermore, because of our initial prices, pairing a single share in JCN with a single share in any other company will always leave us with $\geq \$2$ in cash, so it's not a good idea to purchase JCN. As a result, we need only consider three combinations: two shares in Dale (yielding $\$39 + \$39 + \$30 - \$12 - \$12 = \$84$); one share in Dale and one in Pear (yielding $\$39 + \$45 + \$30 - \$12 - \$15 = \$87$); or two shares in Pear (yielding $\$45 + \$45 + \$30 - \$15 - \$15 = \$90$). So the best thing to do is to buy two shares in Pear.

(c) [5 points] If you had \$120 available to purchase stocks in 1991, how much of each stock should you have bought?

Solution: Dale is the single most profitable company of the four, with an increase of 225% on the initial investment. So in cases where the amount of money is evenly divisible by 12, the best plan is to buy as many shares in Dale as possible. Hence, the best thing to do is to buy 10 shares in Dale.

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 $items$, the item sizes $size$ (all of which are integers), the item values $value$ (which may not be integers), 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) [1 point] Which input to the knapsack problem corresponds to the input $total$ in the stock purchasing problem?

1. $items$ 2. $size$ 3. $value$ 4. $capacity$

Solution: 4: $capacity$. We can think of the stock purchasing problem as stuffing as many shares into a "bag" with capacity equal to our initial money supply.

(e) [1 point] Which input to the knapsack problem corresponds to the input $count$ in the stock purchasing problem?

1. $items$ 2. $size$ 3. $value$ 4. $capacity$

Solution: 1: *items*. The variable *count* gives the number of companies available to purchase shares of. The variable *items* tells us how many items are available to stuff in our bag. These items are roughly equivalent.

(f) [1 point] Which input to the knapsack problem corresponds to the input *start* in the stock purchasing problem?

1. *items* 2. *size* 3. *value* 4. *capacity*

Solution: 2: *size*. The size of an item in the knapsack problem affects how many can be placed in the knapsack. The same is true of the initial price, which affects the prices we can afford in 1991.

(g) [1 point] Which input to the knapsack problem corresponds to the input *end* in the stock purchasing problem?

1. *items* 2. *size* 3. *value* 4. *capacity*

Solution: 3: *value*. The sum of these values is what we try to optimize for both problems.

(h) [6 points] Unfortunately, the algorithm for the knapsack 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 knapsack algorithm. (In other words, if the difference mentioned were the only difference between the problems, would you still be able to use the knapsack algorithm to solve the stock purchasing problem?)

1. In the stock purchasing problem, there is a time delay between the selection and the reward.
2. All of the numbers in the stock purchasing problem are integers. The *value* array in the knapsack problem is not.
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.
4. In the knapsack problem, there are some variables representing sizes of objects. There are no such variables in the stock purchasing problem.
5. In the stock purchasing problem, you can buy more than one share in each stock.
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.

Solution: The only valid reasons are 3 and 5. Reason 2 is a difference, but if the only difference is integer vs. real, then the knapsack algorithm (which handles reals) will also be able to take integers.

Despite these differences, you decide that the knapsack algorithm is a good starting point 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 then start modifying things. After a long night of work, you end up with a couple of feasible solutions. Unfortunately, there is a bit of a hard-drive error the next morning, and the files are all mixed up. You have recovered six different functions, from various states in your development process. The first function is the following:

```
STOCK($total$, $count$, $start$, $end$)
1  $purchase$ = STOCK-TABLE($total$, $count$, $start$, $end$)
2  return STOCK-RESULT($total$, $count$, $start$, $end$, $purchase$)
```

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) [1 point] What is the worst-case asymptotic running time of STOCK-TABLE-A (from Figure 1) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

Solution: 10: $\Theta(count \cdot total)$. The outer for-loop runs $total$ times; the inner for-loop runs $count$ times. All operations in the inner for-loop are $\Theta(1)$ in runtime, so the total time is proportional to the number of iterations.

(j) [1 point] What is the worst-case asymptotic running time of STOCK-TABLE-B (from Figure 2) 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 $start[stock] \leq cash$
            $leftover = cash - start[stock]$
            $current = end[stock] + profit[leftover, stock]$
            if $profit[cash, stock] < current$
                $profit[cash, stock] = current$
                $purchase[cash, stock] =$ TRUE
return $purchase$
```

Figure 1: The pseudocode for STOCK-TABLE-A.

```
STOCK-TABLE-B($total, count, start, end$)
create a table $profit$
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 $start[stock] \leq cash$
            $leftover = cash - start[stock]$
            $current = end[stock] + profit[leftover, stock - 1]$
            if $profit[cash, stock] < current$
                $profit[cash, stock] = current$
                $purchase[cash, stock] =$ TRUE
return $purchase$
```

Figure 2: The pseudocode for STOCK-TABLE-B.

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

Solution: 10: $\Theta(count \cdot total)$. Again, the outer for-loop runs $total$ times, and the inner for-loop runs $count$ times. One iteration of the inner for-loop is $\Theta(1)$ in runtime.

(k) [1 point] What is the worst-case asymptotic running time of STOCK-RESULT-A (from Figure 3) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

Solution: 1: $\Theta(count)$. We perform two loops. The first is a for-loop with $count$ iterations, and each iteration is a $\Theta(1)$ assignment. The second is a while loop. It stops when the variable $stock$ becomes zero. The variable $stock$ starts out equal to $count$, and decreases by 1 at each iteration, so the total number of iterations is $\Theta(count)$.

(l) [1 point] What is the worst-case asymptotic running time of STOCK-RESULT-B (from Figure 4) in terms of $count$ and $total$?

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

Solution: 1: $\Theta(count)$. Again, the first loop has $count$ iterations, and the decreasing variable $stock$ in the second loop ensures that it runs $\Theta(count)$ times.

(m) [1 point] What is the worst-case asymptotic running time of STOCK-RESULT-C (from Figure 5) in terms of $count$ and $total$?

```
STOCK-RESULT-A($total, count, start, end, purchase$)
create a table $result$
for $stock = 1$ to $count$
    $result[stock] = 0$

$cash = total$
$stock = count$
while $stock > 0$
    $quantity = purchase[cash, stock]$
    $result[stock] = quantity$
    $cash = cash - quantity \cdot start[stock]$
    $stock = stock - 1$

return $result$
```

Figure 3: The pseudocode for STOCK-RESULT-A.

```
STOCK-RESULT-B($total, count, start, end, purchase$)
create a table $result$
for $stock = 1$ to $count$
    $result[stock] =$ FALSE

$cash = total$
$stock = count$
while $stock > 0$
    if $purchase[cash, stock]$
        $result[stock] =$ TRUE
        $cash = cash - start[stock]$
    $stock = stock - 1$

return $result$
```

Figure 4: The pseudocode for STOCK-RESULT-B.

```
STOCK-RESULT-C($total$, $count$, $start$, $end$, $purchase$)
create a table $result$
for $stock = 1$ to $count$
    $result[stock] = 0$

$cash = total$
$stock = count$
while $stock > 0$
    if $purchase[cash, stock]$
        $result[stock] = result[stock] + 1$
        $cash = cash - start[stock]$
    else
        $stock = stock - 1$

return $result$
```

Figure 5: The pseudocode for STOCK-RESULT-C.

1. $\Theta(count)$
2. $\Theta(count^2)$
3. $\Theta(count^3)$
4. $\Theta(total)$
5. $\Theta(total^2)$
6. $\Theta(total^3)$
7. $\Theta(count + total)$
8. $\Theta(count^2 + total)$
9. $\Theta(count + total^2)$
10. $\Theta(count \cdot total)$
11. $\Theta(count^2 \cdot total)$
12. $\Theta(count \cdot total^2)$

Solution: 7: $\Theta(count + total)$. Again, the first loop takes time $\Theta(count)$. The second loop, however, is more complicated. With every iteration, either $cash$ or $stock$ decreases by at least 1. When $stock$ reaches 0, the loop will stop. When $cash$ goes below 0, $purchase[cash, stock]$ can no longer be true (because it won't exist), and so $stock$ will start decreasing instead. As a result, the total number of iterations is bounded by $count + total$. Each iteration takes $\Theta(1)$ time, for a total of $\Theta(count + total)$ time.

(n) [2 points] The recurrence relation computed by the STOCK-TABLE-A function is:

1. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s-1]\}$
2. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s-1] + end[s]\}$
3. $profit[c, s] = \max_q\{profit[c - q \cdot start[s], s-1] + q \cdot end[s]\}$
4. $profit[c, s] = \max\{profit[c, s-1], profit[c - start[s], s]\}$

5. $\mathit{profit}[c, s] = \max\{\mathit{profit}[c, s-1], \mathit{profit}[c - \mathit{start}[s], s] + \mathit{end}[s]\}$
6. $\mathit{profit}[c, s] = \max_q\{\mathit{profit}[c - q \cdot \mathit{start}[s], s] + q \cdot \mathit{end}[s]\}$

Solution: 5. The algorithm computes the maximum of two values:

$$\mathit{profit}[\mathit{cash}, \mathit{stock} - 1] \quad \text{and} \quad \mathit{end}[\mathit{stock}] + \mathit{profit}[\mathit{cash} - \mathit{start}[\mathit{stock}], \mathit{stock}]$$

This is exactly what the fifth recurrence relation above computes.

(o) [2 points] The recurrence relation computed by the STOCK-TABLE-B function is:

1. $\mathit{profit}[c, s] = \max\{\mathit{profit}[c, s-1], \mathit{profit}[c - \mathit{start}[s], s-1]\}$
2. $\mathit{profit}[c, s] = \max\{\mathit{profit}[c, s-1], \mathit{profit}[c - \mathit{start}[s], s-1] + \mathit{end}[s]\}$
3. $\mathit{profit}[c, s] = \max_q\{\mathit{profit}[c - q \cdot \mathit{start}[s], s-1] + q \cdot \mathit{end}[s]\}$
4. $\mathit{profit}[c, s] = \max\{\mathit{profit}[c, s-1], \mathit{profit}[c - \mathit{start}[s], s]\}$
5. $\mathit{profit}[c, s] = \max\{\mathit{profit}[c, s-1], \mathit{profit}[c - \mathit{start}[s], s] + \mathit{end}[s]\}$
6. $\mathit{profit}[c, s] = \max_q\{\mathit{profit}[c - q \cdot \mathit{start}[s], s] + q \cdot \mathit{end}[s]\}$

Solution: 2. The algorithm computes the maximum of two values:

$$\mathit{profit}[\mathit{cash}, \mathit{stock} - 1] \quad \text{and} \quad \mathit{end}[\mathit{stock}] + \mathit{profit}[\mathit{cash} - \mathit{start}[\mathit{stock}], \mathit{stock} - 1]$$

This is exactly what the second recurrence relation above computes.

With this information, you should be able to figure out whether STOCK-TABLE-A or STOCK-TABLE-B is useful for the knapsack problem, and similarly for the stock purchasing problem. From there, you can figure out which of STOCK-RESULT-A, STOCK-RESULT-B, and STOCK-RESULT-C is best for piecing together the optimal distribution of stocks and/or items.

(p) [3 points] Which two methods, when combined, let you compute the answer to the knapsack problem?

1. STOCK-TABLE-A and STOCK-RESULT-A
2. STOCK-TABLE-A and STOCK-RESULT-B
3. STOCK-TABLE-A and STOCK-RESULT-C
4. STOCK-TABLE-B and STOCK-RESULT-A
5. STOCK-TABLE-B and STOCK-RESULT-B
6. STOCK-TABLE-B and STOCK-RESULT-C

Solution: 5. From above, we know that the the recurrence relation for STOCK-TABLE-B is the same as the one we use for the knapsack problem, decreasing the number of items by 1 after we choose to use a single item (thereby not letting us use an item twice, as we would need to do in the stock purchasing problem). The table produced by STOCK-TABLE-B contains booleans, so it is not suitable for being paired with STOCK-RESULT-A (which treats the table as containing integers). Furthermore, the knapsack problem should produce booleans indicating whether each item is included, so the boolean result values in STOCK-RESULT-B are just what we want.

(q) [3 points] Which two methods, when combined, let you compute the answer to the stock purchases problem?

1. STOCK-TABLE-A and STOCK-RESULT-A
2. STOCK-TABLE-A and STOCK-RESULT-B
3. STOCK-TABLE-A and STOCK-RESULT-C
4. STOCK-TABLE-B and STOCK-RESULT-A
5. STOCK-TABLE-B and STOCK-RESULT-B
6. STOCK-TABLE-B and STOCK-RESULT-C

Solution: 3. From above, we know that the the recurrence relation for STOCK-TABLE-A is the same as the one we use for the stock purchasing problem, allowing us to purchase more than one share in a company. The table produced by STOCK-TABLE-A contains booleans, so it is not suitable for being paired with STOCK-RESULT-A. However, we do want the values in the table *result* to be integers rather than booleans, so we don't want STOCK-RESULT-B. That leaves us with STOCK-RESULT-C.

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 $1,000,000,000 in the year 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 purchaseable.

You choose to begin, as always, with a small example:

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

(r) [5 points] If you had $30 available to purchase stocks in 1991, how much of each stock should you have bought, given the limits imposed above?

Solution: Unfortunately, we can no longer purchase the two shares in Pear that we did for the unlimited version of this problem. In fact, some of the conclusions we drew about the cost-effectiveness of JCN and Macroware are no longer valid, due to the limits. Note, however, that if we purchase any shares in Macroware and no shares in Pear, then we are not getting the most out of our money — once again, we can replace a single share in Macroware with a single share in Pear. So we still don't need to consider any combinations involving Macroware.

What about combinations involving JCN? Well, if we purchase one share in JCN, then the rest of the money available is $\$20$. We know that even without limits, the most we can make with $\$20$ is $\$50$, so we have an upper bound on the profitability of this venture: if we purchase a share in JCN, the most we can make is $\$63$. This is easily surpassed by the $\$87$ we make by purchasing one share in Pear and one share in Dale, which turns out to be the maximum.

(s) [5 points] If you had \$120 available to purchase stocks in 1991, how much of each stock should you have bought, given the limits imposed above?

Solution: JCN doesn't get us very much profit, so in this case, we want to buy as much of Dale, Macroware, and Pear as we can. That brings us to $\$87$. With the remaining $\$33$, we purchase 3 shares of JCN.

(t) [20 points] Give pseudocode for an algorithm STOCKLIMITED 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 that your algorithm implements, and talk about why the recurrence relation solves the problem at hand.

Solution: There are a couple of possible answers here. The first solution involves changing the set of choices that we make, but not changing the subproblems at all. We based our choices in the stock-purchasing problems around our choices in the knapsack problem: either purchase a share in this stock, or don't purchase a share in this stock. But when we're allowed to purchase more than one share in a company, we could consider this to be an expansion of the set of choices. In the knapsack problem, we had the choice between a quantity of 0 and a quantity of 1. In the stock purchasing problem, the set of values that our quantity can range over is much larger. So we can rewrite our recurrence relation (with full detail of the base cases) as follows:

$$p[c,s] = \begin{cases} c & \text{if } s = 0 \\ \max\limits_{0 \le q \le \min\{limit[s], c/start[s]\}} p[c - q \cdot start[s], s-1] + q \cdot end[s] & \text{otherwise} \end{cases}$$

The valid choices for our quantity q are only those choices that lie within the limits set by the problem, and those values that don't cost more money than we have. This yields the bounds $0 \le q \le \min\{limit[s], c/start[s]\}$. The rest of the equation is straightforward: if we choose to purchase q of the stock s, then we must pay $q \cdot start[s]$, thereby reducing the amount of cash that we have. As a reward, we get $q \cdot end[s]$ added to our profit. When we use the correct order to evaluate this in a bottom-up fashion, we get the following pseudocode:

```
STOCKLIMITED($total, count, start, end, limit$)
create a table $profit$
for $cash = 0$ to $total$
    $profit[cash, 0] = cash$
    for $stock = 1$ to $count$
        $profit[cash, stock] = profit[cash, stock - 1]$
        $maximum = \min\left\{limit[stock], \frac{cash}{start[stock]}\right\}$
        for $quantity = 1$ to $maximum$
            $leftover = cash - start[stock] \cdot quantity$
            $current = end[stock] \cdot quantity + profit[leftover, stock - 1]$
            if $profit[cash, stock] < current$
                $profit[cash, stock] = current$

return $profit[total, count]$
```

This is probably the simplest solution, but there are several others. One option is to add more subproblems: add a third argument to the subproblem indicating the new lowered limit for the current stock. When we choose to purchase a share in the stock, the recursive subproblem we examine has a limit that is smaller by 1. Let $maximum[s]$ be defined as follows:

$$maximum[s] = \begin{cases} 0 & \text{if } s = 0 \\ \min\left\{limit[s], \frac{total}{start[s]}\right\} & \text{otherwise} \end{cases}$$

Then the recurrence relation will be:

$$p[c, s, \ell] = \begin{cases} c & \text{if } s = 0 \\ p[c, s-1, maximum[s-1] & \text{if } \ell = 0 \\ p[c, s-1, maximum[s-1] & \text{if } start[s] > c \\ \max\left\{ \begin{array}{c} p[c - start[s], s, \ell - 1] + end[s], \\ p[c, s-1, maximum[s-1]] \end{array} \right\} & \text{otherwise} \end{cases}$$

When evaluated in the appropriate order, this should give us something like the following pseudocode:

```
StockLimited($total$, $count$, $start$, $end$, $limit$)
create a table $maximum$
$maximum[0] = 0$
for $stock = 1$ to $count$
    $maximum[stock] = \min\left\{limit[stock], \frac{total}{start[stock]}\right\}$

create a table $profit$
for $cash = 0$ to $total$
    $profit[cash, 0, 0] = cash$
    for $stock = 1$ to $count$
        for $\ell = 0$ to $maximum[stock]$
            $profit[cash, stock, \ell] = profit[cash, stock - 1, maximum[stock - 1]]$

            if $\ell > 0$ and $start[stock] \leq cash$
                $current = end[stock] + profit[cash - start[stock], stock, \ell - 1]$
                $profit[cash, stock, \ell] = \max\{profit[cash, stock, \ell], current\}$

return $profit[total, count, maximum[count]]$
```

There is also yet another way to do this, which is similar to the above. With limits, the problem becomes even closer to the knapsack problem. We just need to create $limit[stock]$ different items associated with the stock $stock$. To avoid problems with infinity, we do the same trick as above. Then we need only use a modified version of the knapsack algorithm to compute the desired result:

```
StockLimited($total$, $count$, $start$, $end$, $limit$)
create a list $all\text{-}stocks$
for $stock = 1$ to $count$
    for $quantity = 1$ to $\min\left\{limit[stock], \frac{total}{start[stock]}\right\}$
        append $stock$ to $all\text{-}stocks$

create a table $profit$
for $cash = 0$ to $total$
    $profit[cash, 0] = cash$
    for $share = 1$ to length of $all\text{-}stocks$
        $stock = all\text{-}stocks[share]$
        $profit[cash, share] = profit[cash, share - 1]$
        if $start[stock] \leq cash$
            $current = end[stock] + profit[cash - start[stock], share - 1]$
            $profit[cash, share] = \max\{profit[cash, share], current\}$

return $profit[total$, length of $all\text{-}stocks]$
```

All of these solutions are $\Theta(count \cdot total^2)$ in the worst case. There are ways to speed this up, but the details can get quite complicated, and have little to do with the properties of dynamic programming. As a result, we will not give the algorithm here.