Solution for Test 3, Fall 2010

1. Explain/justify the recursive formula for matrix-chain multiply. The value m[i,j] is the minimum cost for multiplying matrices $A_{i-1} \ldots A_j$, with matrix dimensions $p_0 \ldots p_n$.

- First case: if there is only one matrix in the list of matrices to multiply, then the cost is zero.
- Second case: if we have two or more matrices to multiply, try all possible places to put the top-level parenthesis, and pick the minimum cost. The variable k indicates where to put the top-level parenthesis. The total cost for a fixed value of k is computed as the sum of three components: (1) the cheapest cost for multiplying the matrices before the top-level parenthesis (this is notated as m[i,k]), (2) the cheapest cost for multiplying the matrices following the top-level parenthesis (this is notated as m[k,j]) and (3) the cost of doing the matrix multiply at the top-level parenthesis (notated as $p_i \, p_k \, p_j$.

2. Explain/justify the recursive formula for longest common subsequence of sequences X and Y. X_i represents the first i items in X, and x_i is the ith item in X. The value c[i,j] is the length of the longest common subsequence of X_i and Y_j.

- First case, i=0 or j=0 means that one (or both) of the sequences X_i and Y_j is empty, so therefore a common subsequence must have length zero (there cannot be any characters in common with a sequence that is empty)
- Second case: the last character of X_i matches the last character of Y_j; that contributes 1 to the length of the longest common subsequence of X_i and Y_j. Add to that the length of the longest common subsequence of X_{i-1} and Y_{j-1}, which is written as c[i-1,j-1].
- Third case: the last character of X_i does not match the last character of Y_j. In this case, we have to pick the beset of two options: throw away the last character of X and match the remainder (find LCS of X_{i-1} and Y_j, which is written as c[i-1,j]), or throw away the last character of Y and match the remainder (find LCS of X_i and Y_{j-1}, which is written as c[i,j-1]).

For more details, see pages 392-393 of the text.

3. Here is input to matrix-chain multiply: $p_0 \ldots p_4$ = 10, 100, 5, 50, 20. Fill in the top right entry.

	0	1	2	3	4
0	▨	0	5000	7500	11000
1	▨	▨	0	25000	15000
2	▨	▨	▨	0	5000
3	▨	▨	▨	▨	0
4	▨	▨	▨	▨	▨

table[0,4] represents the multiplication of $A_1A_2A_3A_4$. Compute the cost as the minimum of these three options:

$(A_1)(A_2A_3A_4)$ table[0,1]+table[1,4]+p0p1p4 = 0+15000+10*100*20 = 20,000

$(A_1A_2)(A_3A_4)$ table[0,2]+table[2,4]+p0p2p4 = 5000+5000+10*5*20 = 11,000

$(A_1A_2A_3)(A_4)$ table[0,3]+table[3,4]+p0p3p4 = 7500+0+10*50*20 = 17,500

The second option is the cheapest, so the answer is 11,000

4. This is Fibonacci code with memoization.

```
main {
    int table[MAX];  // Table[i] stores the answer for Fib(i).
                     // MAX is an integer > 100.

    // Fill the table with -1 values to indicate that the entries are empty.
    for (int i=0; i<MAX; i++)
        table[i] = -1;

    call Fib(100);   // compute the 100th Fibonacci number
    }

// Compute the value of the nth Fibonacci number
Fib(int n) {
    int answer;
    // If the answer is in “table”, then return that answer.
    if (table[i]!=-1) return(table[i]);

    // The answer isn’t in “table”.  Compute the answer and the store
    // it in “table”.
    if (n < 2)
        answer=n;
    else
        answer = Fib(n-1) + Fib(n-2);

    table[i] = answer;
    return(answer);
    }
```

5. This is a recursive formula for cost(i,j): the cost of traveling from location i,j to the bottom row:

$$\text{cost}(i,j) = \begin{cases} C[i,j] & \text{if } i=N \\ C[i,j] + \min(\text{cost}(i+1,j), \text{cost}(i+1,j+1)) & \text{if } i<N \end{cases}$$

The above formula uses i=N as the boundary condition. It’s just as good to use i>N as the boundary condition, writing “cost(i,j) = 0   if i>N”.

The code is as follows:

```
// Given array C: C[i,j] is the cost of visiting location [i, j].
main() {
    Cost(1,1,C); // find min cost of path from row 1, col 1 to row N
    }

// The function “Cost” finds the cost of the cheapest path from
// row i, col j down to row N.
int Cost(i, j, C) {
    // Here, i==N is used as the escape clause.  Equally good is to
    // use i>N as escape clause and return 0.
    if (i == N)
        return (C[i,j]);

    // Try continuing the path straight down, or continuing
    // it diagonally down, and pick the cheaper one.
    return ( C[i,j] + min( Cost(i+1,j,C), Cost(i+1,j+1,C) ));
    }
```

Working backwards

The code above starts at 1,1 and works downwards. An alternative is to go the other way, starting at the bottom and computing the path toward the top. That is trickier to code correctly because the path can end anywhere on the bottom row, so all locations on the bottom row have to be tried in order to find the minimum cost. Also it is necessary to test whether we are on the main diagonal, or in the first column, to avoid going out of bounds. Below is code for computing the cost this way.

```
// Given array C: C[i,j] is the cost of visiting location [i, j].
main() {
    int cheapest = MAXINT;  // initialize to a huge value
    for (int j=1; j<=N; j++) {
        int temp = RCost(N, j, C);
        if (temp<cheapest)
            cheapest = temp;
        }
    // The answer is given by the value in "cheapest"
    }


// The function "RCost" ("reverse cost") finds the cost of the cheapest path
// from row i, col j up to row 1.
int RCost(i, j, C) {

    if (i<j) error in program // this should never happen

    if (i==1)
        return (C[1,1]);

    if (i==j) // we are on the main diagonal, so we have to take
              // a diagonal step, we cannot go upward.
        return(C[i,j]+RCost(i-1,j-1,C));

    if (j==1) // we are in the leftmost column, so we have to take
              // an upward step, we cannot go diagonally.
        return(C[i,j]+RCost(i-1,j,C));

    // Pick the cheapter alternative of going up or going diagonally.
    return (C[i,j] + min( RCost(i-1,j,C), RCost(i-1,j-1,C) ));
    }
```