CMSC351 - Fall 2014, Homework #4

Due: November 14th at the start of class

PRINT Name:

- Grades depend on neatness and clarity.
- Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.
- Write your answers in the spaces provided. If needed, attach other pages.
- The grades would be out of 100. Four problems would be selected and everyones grade would be based only on those problems. You will also get 25 bonus points for trying to solve all problems.

Problem 1 Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $(5, 10, 7, 12, 3, 40, 6)$. You have to compute the cost of an optimal solution to all subproblems.

Answer Let $(k_0, k_1, \ldots, k_7) = (5, 10, 7, 12, 3, 40, 6)$. First of all the sequence of dimensions mean that we have 6 matrices with these specified dimensions, i.e., Matrix A_i has k_{i-1} rows and k_i columns. We want to find the minimum number of operations need to find the result of $A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$. Let $C[i, j]$ be the minimum number of operations needed to calculate the result of multiplication from i-th to the j-th matrix. Note that to do this multiplication we can split the matrices into two consecutive groups and add the minimum number of operations needed to calculate result of each group and the number of operations needed to calculate result of multiplying these two groups.

$$C[i, j] = \min_{i \le t \le j-1} \{C[i, t] + C[t + 1, j] + k_{i-1} \times k_t \times k_j\}$$

$$B[i, j] = argmin_{i \le t \le j-1} \{C[i, t] + C[t + 1, j] + k_{i-1} \times k_t \times k_j\}$$

C	1	2	3	4	5	6
1	0	350	770	612	1212	1422
2		0	840	462	1662	1362
3			0	252	1092	1098
4				0	1440	936
5					0	720
6						0

B	1	2	3	4	5	6
1	0	1	2	1	4	4
2		0	2	2	4	4
3			0	3	4	4
4				0	4	4
5					0	5
6						0

This means $(A_1 \times (A_2 \times (A_3 \times A_4))) \times (A_5 \times A_6)$ is an optimal solution.

Problem 2 Given a sequence of numbers $x_1, x_2, \ldots, x_n$, give an $O(n^2)$ algorithm to find the longest increasing subsequence. Note that the subsequence does not have to be consecutive.

(a) For every $0 \leq i \leq n$, let $B[i]$ be the longest increasing subsequence of sequence of numbers $x_1, x_2, \ldots, x_i$. Compute $B[i]$ based on subproblems of smaller size.

Answer First suppose $B[i]$ is the *length* of the longest increasing subsequence of sequence of numbers $x_1, x_2, \ldots, x_i$ and not the sequence itself. I will explain at the end how to compute the sequence itself as well.

There are a few common mistake here. Consider this solution:

$$B[i] = 1 \quad \text{if there is no } x_k \leq x_i, k < i \tag{1}$$

$$B[i] = \max_{1 \leq k < i, x_k \leq x_i} \{B[k] + 1\} \quad \text{otherwise} \tag{2}$$

The length of the longest increasing subsequence is maximum of all the above $B[i]$'s. However, what is calculated here is *not* $B[i]$ asked in the problem. The above gives you the length of the longest increasing subsequence of $x_1, x_2, \ldots, x_i$ *ending in* x_i and not the length of the longest subsequence of $x_1, x_2, \ldots, x_i$. Therefore, you have to add this line to the very end of the psedocode (after calculating all $B[i]$'s in this way.)

```
1: for i ← 1 to n do
2:     B[i] = max{B[i-1],B[i]}
3: end for
```

So if I wrote on your homework *This is not exactly B*, it is because you stored the length of the longest increasing subsequence of $x_1, x_2, \ldots, x_i$ *ending in* x_i in B instead of the length of longest increasing subsequence of $x_1, x_2, \ldots, x_i$.

Now you might think that I will just change calculate $B[i]$ as $B[i] = \max B[i-1], \max_{1 \leq k < i, x_k \leq x_i}$. If you do this your algorithm would be wrong! Consider this example:

1 2 100 0 7

With the above algorithm : $B[1] = 1, B[2] = 2, B[3] = 3$. Then $B[4] = 3$ since you are getting maximum of $B[2]$ and 1. Now $B[5]$ with your algorithm would be $B[3] + 1 = 4$ (since $0 < 7$)which is not correct. $B[3]$ should be equal to 3.

So the correct way of computing $B[i]$ is to have an array C which represents the length of the longest increasing subsequence of $x_1, x_2, \ldots, x_i$ *ending in* x_i, and then at the end and after you calculated all $C[i]$ let $B[i] = max_{j \leq i}\{C[j]\}$. (You can also use array B for both purposes!)

Here is one correct answer : Let $C[1] = 1$, the

$$C[i] = \max_{1 \leq k < i, x_k \leq x_i} \{C[k] + 1\} \tag{3}$$

Then at the end:

$$B[i] = \max_{j \leq i} C[j] \tag{4}$$

(b) How many subproblems do we have? What is the running time for computing each cell of matrix B?

Answer Here we have n subproblems : $B[1, \ldots, n]$, and the running time for computing $B[i]$ is $O(i)$

(c) What are base cases? What would be their values?

Answer The best case is $B[1] = 1$. And basically $B[i] = 1$ for every i.

(d) How to fill matrix B based on a bottom-up method? Suggest a pattern.

Answer We can fill array B from left to right, i.e., start from 1 and go to n.

(e) Write a pseudo-code for filling matrix B? What is the running time of your algorithm?

Algorithm 1 LIS

```
1:  procedure LCS(x_1, x_2, ..., x_n)
2:      for i ← 1 to n do
3:          C[i] = 1                                   ▷ C[i] is at least 1 i.e, x_i itself
4:          for j ← 1 to i − 1 do
5:              if x_j ≤ x_i then
6:                  C[i] ← max{C[i], C[j] + 1}         ▷ If adding x_i to the best sequence
    ending in x_j is giving us a better answer, then we update our answer!
7:              end if
8:          end for
9:      end for
10:     for i ← 1 to n do
11:         B[i] = max{B[i − 1], C[i]}
12:     end for
13:     return B[n]
14: end procedure
```

By the way, when grading problem 2 I did not care about whether you have actually the longest subsequence itself. All I cared about was the *length*. If you want to have the longest subsequence itself, then you should have an array prev, where prev(i) stores the index j that you updated $C(i)$ from. At the end, for the maximum $C(i)$ go to prev(i) then prev(prev(i)) then prev(prev(prev(i))) and so on while you can to find the actual subsequence.

Problem 3 You are given an $m \times n$ table with some of its cells blocked. A man is standing on the upper left corner, i.e., cell $(1, 1)$. Each day if the man is standing on the cell with coordinates (x, y) it can either move to cell $(x+1, y)$, cell $(x, y+1)$, or cell $(x+2, y+2)$ provided that the cell it is moving to is not blocked or outside the table. Give an $O(mn)$ algorithm to find the number of ways the man can go to cell (m, n). For example for a 3×3 table with no blocked cell, there are 7 ways for going to cell $(3, 3)$.

Solution We fill out matrix d over the cells of the table. We move row by row and cell by cell from left to right at each row. Matrix d can be filled by the following rule for any $1 \leq i \leq m$ and $1 \leq j \leq n$:

$$d(i,j) = \begin{cases} 0 & \text{If cell is blocked} \\ d(i, j-1) & \text{If } i = 1 \text{ and } j \geq 2 \\ d(i-1, j) & \text{If } i \geq 2 \text{ and } j = 1 \\ d(i-1, j) + d(i, j-1) & \text{If } i = 2 \text{ and } j \geq 2 \\ d(i-1, j) + d(i, j-1) & \text{If } i \geq 2 \text{ and } j = 2 \\ d(i-1, j) + d(i, j-1) + d(i-2, j-2) & \text{If } i \geq 3 \text{ and } j \geq 3 \end{cases}$$

And the best case (filled before any other cell) is $d(1, 1) = 1$. The answer would be in $d(m, n)$.

Note that since we set $d(i, j) = 0$ whenever the cell is blocked, we do not have to check whether the cells we are updating from are blocked or not. For example, when you are filling $d(4, 5)$ you do not have to check whether $d(2, 3)$ is blocked since the answer in $d(2, 3)$ would be zero if it is blocked which has no effect in $d(4, 5)$ anyways.

Problem 4 Design an $O(mn)$ algorithm to compute the **length** of a longest common subsequence of two given sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$. Your algorithm should use $O(n)$ extra memory.

Hint: This algorithm is almost the same as the algorithm we have designed in the class for th LCS problem. However, your algorithm should use $O(n)$ extra memory rather than $O(mn)$.

Solution First forget about $O(n)$ extra memory and recall that if we had enough memory we could solve the problem by the following dynamic programming update formula: ($d(i,j)$ denotes the length of LCS of first i elements of array b and first j elements of array a.

$$\begin{aligned} d(0,j) &= 0 \quad \text{for any } j \\ d(i,0) &= 0 \quad \text{for any } i \\ d(i,j) &= \max\{d(i-1,j), d(i,j-1)\} \quad \text{If } a_j \text{ and } b_i \text{ do not match} \\ d(i,j) &= \max\{d(i-1,j), d(i,j-1), d(i-1,j-1)+1\} \quad \text{If } a_j \text{ and } b_i \text{ match} \end{aligned} \tag{5}$$

Now note that when we want to find the value of $d(i,j)$, all we need is the values in rows i and $i-1$. That is when we are filling out $d(10,*)$, for example, we do not need the values in $d(8,*)$ or the rows before that. So when we are filling out $d(10,*)$, we can use the space that we used when filling $d(8,*)$. Therefore, we only need two rows with n cells each. Find the values in row 1 to row m of matrix d in just two rows as follows. When updating the values of matrix d in row i, use row 0 if i is even and row 1 if i is odd. More formally, let p be 0 if i is even and 1 if i is odd. Let q be not of p meaning $q = 1$ if $p = 0$ and $q = 0$ otherwise.

$$\begin{aligned} d(p,0) &= 0 \\ d(p,j) &= \max\{d(q,j), d(p,j-1)\} \quad \text{If } a_j \text{ and } b_i \text{ do not match} \\ d(p,j) &= \max\{d(q,j), d(p,j-1), d(q,j-1)+1\} \quad \text{If } a_j \text{ and } b_i \text{ match} \end{aligned} \tag{6}$$

And the base case is

$$d(0,j) = 0 \quad \text{for any } j \tag{7}$$

Note that everything is the same as upper formula except that we replace i with p and $i-1$ with q.

Problem 5 There are n courses presented in Physics department on Monday. Course i starts at time s_i and finishes at time f_i. Design an $O(n \log n)$ algorithm to find the minimum number of class rooms needed and an assignment of the courses to the classrooms so that no two courses assigned to the same room would overlap.

Hint: Assume we have n classrooms and order all classrooms from 1 to n. Design a greedy algorithm that, at each step, selects the course with the minimum start time, assigns it to the smallest indexed classroom that is feasible given the courses already assigned, and removes this course from the list. Prove this algorithm returns the optimum solution.

Solution Suppose not. Then there is an instance for which the optimum solution is n classrooms but the greedy algorithm ends up with $n' > n$ classrooms. Let course i be the first course that is assigned to classroom $n+1$ by the greedy algorithm. Upon assignment of course i, all classrooms 1 to n must be occupied. Otherwise course i would be assigned to one of these classrooms. Consider the courses running in classrooms 1 to n at time s_i along with course i. All these $n+1$ courses are running at time s_i. Therefore, we need at least $n+1$ classrooms in any optimal solution which contradicts the fact that the optimal solution uses n classrooms.

Problem 6 Suppose we want to make change for N cents with minimum number of coins. One greedy algorithm is to start from the coin with the highest value and pick coins from it while possible then move to the second highest and so on.

(a) Prove that the greedy algorithm is optimal if the available coins are $1, 2, 5, 10$ cents.

Answer Suppose not and assume there exists N such that no optimal solution uses the first greedy choice. Now consider the following possibilities:

- If $N \geq 10$. In this case the first greedy choice would be a dime. Consider an optimum solution which doesn't use a dime. First note that the optimum solution can use at most one nickel (two nickels can be replaced by one dime for a better solution). With the same reasoning, the optimum solution uses at most two 2-cents coins (three 2-cents coins can be replaced by one nickel and one penny) and at most one penny (two pennies can be replaced by one 2-cents coins). This all add up to 10 cents. However, even for 10 cents an optimal algorithm will use a dime still.
- If $5 \geq N < 10$. In this case the first greedy choice would be a nickle. Consider an optimum solution which doesn't use a nickle. First note that the optimum solution can use at most two 2-cents coins (three 2-cents coins can be replaced by one nickel and one penny) and at most one penny (two pennies can be replaced by one 2-cents coins). This all add up to 5 cents. However, even for 5 cents an optimal algorithm will use a nickle still.
- If $2 \geq N < 5$. In this case the first greedy choice would be a 2-cents coin. Consider an optimum solution which doesn't use a 2-cents coin. First note that the optimum solution can use at most one penny (two pennies can be replaced by one 2-cents coins). This all add up to 1 cent. However, $N \geq 2$.
- If $N = 1$. Any solution will use a penny in this case, and thus the greedy algorithm is optimal.

(b) Give a counter-example to show that if the available coins were $1, 10, 15$ cents, the greedy algorithm would fail.

Answer Let $N = 20$ cents. The greedy algorithm solution would be $15, 1, 1, 1, 1, 1$ whereas the optimum solution is $10, 10$.

Problem 7 Give an $O(nm)$ dynamic programming algorithm to calculate the minimum number of coins that you need to make a change for m cents using coins of values $c_1, c_2, \ldots, c_n$.

(a) For every $0 \leq i \leq n$ and $0 \leq j \leq m$, let $B[i, j]$ be the minimum number of coins that you need to make a change for j cents using coins of values $c_1, c_2, \ldots, c_i$. Consider a decision regarding coin c_i and compute $B[i, j]$ based on subproblems of smaller size.

Answer There are two possible cases here. Either you use the coin c_i at least once or you do not use it at all. If you use it, the number of coins needed would be one plus the minimum number of coins needed to make a change for $j - c_i$ cents or $B[i, j - c_i] + 1$. If you do not use coin c_i, then it is like making change for j cents with the first $i - 1$ coins. The best answer in this case is in $B[i - 1, j]$. Since we want the minimum number of coins needed, we take the minimum value of these two. Therefore, $B[i, j] = \min\{B[i - 1, j], B[i, j - c_i] + 1\}$ if $j \geq c_i$ and $B[i, j] = B[i - 1, j]$ if $j < c_i$. Base cases would be explained later.

(b) How many subproblems do we have? What is the running time for computing each cell of matrix B?

Answer x we are calculating $B[i, j]$ for every $0 \leq i \leq n$ and $0 \leq j \leq m$, there are mn subproblems. Finding the value of each cell takes $O(1)$ time.

(c) What are base cases? What would be their values?

Answer To make things easy we let $B[0, 0] = 0$ and $B[0, x] = \infty$. You can see that with these base cases and updating rules explained in part (a) all the cells can be filled.

(d) How to fill matrix B based on a bottom-up method? Suggest a pattern.

Answer Fill the matrix with a nested loop. The outer loop is i from 1 to n and the inner loop is j from 1 to m.

(e) Write a pseudo-code for filling matrix B? What is the running time of your algorithm?

Algorithm 2 Min Coins for Making Change

```
1: procedure CHANGE(n, m)
2:     B[0, 0] ← 0
3:     for j ← 1 to m do
4:         B[0, j] ← ∞
5:     end for
6:     for i ← 1 to n do
7:         for j ← 0 to m do
8:             if j ≥ c_i then
9:                 B[i, j] ← min{B[i − 1, j], B[i, j − c_i] + 1}
10:            else
11:                B[i, j] ← B[i − 1, j]
12:            end if
13:        end for
14:    end for
15:    return B[n, m]
16: end procedure
```