Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — October 11, 2007
Professors Ron Rivest and Srini Devadas — Handout 9

Problem Set 3

This problem set is due **October 23** at **11:59PM**.
Solutions should be turned in through the course website in PS or PDF form using LaTeX or scanned handwritten solutions, or they may be handwritten and turned in to a member of the 6.006 course staff on or before the due date. Hand-drawn diagrams may also be referenced in your LaTeX writeup and turned in at the next day's recitation.
A template for writing up solutions in LaTeX is available on the course website.

Exercises are for extra practice and should not be turned in.
Exercises:

- Consider the amount of memory used in a dynamic programming approach to a problem. Show, by means if illustration, that even though you do need to solve all subproblems and save their solutions for later use, you might not need to save all of the these solutions *simultaneously*; the memory actually used might be less than the number of subproblems.

- Is there any significant difference between the following two approaches to dynamic programming, in terms of their efficiency?

  1. Solve all subproblems, in order of size, from smallest to largest.
  2. Solve each problem and subproblem recursively, starting with the largest subproblem, but saving (via memoization) previous solutions to a subproblem to avoid having to solve it more than once.

  Explain.

- Exercise 15.3-5 from CLRS.

---

1. Image Compression

   In the image compression example in class, we divided up the image into $k \times k$ blocks and found the "best" partition of the image into t non-overlapping monochrome rectangles, with each rectangle a union of blocks. We used the sum of squares of differences between the actual pixel values and the average pixel value in each region as a metric for deciding what partition was "best". Suppose that instead, our metric was the sum of squares of differences between the actual pixel values and the *median* pixel value in a region.

   (8 points) Explain why dynamic programming may not be a good approach to finding the "best" partition of the image in this case.

2. Tris

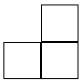
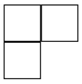
   In this problem, we consider a simplified version of Tetris. (If you are not familiar with this game, ask a TA or look it up on Wikipedia.)

   In this game, each piece is made from 3 squares, so it is either a "bar" (3 in a row) or an "ell". Rotations are not allowed, so there are 2 types of bars and 4 types of ells, accounting for orientation.

   Let us assume that the playing field is three squares wide and arbitrarily tall. The pieces fall from top to bottom, one at a time, and can be shifted left and right but not rotated, until they rest on the bottom of the playing field or a previously placed piece. Once a pieces "touches down", it can't be moved further (e.g. horizontally). Lines are not cleared when they fill up.

   Suppose you know ahead of time what the sequence of n pieces will be (bars or ells and their orientations). How well can the pieces be packed into the bottom?

   Give an algorithm to minimize the total height (height of the highest column) of the stack of pieces, and among stacks with the the minimum height, to minimize the number of "holes" (unfilled squares that are below any filled squares). That is, your algorithm returns the height and the number of holes.

   A:  B: 

   For example, consider these ell pieces A and B. If we get two A pieces, the smallest height is 3, with 1 hole; with 2 B pieces, we can get height 3 with 2 holes; with A followed by B, height 2 with 0 holes; and with B followed by A, height 4 with 1 hole.

   Hint: Consider the "state" of the the puzzle after k pieces have been placed as a triple (a, b, c) where $a + b + c = 3k + h$, representing the height of the stack in columns 1, 2, and 3 respectively, where h is the number of holes.

   (a) **(12 points)** Explain carefully how your algorithm works.

   (b) **(8 points)** Show that your algorithm runs in time polynomial in n.

   (c) **Optional:** Describe an algorithm to solve this problem when rotations are permitted, and explain how the problem is different.

   (d) **Optional:** Does your approach generalize to standard Tetris? What if filled lines in Tetris no longer are cleared?

3. Winning the Stock Market

   Suppose you started with \$1000 on the day you were born. If you had perfect future knowledge of daily stock prices, how much money could you have had on January 1, 2007, and what trades would you have had to make to get there?

   To put some limits on the open-ended nature of this question, we introduce a few restrictions:

- You may only trade the 30 stocks in the current Dow Jones Industrial Average. The daily stock price data is available in the 6.006 locker on Athena in /mit/6.006/stocks/ or from http://courses.csail.mit.edu/6.006/fall07/data/stocks/.
- You may only buy and sell at the daily closing price (use the "adjusted close" column in the data, which takes stock splits into account).
- You may only execute trades, in which you change some combination of cash and stocks for a different combination of cash and stocks of an equal value, at most once every 60 days.
- You are allowed to hold fractional numbers of shares. There is no upper bound on the number of shares you can have.
- You may not short sell stocks.
- At the end, you must hold only cash (no stocks).

You may find the `csv` package useful for reading the comma-separated-values price data files, and the `date` class from the `datetime` package useful for manipulating dates. `csv` documentation is available from http://docs.python.org/lib/module-csv.html; for `date`, see http://docs.python.org/lib/datetime-date.html.

An example usage of csv to read all the stock prices of IBM into a list L:

```
L = list(csv.reader(open("IBM.csv")))
```

An example of finding the number of days' difference between two dates:

```
(datetime.date(2007,10,11) - datetime.date(2007,9,4)).days
```

(a) **(4 points)** Show that the optimal strategy is never to hold more than one kind of stock. *Note:* For this part, it is important that you can hold fractional shares!

(b) **(12 points)** Let d be the total number of days with price data, n be the total number of allowable stocks, and t the minimum number of days between trades. Give an efficient dynamic programming algorithm and describe its running time in terms of d, n, and t.

(c) **(2 points)** How much money would you have had on January 1, 2007?

(d) **(1 point)** Suppose there is a 1% transaction fee associated with each trade. That is, if you owned 100 shares of X, each worth \$10, and the price of a share of Y were \$5, you could sell X and receive \$990, or sell X and buy Y, receiving 198 shares of Y. How much money would you have had on January 1, 2007?

(e) **(1 point)** Suppose there is a 10% transaction fee. How much money would you have had on January 1, 2007?

(f) **(2 points)** What is the average yearly rate of return r on your initial investment for each of the previous 3 parts? $(1 + r)^t$ should be the ratio of your final amount to your initial investment, where t is the time between the day you were born and January 1, 2007 in years (possibly fractional).

The items you should turn in are:

- **(10 points)** Your code, containing a function `most_money(fee)`, which returns the amount of money you would have on January 1, 2007, starting with $1000 on the day you were born, with `fee` = 0, 0.01, or 0.1 for each of the three cases. You can assume that the .csv data files will be in the same directory from which your function will be run.
- A file listing the optimal trades for each of the three cases: no transaction fee, 1%, and 10%.