Design and Analysis of Algorithms, Fall 2014
Exercise II: Solutions

II-1 Where in the matrix multiplication-based DP algorithm for the all-pairs shortest paths problem do we need the associativity of matrix multiplication?

The algoritm computes the product W^{n-1} with the exception that instead of the usual definition, the product of matrices A and B is defined by

$$A \cdot B = C, \quad \text{where} \quad C_{ij} = \min_k \{A_{ik} + B_{kj}\}.$$

The slow version of the algorithm uses a recurrence $W^{i+1} = W^i \cdot W$, which gives the correct result. The fast version uses repeated squaring: $W^{2i} = W^i \cdot W^i$. We do not know a priori that the second recurrence computes W^{2i} correctly. However, from the associativity of multiplication it follows that $W^i \cdot W^i = W^{2i-1} \cdot W$, that is, the fast version works correctly.

II-2 Give an $O(n^2)$-time algorithm to find the maximum length of monotonically increasing subsequences of a given sequence of n numbers. For instance, if given $(3,1,2,5,2,6,8,6,7,3,5)$ as input, the algorithm should output 6 (the length of the subsequence $(1,2,5,6,6,7)$). (Side note: Even an $O(n \log n)$-time algorithm exists.)

We write "MIS" short for monotonically increasing subsequence. Let $A[i]$ denote the ith number in the sequence. For all $1 \leq i \leq n$ we define $L[i]$ as the length of the longest MIS that ends in $A[i]$. Then $L[1] = 1$ and for all $i > 1$ we have that

$$L[i] = \max\{L[k] : 1 \leq k \leq i-1,\ A[k] \leq A[i]\} + 1.$$

In other words, $L[i]$ is found by finding the longest MIS among the preceding numbers that can be continued by $A[i]$. Assuming the values $L[k]$ for $1 \leq k \leq i-1$ are already computed, $L[i]$ is easily computed in linear time. This yields a simple dynamic programming algorithm that computes all $L[i]$ in increasing order of i in $O(n^2)$ time. The solution is then obtained as $\max\{L[i] : 1 \leq i \leq n\}$. To find the actual longest MIS (instead of just the length), the algorithm should also keep track of the maximizing indices k, which can then be used to reconstruct the MIS.

The problem can also be solved in $O(n \log n)$ time by formulating the dynamic programming in a different manner. The algorithm initializes an array E and performs n iterations, maintaining the following invariant: After the ith iteration, each $E[\ell]$ for $1 \leq \ell \leq i$ contains the smallest number that ends a MIS of length exactly ℓ among the first i elements of A, or ∞ if no such subsequence exists.

Assuming the invariant holds, it's easy to see that E is always increasing: If there's a MIS of length ℓ ending at $A[i]$, there's also a MIS of length $\ell+1$ ending at $A[j]$ for $A[i] < A[j]$. Therefore $A[i]$ couldn't be the smallest number that ends a MIS of length ℓ.

To maintain the invariant, for iteration i the algorithm uses binary search to find the largest $E[\ell] \leq A[i]$. Since $E[\ell]$ ends a MIS of length ℓ, it is extended by $A[i]$ to produce a MIS of length $\ell+1$, and $A[i]$ is clearly the smallest number ending such a MIS so far. Hence, the algorithm updates $E[\ell+1] = A[i]$. If no $E[\ell] \leq A[i]$ exists, the algorithm updates $E[1] = A[i]$. It turns out no other changes are required.

Therefore, for each of the n iterations the algorithm performs a $O(\log n)$ amount of work, giving the claimed time complexity. After all iterations, the answer is the largest index k for which $L[k] < \infty$. Again, slight modifications are required to find the actual MIS.

II-3 (**CLRS 15-3 Bitonic euclidean traveling-salesman problem**) The euclidean traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time (see Chapter 34).
J. L. Bentley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

The first step is to sort the points according to x-coordinate. Let the sorted points be $1,\ldots,n$ and let $d(i,j)$ be the distance between points i and j.

A bitonic tour starts at the leftmost point and ends at the rightmost point. It consists of two paths, the upper and lower (imaging a line connecting the starting and end points), such that each point is visited by at least one of the paths. We describe a dynamic programming algorithm which uses partially constructed bitonic tours.

For $i,j \in \{1,\ldots,n\}$ and i, j on separate paths, let $B[i,j]$ be the minimum total cost of two paths. Now the length of the optimal bitonic tour is given by $B[n,n]$. Since $B[i,j] = B[j,i]$, we are only interested in pairs i,j with $1 \le i \le j \le n$. The base case is $B[1,1] = 0$. For the rest of the cases we compute $B[i,j]$ as follows (the idea is deciding the predecessor of point j):

- **Case 1:** If $i < j-1$, then the path ending in j must also visit $j-1$, because the other path cannot visit $j-1$ and then backtrack to i. Thus we have

$$B[i,j] = B[i,j-1] + d(j-1,j), \text{ if } i < j-1.$$

- **Case 2:** If $i = j-1$ or $i = j$, then the optimal solution must have a path which ends in j and comes from some node k with $1 \le k < j$. The optimal solution is therefore given by selecting the optimal predecessor by setting

$$B[i,j] = \min_{1 \le k < j} \{B[i,k] + d(k,j)\}.$$

  where $B[i,k]$ can be replaced by $B[k,i]$ (already computed), since $B[i,j] = B[j,i]$.

There are $O(n^2)$ entries that fall under the first case, and each of these takes a constant time to handle. The second case occurs only $O(n)$ times, with each taking $O(n)$ time. Therefore the total running time is $O(n^2)$.

To get the actual optimal bitonic tour, we modify the algorithm to keep track of the optimal predecessor chosen in the case 2 in a separate table. The optimal tour can then be constructed from this information.

II-4 (**CLRS 15-4 Printing neatly**) Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i < j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be non-negative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Let

$$s(i,j) = M - j + i + 1 - \sum_{k=i+1}^{j} l_k$$

be the number of trailing spaces required when the words $i+1,\ldots,j$ are put on a same line. Observe that is possible to put these words on a line only if $s(i,j) \geq 0$. We define the cost of putting the words from $i+1$ to j on a same line as

$$c(i,j) = \begin{cases} \infty, & \text{if } s(i,j) < 0 \\ 0, & \text{if } j = n \text{ and } s(i,j) \geq 0 \\ s(i,j)^3, & \text{otherwise.} \end{cases}$$

The dynamic programming is now straightforward to formulate. Let $C[j]$ be the optimal cost of printing the words from 1 to j, such that word j ends a line. Then we have

$$\begin{aligned} C[0] &= 0 \\ C[j] &= \min_{0 \leq i < j} \{C[i] + c(i,j)\} \quad \text{for } j > 0. \end{aligned}$$

Computing C takes $\Theta(n^2)$ time, as $c(i-1,j)$ can be computed from $c(i,j)$ in constant time. The space requirement is $\Theta(n)$. A straightforward modification gives us $O(Mn)$ time, as there cannot be more than $(M+1)/2$ words on a single line.

Again, to actually get the optimal line lengths and print the paragraph, we need to keep track of minimizing values i, which indicate the last word on the previous line, given that the last word on the current line is j.

II-5 (**CLRS 25.1-9**) Modify Faster-All-Pairs-Shortest-Paths so that it can determine whether the graph contains a negative-weight cycle.

We observe that if the graph contains a negative-weight cycle, then there is a negative-weight path of length $k \leq n$ from some vertex i to i. This means that in the matrix-multiplication-based all-pairs shortest path algorithm we will get a negative value on the diagonal of the matrix W^m for all $m \geq k$. Therefore we can detect the existence of negative cycles simply by checking whether there are negative values on the diagonal of the matrix $W^{2^{\lceil \log_2 n \rceil}}$. Note that we have to modify the algorithm to compute matrices up to at least W^n instead of W^{n-1}, as the shortest negative weight cycle might have length n.