1. First, Double Trouble is clearly in NP; one need only guess a solution and check to see if it meets the two requirements.

We will reduce Hamiltonian Circuit (HC) to Double Trouble (DT).

The easiest solution is to map a problem in HC into DT by assigning each edge cost to be zero. (The exam should have stated (as it now does) that the edge costs must be positive.) For completeness, we give a solution that uses all positive edge costs.

Method: derestriction by local substitution.

Solutions for positive edge costs.

Given an HC problem for graph G with vertices $\{1, 2, \ldots, n\}$, and edge set E, we create a new graph H where the HC problem for G corresponds to an equivalent DT problem for H as follows. Let H have vertices named a, b, c, and $2, 3, \ldots, n$. Conceptually, vertex 1 is replaced by vertices a, b and c. Vertices 2 through n are used as in G: in H, there is an edge from i to j (for $i, j \neq 1$) if there is such an edge in G. In addition, H has an edge from a to j and from c to j if G has an edge from 1 to j. Now add to H the edges $\{a, b\}$, $\{b, c\}$, and $\{a, c\}$. It is easy to see that any HC in H must have a, b, c connected in a row (as otherwise b cannot be present), and the other edges must correspond to a cycle in G. Similarly, any HC in G, plus $\{a, b\}$ plus $\{b, c\}$ gives an HC in H. Let e be the number of edges in H (which includes $\{a, b\}$ $\{b, c\}$ and $\{a, c\}$). Let $\{a, b\}$ and $\{b, c\}$ have a weight equal to x, $\{a, c\}$ have a weight of z, and all other edges have a weight of 1, All HCs in H have a weight of $2x + n$. The sum of the edge weights is $e - 3 + 2x + z$. So set $e - 3 + 2x + z = 2(n + 2x)$. This will guarantee that any solution to HC is a solution to the corresponding DT and vice versa. We just require that $e - 3 - 2n + z = 2x$. Pick z large enough to make x positive.

Let, say $z = 2n + e + 3$, and $x = e$. The purpose of edge (ac) and its weight z was just to account for cases where $e \leq 2n + 3$.

Correct solutions do not have to be nearly this detailed.

2. This problem is analogous to the dynamic programming problem for matrix multiplication.

Let $Cost[i, j]$ be the cheapest total cost for a pairwise merging of restaurants i through j.

Then

$Cost[i, i] = 0$;

$$Cost[i, j] = min_{i \leq k < j} \Big\{ Cost[i, k] + Cost[k + 1, j] + \max(S[i] + S[i + 1] + \cdots + S[k], \\ lS[k + 1] + S[k + 2] + \cdots + S[j]) \Big\}, \ i < j.$$

Clearly, an efficient implementation of this formulation will run in $O(n^3)$ time.

3. This data structure problem is to maintain a chain of elements in a way where insertions or deletion can occur anywhere in the chain, and run in $O(\log n)$ time. Moreover, the following question can also be answered in $O(\log n)$ time, where n is the number of elements in the list:

Given an element in the chain, how many elements precede it in the chain?

The solution is to maintain the chain as, say, the leaves of a structure that has the same linking as a 2-3Tree. Access is at the leaf level, and corresponds to either an insertion, a deletion, or a query. Insertion and deletion rules are the same as for the search tree. There just happen to be no key values in the structure. (This causes no difficulty since search keys are just used to locate elements in a search tree, and in this case, we get pointers to locate the elements directly.)

We augment the tree by keeping, in each internal vertex v, the number of chain elements in the subtree rooted at v. Updating these values during an insertion or deletion is straightforward. Similarly, computing the number of predecessors of some leaf is easily done by backing up the tree to the root and adding up the population counts of the leftward children of the nodes along the path from the element in question to the root of the tree.

4. The easiest answer is to give the code.

```
procedure Count(C[1..n,1..n],Pcount[1..n,1..n]);
  forall pairs i, j do
    if C[i, j] ≠ ∞ then Pcount[i, j] ← 1 else Pcount[i, j] ← 0 endif
  endfor;
  for k ← 1 to n do
    for i ← 1 to n do
      for j ← 1 to n do
        if C[i, j] > C[i, k] + C[k, j] then
          C[i, j] ← C[i, k] + C[k, j];
          Pcount[i, j] ← Pcount[i, k] × Pcount[k, j]
        elseif C[i, j] = C[i, k] + C[k, j] then
          Pcount[i, j] ← Pcount[i, j] + Pcount[i, k] × Pcount[k, j]
        endif
      endfor
    endfor
  endfor
end_Count
```

5a. The Tower of Hanoi. Recall that the algorithm is: if there are any rings to move, recursively move the top $n-1$ rings from pole A to pole C. Then move the top ring remaining on A to B, and recursively move the $n-1$ rings from C to B. Let $T(k)$ be the number of ring moves to solve this part.

The recurrence is $T(n) = 2T(n-1) + 1$, and $T(1) = 1$.

b. Now a ring of size k requires $T(k-1)$ moves, if $k > 1$. Let $S(k)$ be the number of ring moves to solve this part. The recurrence reads:

$S(1) = 1$;

$S(k) = 2S(k-1) + T(k-1)$, for $k > 1$.

c. Let $W(k)$ be the number of ring moves to solve this part. Now the work to move a ring of size k is $W(k-1)$. The recurrence reads:

$W(1) = 1$;

$W(k) = 3W(k-1)$, $k > 1$.

6a. Let G be any DFA (or NFA) that recognizes L. To get the construction for $OneReverse(L)$ right takes a little care. A reversed substring, within a string, requires a jump ahead from some state i to a future state j, followed by a backwards traversal of the DFA to i, at which point the computation jumps to state j to continue its forward wanderings through G.

We can build an NFA as follows. Let the states of G be the vertices $1, 2, \ldots, n$. Construct $n^2 + 2$ copies of G. Reverse the edge directions in n^2 of the copies, and remove all start and finish status assignments for vertices in these graphs. Let the names of these graphs be $G_{i,j}$. Let one of the other two graphs be G_s, and the other be G_f. Remove the start status assignment for the start vertex in G_f. For elegance, remove the finish status assignments for vertices in G_s.

Now build the NFA as follows. For each pair (i, j), connect an epsilon transition edge from state i in G_s to state j in $G_{i,j}$, and likewise from state i in $G_{i,j}$ to state j in G_f.

It is easy to see that the resulting NFA recognizes $OneReverse(L)$ as defined in this problem.

b. Here the proof is by contradiction. A good starting language is: Let L be the set of strings $\{a^n b^m c^m d^n | m, n \geq 0\}$. It is easy to build a PDA to show that L is CF.

Let $LL = OneReverse(L)$.

Let $L_2 = LL \cap a^*b^*d^*c^*$. Then $L_2 = \{a^n b^m d^n c^m | m, n \geq 0\}$.

Suppose that LL is CF. Then L_2 is CF since L_2 is the intersection of LL and a regular language. But the pumping lemma shows that this is not the case.

In particular, it says: There is a fixed length p where the following holds. Suppose $s \in L_2$ is longer that p. Then we can write $s = uvwxy$ where $|vx| > 0$, $|vwx| < p$ and $uv^i wx^i y$ must also be in L_2 for all $i \geq 0$. But this is nonsense. The restrictions on L_2 will require that both v and x must be strings of just one type of letter as otherwise $uv^i wx^i y$ will not be of the form $a^*b^*d^*c^*$. Let $s = a^p b^p d^p c^p$. Then w is too short to be all of the b's or all of the c's in s. Consequently, uwx can have at most two types of letters, since u has just one, as does x. But uwx cannot have just one type of letter as the pumping lemma would then produce strings outside of L_2. Similarly, uwx cannot be restricted to the two types a and b, or b and d, or d and c for exactly the same reason. No matter what, the resulting strings will be outside of L_2.

Hence, the assumption that $OneReverse(L)$ was CF is incorrect.