Assignment 3
COSC 3P03: Design and Analysis of Algorithms
Winter 2016

Due: March 17, Thursday, 5:00 PM.

1. (20) Let S be a sequence of n distinct integers stored in an array as array elements $S[1], S[2], \cdots, S[n]$. Use the technique of dynamic programming to find the length of a longest ascending subsequence of entries in S. For example, if the entries of S are 11, 17, 5, 8, 6, 4, 7, 12, 3, then one longest ascending subsequence is 5, 6, 7, 12. Specifically:

define a proper function and find the recurrence for this function;

show that the principle of optimality holds with respect to the function;

design an algorithm that finds the length of a longest ascending subsequence of any sequence of distinct integers using the recurrence defined above and prints out one such subsequence;

Your algorithm must run in $O(n^2)$ time. Explain why your algorithm has a complexity of $O(n^2)$.

Let C_i be the length of a longest increasing subsequence in $S[1..i]$ **ending** at $x_i = S[i]$, $1 \leq i \leq n$. This means that $S[i]$ is the last element in the longest increasing subsequence in $S[1..i]$. The recurrence is given as follows:

$$\begin{aligned} C_1 &= 1 \\ C_i &= \max \begin{cases} C_k + 1_{1 \leq k \leq i-1} & \text{if } S[i] > S[k] \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

We first compute C_1, C_2, ..., C_n, then find $\max_{1 \leq i \leq n}\{C_i\}$. Computing C_i takes $O(i)$ time. Thus it takes $O(n^2)$ time to get all C_i's. Finding the max requires $O(n)$ time. Therefore, the total time is $O(n^2)$.

To find one such ascending subsequence, we can do some bookkeeping as follows: Create an array $K[1..n]$ such that $K[i]$ contains k where $S[k] < S[i]$ and $C_i = C_k + 1$. If C_m is the largest among $C_1, C_2, \cdots, C_n$, then the last one in the longest ascending subsequence is $S[m]$ and we find the subsequence recursively from $C_{K[m]}$.

For Questions 2 and 3, in addition to a hardcopy of your algorithms and programs, please also submit the programs using submit3p03.

2. (30) (Programming) Implement the algorithm discussed in class to find an optimal way to multiply n matrices. Please note that you need to follow the following steps:

- ask the user for an input, n and $r_0, r_1, \cdots, r_n$ that represent the dimensions of the n matrices.
- implement the dynamic programming algorithm covered in class.
- then print out the optimal way of multiplying these n matrices. For example, if $n = 4$ and the optimal way is to multiply M_2 and M_3 first, followed by multiplying the product just obtained with M_4, followed by multiplying M_1 with the last product, you should print out $(M_1 \times ((M_2 \times M_3) \times M_4))$.

We need to modify the algorithm presented in class by adding an $n \times n$ matrix $K = (k_{ij})$ where k_{ij} is used to record the k that gives us m_{ij} as in the recurrence

$$m_{ij} = \min_{i \le k < j} \{m_{ik} + m_{k+1,j} + r_{i-1} \times r_k \times r_j\}.$$

Once we have the matrix K, the routing for printing out the best order for $M_i \times M_{i+1} \times \cdots \times M_j$ is as follows:

```
order(i, j)
     if (i=j)
          print Mi
     else
          print (
          order(i, K[i][j])
          print X
          order(K[i][j]+1, j)
          print )
```

3. (30) (Programming) A one-way street $[0, \inf)$ of infinite length has a bus stop every mile. A passenger is charged according to the number of miles he rides on the bus. The bus cannot travel more than m miles ($m \ge 1$) each time (in other words, the bus can travel 1 mile, 2 miles,..., m miles non-stop. In case a rider wants to travel more than m miles, the trip has to be taken by several bus rides. For example, a trip of $m + 3$ miles can be taken as bus rides of 1 miles, 3 miles, and $m - 1$ miles; or m miles, followed by 3 miles, etc. Similarly, a trip of $m - 1$ miles could be a ride of $m - 1$ miles, or a ride of 2 miles followed by $m - 3$ miles, or even $m - 1$ rides of 1 mile each. The input is a table of length m as follows:

$Miles$	1	2	...	m
$Prices$	p_1	p_2	...	p_m

where p_i's are real numbers. If a person wants to travel n miles, design an algorithm using the technique of dynamic programming to find the optimal way for him to take the rides so that the total cost is minimum. Please note that you cannot assume anything about prices except that they are real numbers (yes, that means that they could be 0 or negative).

- define an appropriate function;
- write the recurrence;
- show that the principle of optimality holds with respect to the function;
- design an algorithm using the recurrence;
- (programming) implement your algorithm that (1) builds a price table of size m (in whatever way you want); (2) asks the user for n and then computes the optimal cost. For simplicity, you do not need to find one actual travel plan which has the minimum cost.
- be sure to test your program on several input (price tables, and different n's, which could be $>$, $=$, or $< m$).
- Your algorithm should run in $O(n^2)$ time. Justify that your algorithm runs in $O(n^2)$ time indeed.

(a) We define d_i as the optimal cost of travelling i miles. The final answer we want is d_n.
(b) Recurrence:

$$\begin{aligned} d_1 &= p_1 \\ d_i &= \min_{1\le k\le i-1}\{p_i, d_k + d_{i-k}\},\ if\ 1 < i \le m \\ d_i &= \min_{1\le k\le i-1}\{d_k + d_{i-k}\}. \end{aligned}$$

(c) Principle of optimality: if d_i is to be obtained by travelling k miles 1st, followed by travelling $i-k$ miles (or by symmetry, $i-k$ miles 1st, then k miles), then d_k and d_{i-k} must be the optimal cost to travel k and $i-k$ miles, respectively. Otherwise, d_i won't be optimal.
(d) Algorithm:

input n

if $n = 1$, return $d_n = p_1$

else if $n \le m$, compute d_2, d_3, ..., d_n, using the second line of the recurrence

else ($n > m$) compute d_2, d_3, ..., d_n using the 3rd line of the recurrence

output d_n.

(e) Time: to compute d_n, we need to compute d_1, d_2, ..., d_{n-1}. Computing d_i takes $O(i)$ time, thus the total time is $t(n) = O(1+2+\cdots(n-1)) = O(n^2)$. Note that by symmetry, the range for k can be reduced to $1 \le k \le \lfloor i/2 \rfloor$. Doing so, however, won't change the $O(n^2)$ total running time, though.

Please note that if $n > m$, then the last leg of the optimal trip has a length m or less. Therefore, in the recurrence for this case, we only need to check for $1 \le k \le m$. So now d_i can be computed in $O(m)$ time, resulting in an $O(mn)$ algorithm. For students, both $O(mn)$ and $O(n^2)$-time algorithms are acceptable.

4. (20) Adding up n integers. Given a sequence of integers a_1, a_2, ..., a_n, and we want to compute $a_1 + a_2 + ... + a_n$. Of course, this is done by $n-1$ additions. Suppose that each time, we have to add two neighboring elements, and the cost of adding two numbers a and b is the value $a+b$. In addition, we cannot rearrange the order of the numbers in the input. For example, for input 7, 2, 4, 5, 3, one way of computing the sum is by the following series of operations: $7 + (2 + 4) + 5 + 3 \to 7 + 6 + (5 + 3) \to (7 + 6) + 8 \to (13 + 8) \to 21$ with a cost of 6+8+13+21 = 48. On the other hand, the following series of operations $(7 + 2) + 4 + 5 + 3 \to (9 + 4) + 5 + 3 \to (13 + 5) + 3 \to (18 + 3) \to 21$ has a cost of 9+13+18+21 = 61. The goal is to find cheapest way to add up these n numbers using the dynamic programming method.

(a) Define $S[i, j]$ as follows:

$$S[i,j] = \begin{cases} a_i + a_{i+1} + \cdots a_j & i \le j \\ 0 & i > j \end{cases}$$

Show how to compute all n^2 entries of S in time $O(n^2)$ by dynamic programming (no need to show that the principle of optimality holds since it's not an optimization problem). Note that you need to define a recurrence (together with initial condition). Also, show the array S for the given example.

The recurrence:

$$S[i,j] = \begin{cases} 0 & i > j \\ a_i & i = j \\ S[i, j-1] + a_j & i < j \end{cases}$$

Similarly, it can also be computed as

$$S[i,j] = \begin{cases} 0 & i > j \\ a_i & i = j \\ a_i + S[i+1,j] & i < j \end{cases}$$

To compute S, we can do it in either a row-major (given below) or column-major order.

```
S = 0 /* initialize S */
for i=1 to n
   for j=i to n
       compute S[i,j] using the recurrence
```

Time: because $S[i,j]$ requires $O(1)$ time and S has $O(n^2)$ entries, the total time is $O(n^2)$.
For our example, the S is as follows:

$$\begin{pmatrix} 7 & 9 & 13 & 18 & 21 \\ 0 & 2 & 6 & 11 & 14 \\ 0 & 0 & 4 & 9 & 12 \\ 0 & 0 & 0 & 5 & 8 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

(b) Give a dynamic programming algorithm for computing the cheapest cost of adding n positive integers: Define a cost function, verify that the principle of optimality holds, and give a recurrence. Finally, use the above example to compute the cost matrix. You may assume that S (from part (a)) has already been computed and is available to your algorithm. What is the running time of your algorithm and why? Also, show the cost matrix for the given example.

This is almost identical to the problem of finding optimal order to multiply n matrices discussed in class.

Let m_{ij} be the optimal cost of computing $a_i + a_{i+1} + \cdots + a_j$, $i \le j$. Then

$$m_{ij} = \begin{cases} 0 & i = j \\ \min_{i \le k < j}\{m_{ik} + m_{k+1,j} + S[i,k] + S[k+1,j]\} & i < j \end{cases}$$

Note that $S[i,k] + S[k+1,j] = S[i,j]$..
Algorithm: compute the $n \times n$ matrix using the above recurrence. For our example, the final result is

$$\begin{pmatrix} 0 & 9 & 19 & 35 & 48 \\ 0 & 0 & 6 & 17 & 28 \\ 0 & 0 & 0 & 9 & 20 \\ 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$