# CS-3250

# Section 01

-

# Homework #5

Patrick Pei

Professor Jeremy Spinrad

Due 3-31-2017

# Exercises

**1** Show that in the matrix chain multiplcation problem, the greedy algorithm of performing the cheapest multiplication next does not always give the optimal sequence of multiplications.

| matrix | $M_1$ | $M_2$ | $M_3$ | $M_4$ |
|---|---|---|---|---|
| dimension | 10 x 10 | 10 x 9 | 9 x 10 | 10 x 10 |

By example, it can be shown with the above matrices that the greedy algorithm of performing the cheapest multiplication next does not always give the optimal sequence of multiplications. In the execution of the greedy algorithm, at first the cheapest multiplication is with the innermost matrices, $M_2$ and $M_3$, with cost $10 * 9 * 10 = 900$. Finally, the rest of the operations are equal since the matrix produced by the multiplication of the innermost matrices, $M_2$ and $M_3$, which will be labeled $M_{23}$, is identical to the remaining outer matrices. Thus, with three remaining matrices of equal dimension, it is clear that any multiplication with take $size^3$ or in this example, $10^3 = 1000$ operations. This is repeated twice since only one matrix is reduced at a time for another cost of 1000. From this, it is clear that the optimal solution produced by the greedy algorithm of performing the cheapest multiplication next is of cost 2900.

Next, it will be proven that the solution produced by the greedy algorithm of performing the cheapest multiplication next is not the optimal solution. An alternative to choosing the most attractive option at the moment would be to utilize dynamic programming. Ms learned in class, the matrix multiplcation problem takes $O(N^3)$ time and $O(N^2)$ space where $N$ is the number of matrices in the chain and the premise is to keep the optimal costs at subsequences of the multiplication given the property that all multiplications are adjacent to one another, and one at a time adding matrices until all the original matrices have been considered. By using this algorithm and analyzing not only the costs but also the breakpoints, it can be seen that the optimal solution sequence is to first multiply $M_1$ with $M_2$ to obtain $M_{12}$ and then to multiply $M_3$ with $M_4$ to obtain $M_34$ and finally to multiply $M_{12}$ with $M_{34}$ to obtain a cost of 2700, which is less than the solution produced by the greedy algorithm thus showing that the greedy algorithm does not always give

the optimal sequence of multiplications. Note: by the notation described in the CLRS textbook, the greedy algorithm would produce $((M_1(M_2M_3))M_4)$ or equivalently $(M_1((M_2M_3)M_4))$ whereas the optimal solution is in fact $((M_1M_2)(M_3M_4))$.

**2** Show that in the matrix chain multiplication algorithm, it is possible for the optimal breakpoint of $M_{in}$ to be larger than the optimal breakpoint of $M_{(i+1)n}$. This is why the optimization used for the optimal binary search tree algorithm cannot be used for the matrix chain multiplication algorithm.

It can be shown that it is possible for the breakpoint of $M_{in}$ to be larger than the optimal breakpoint of $M_{(i+1)n}$ by example. In the following matrix chain, the optimal breakpoint of $M_{1n}$ is larger than the optimal breakpoint of $M_{2n}$.

$$M_1M_2M_3M_4$$

| matrix | $M_1$ | $M_2$ | $M_3$ | $M_4$ |
|---|---|---|---|---|
| dimension | 1 x 5 | 5 x 4 | 4 x 3 | 3 x 2 |

By use of the matrix chain mulitplication dynamic programming algorithm learned in class, the following table is a representation of the state of the table, which is used in the tabular bottom-up dynamic programming approach, at the end of the execution of the algorithm. The superscript in each cell is a way to track the optimal breakpoint in each sequence.

| | $M_1$ | $M_2$ | $M_3$ | $M_4$ |
|---|---|---|---|---|
| $M_1$ | 0 | $20^1$ | $32^2$ | $38^3$ |
| $M_2$ | - | 0 | $60^2$ | $64^2$ |
| $M_3$ | - | - | 0 | $24^3$ |
| $M_4$ | - | - | - | 0 |

As is clear in this example, the optimal breakpoint of $M_{1n}$ is larger than the optimal breakpoint of $M_{2n}$. The reason behind this is that in this example the rows dimension of the $M_i$ matrix is lower than any other dimension in the rest of the matrices which makes it desirable in the context of keeping cost low. At this point, the optimal solution, since the rows dimension is so low, would obviously be to try to use this rows

dimension as much as possible. Thus, in subsequent increasing sequences from $M_{1k}$, each of these sequences must use the product as much as possible if it is truly optimal and this only comes from increasing the number of times the best value is used. The only way to increase the number of times the best value is used is to keep using it which clearly creates breakpoints as far right as possible for any sequence $M_{1k}$ where the optimal breakpoint would be $k-1$. This is obviously because to maximize the number of times it was used is limited by the fact that it always has to be multiplied in the end by something and the smallest it can possibly be multiplied by is a sequence with a single matrix $M_{kk}$. However, when this matrix $M_i$ is disincluded in the sequence and instead the optimal breakpoint of the sequence $M_{(i+1)n}$ is examined, all that needs to be satisfied is that the optimal breakpoint is not as far right as possible and is not equal to the optimal breakpoint of $M_{in}$. In this example, since the exact opposite is true, where the desirable multiplications are skewed closer to $M_n$, the optimal breakpoints are as far left as possible (to use these on the right as much as possible). Thus, by example, it is shown that it is possible for the breakpoint of $M_{in}$ to be larger than the optimal breakpoint of $M_{(i+1)n}$.

**3** In the sawmill problem, you are given a piece of wood, and a set of $n$ cuts which you wish to make on the piece of wood. The price of cutting a piece of wood is 1 dollar per foot of total length of the current piece of wood you are cutting.
Design an algorithm for finding the optimal order of cuts. Use dynamic programming; I guarantee that a greedy strategy will not work. The algorithm is very similar to the matrix chain multiplication algorithm.

In order to find the optimal order of cuts, dynamic programming must be used. This problem is extremely similar to the matrix chain multiplication problem in that the final result is always the same regardless of which approach is taken but the cost is dependent on the approach and thus needs to be minimized. In the matrix chain multiplcation problem, the number of operations is dependent on the ordering of multiplications whereas in the sawmill problem, the total cost is dependent on the order of the cuts. A bottom up tabulation approach will be used with a matrix keeping track of the optimal cost of cutting segments $i$ through $j$. Similarly to the matrix chain multiplication problem, a matrix $M_{ij}$ will represent the optimal

cost of cutting the board from start of $S_i$ to the end of $S_j$ into all segments $S_i$ through $S_j$, where $S_i$ is the $i^{th}$ segment.

The first step is to populate the matrix at $M_{ii}$ for all $i$ to 0 because obviously no cut is needed to cut a board into a segment if the board to cut is the segment itself. Next, just as in the matrix chain multiplication problem, all subsequences will be calculated by reusing the solutions to the previous overlapping subproblems. Furthermore, the optimal solution to $M_{1n}$ will be found by finding solutions to segments of length 2 to $n$ where $n$ is the number of total segments. Finally, just as was done in the matrix chain multiplication problem, the breakpoints must also be kept track of in order to give the complete ordering of cuts after the optimal cost is found. The following pseudocode (in the format of the matrix chain order problem of CLRS explains the algorithm in further detail. Realistically, just as CLRS implements the optimal breakpoints as another matrix of $O(N^2)$, the same would be done instead of the easier notation of adding a superscript to each cell as shown in the problem above.

SAWMILL-PROBLEM(S) where S is the segments

```
1  let M[1..n,1..n] be a new table
2  for i = 1 to n
3          M[i,i] = 0
4  for L = 2 to n  // L is the length of the number of segments being cut
5           for i = 1 to n - L + 1
6                   j = i + L - 1
7                  M[i, j] = infinity
8                   for k = i to j - 1
9                           q = M[i, k] + M[k + 1][j] + (S[j] - S[i - 1])
10                          if q < M[i, j]
11                                  M[i, j] = q
12                                  M[i, j](Breakpoint) = k
```

The reason that this algorithm works is because when computing $M_{ij}$, the only results that are needed are

the optimal solutions to the ordering of computing segments that are smaller than $j - i + 1$. The algorithm above, just like the matrix chain multiplication, the time complexity is $O(N^3)$ and the space complexity is $O(N^2)$ where N is the number of total segments. Finally, the following pseudocode will demonstrate how to actually extract ordering of the optimal solution. More optimally, a matrix of breakpoint would actually be stored.

PRINT-ORDER(M, i, j) where M is matrix computed above

```
1  if i == j
2          print "S" i
3  else
4          print "("
5          PRINT-ORDER(M, i, M[i, j](Breakpoint))
6          PRINT-ORDER(M, M[i, j](Breakpoint), j)
7          print ")"
```

The initial call is clearly PRINT-ORDER(M, 1, N).

**4** Show that the greedy algorithm of always choosing cuts as close to the center of the board as possible will not always give the optimal order of cuts for the sawmill problem.

The following configuration of board length and cuts demonstrates that the greedy algorithm of always choosing cuts as close to the center of the board as possible will not always give the optimal order of cuts for the sawmill problem.

$$Total\ board\ length = 10$$

$$Cuts\ at\ 4, 5, 6$$

The greedy algorithm will attempt to cut as close to the center of the board as possible and will first cut at 5 for a cost of 10. The next cuts will evidently cost 5 each for a total cost of 20. Instead, the optimal

solution as produced by the algorithm above would be to cut in order of 4, 6, 5 giving a total cost of 10 + 6 + 2 = 18. Clearly, 18 is less than 20 thus proving that the greedy algorithm of always choosing cuts as close to the center of the board as possible will not always give the optimal order of cuts for the sawmill problem. In fact, the greedy algorithm in this case actually produces the worst possible solution.