CS3933 Winter 2007 – Solo Assignment 4
Due Wednesday March 28, 2007, at 5pm.
Sample Solution

Note that, even though desired running times are not specified, faster algorithms are better and thus worth more marks.

1. For a given sequence of n positive integers a_i, consider the following recurrence:

$$M(i,j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i > j \\ \max_{i \le k < j}\{M(i,k) + M(k+1,j) + a_i \cdot a_{k+1} \cdot a_j\} & \text{otherwise} \end{cases}$$

   (a) (5 marks) Design and write a dynamic programming algorithm that will, given n and the a_i values, compute the value of $M(1,n)$ according to this recurrence.

   Algorithm Grouping(integer n, 2-d matrix a)
   - Let M be a 2-dimensional table, $n \times n$, of numbers
   - for i from 1 to n
     - $M[i,i] \leftarrow 1$
   - for i from n down to 1
     - for j from $i+1$ to n
       - $M[i,j] \leftarrow M[i,i] + M[i+1,j] + a_i \cdot a_{i+1} \cdot a_j$
       - for k from $i+1$ to $j-1$
         - if $M[i,j] < M[i,k] + M[k+1,j] + a_i \cdot a_{k+1} \cdot a_j$
           - $M[i,j] \leftarrow M[i,k] + M[k+1,j] + a_i \cdot a_{k+1} \cdot a_j$

   return $M[1,n]$

   Note that the i and j loops could be reversed (with the appropriate relative index changes), or the computation could be done diagonal by diagonal.

   (b) (6 marks) Asymptotically analyze the running time of your algorithm.

   Upper bound:
   The first loop (on i, not nested) iterates exactly n times.
   Of the nested loops, the outer loop (on i) iterates exactly n times, the loop on j iterates at most n times for each i value, and the innermost loop (on k) iterates at most n times for each i and j pair. Therefore the total number of iterations of the innermost loop is at most n^3. Since the prior loop iterates only n times, and the rest of the algorithm contributes at most a constant factor, the algorithm as a whole runs in time $\in O(n^3)$.

Lower bound:
Consider the iteration triples (i, j, k) where $i \leq \frac{n}{3}$, $j > \frac{2n}{3}$, and $\frac{n}{3} \leq k < \frac{2n}{3}$.
The outer loop (on i) has at least $\frac{n}{3} - 1$ iterations where $i \leq \frac{n}{3}$.
The middle loop (on j) has, for each of those values of i, at least $\frac{n}{3} - 1$ iterations where $j > \frac{2n}{3}$, since all of these i and j values will respect the $i < j$ constraint.
The innermost loop (on k) has, for each of those i and j pairs, at least $\frac{n}{3} - 1$ iterations where $\frac{n}{3} \leq k < \frac{2n}{3}$, since all of these values will respect the $i \leq k < j$ constraint.
Therefore, the total number of iterations is at least $(\frac{n}{3} - 1)^3$, so the algorithm runs in time $\in \Omega(n^3)$.

Since this algorithm runs in time both $\in O(n^3)$ and $\in \Omega(n^3)$, then it runs in time $\in \Theta(n^3)$.

2. Stock market investing requires many algorithms for analyzing trends and investing patterns. Consider a simplified situation where there are m stocks, and you have information about their prices for n days. You will always have all of your money invested in one stock; all you can do is, for each day, change which stock all of your money is invested in. You want to determine, in hindsight once the stock prices are known, what the best stock buying pattern is.

   Your input is the initial amount of money (your stake S) and a table of stock change percentages $\{a_i[j] \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$, where $a_i[j]$ is the percentage by which stock i changes on day j. So, if you have X dollars invested in stock i at the start of day j, then at the end of day j you would have $X \cdot (1 + \frac{a_i[j]}{100})$ invested in stock i. These $a_i[j]$ percentages can be either positive or negative.

   Since we have the full table of rates for all m stocks for the n days, we can calculate the best possible stock to invest in on each day. However, switching our investment from one stock to another comes with a price, since we must lose 5% of our investment when we sell stock.

   At the end of the n days, we will sell our stock (of whichever stock we are invested in at the end of day n).

   Hint: this last step is effectively an "extra" step with respect to solving the problem. There are thus m different potential final states (before this last step), one for each possible stock that you could be holding at the end of day n.

   (a) (7 marks) Design a dynamic programming algorithm to find the maximum possible money at the end of the n days, given S, n, m, and the table $\{a_i[j]\}$.

```
Algorithm Money(S, n, m, {a_i[j]})
    Let M be a 2-dimensional table, m × (n + 1), of numbers
    for i from 1 to m
        M[i, 0] ← S
    for j from 1 to n
        for i from 1 to m
            M[i, j] ← M[i, j − 1] · (1 + a_i[j]/100)
            for k from 1 to m
                if k ≠ i and M[i, j] < M[k, j − 1] · (0.95) · (1 + a_i[j]/100)
                  M[i, j] ← M[k, j − 1] · (0.95) · (1 + a_i[j]/100)
    max ← M[1, n]
    for i from 2 to m
        if max < M[i, n]
        max ← M[i, n]
    return max · (0.95)
```

(b) (3 marks) Adjust your algorithm from part (a) so that it will output a list of all stock purchases and sales that will produce the maximum money amount calculated in (a).

```
Algorithm Money(S, n, m, {a_i[j]})
    Let M be a 2-dimensional table, m × (n + 1), of numbers
    Let T be a 2-dimensional table, m × n, of numbers
    for i from 1 to m
        M[i, 0] ← S
    for j from 1 to n
        for i from 1 to m
            M[i, j] ← M[i, j − 1] · (1 + a_i[j]/100)
            T[i, j] ← i
            for k from 1 to m
                if k ≠ i and M[i, j] < M[k, j − 1] · (0.95) · (1 + a_i[j]/100)
                  M[i, j] ← M[k, j − 1] · (0.95) · (1 + a_i[j]/100)
                  T[i, j] ← k
    max ← M[1, n]
    last ← 1
    for i from 2 to m
        if max < M[i, n]
        max ← M[i, n]
```

```
        last ← i
    Trace(T, last)
    print "end of day n: sold stock last"
    return max · (0.95)

Algorithm Trace(T, i, j)
    if j > 1
        Trace(T, T[i, j], j − 1)
        if T[i, j] ≠ i
            print "day j: sold stock T[i, j], bought stock i"
    if j = 1
        print "day 1: bought stock i"
```

(c) (3 marks) Analyse the running time of your algorithm from part (b).

The first (non-nested) loop iterates exactly m times.

For the nested loops, the outer loop (on j) iterates exactly n times, while the other two loops (on j and k) iterate exactly m times. Thus the innermost code (for the loops) is executed exactly $n \cdot m^2$ times.

The final for loop iterates exactly $m-1$ times, and the rest of the $Money()$ routine contributes a constant factor. Thus the $Money()$ routine runs in time $\in \Theta(nm^2)$.

The $Trace()$ procedure is recursive. Since it makes one call, and decrements j by one each time until it reaches $j = 1$, it produces a chain of n recursive calls. Since each recursive call will take constant time, the $Trace()$ routine runs in time $\in \Theta(n)$.

Therefore, the overall algorithm runs in time $\in \Theta(nm^2)$.