4005-800 Algorithms

Homework 5

Christopher Wood

April 29, 2012

PROBLEM 1-a.

Solution.

To find the optimal parenthesization of a matrix chain product whose sequence of dimensions is $< 5, 10, 3, 12, 5, 50, 6 >$, we simply use the $minMuls$ and $genParens$ functions to find the optimal number of multiplications and then insert the right parentheses, respectively. The steps of the mulMuls algorithm is shown below.

0	-	-	-	-	-
-	0	-	-	-	-
-	-	0	-	-	-
-	-	-	0	-	-
-	-	-	-	0	-
-	-	-	-	-	0

-	-	-	-	-	-
-	-	-	-	-	-
-	-	-	-	-	-
-	-	-	-	-	-
-	-	-	-	-	-
-	-	-	-	-	-

m and c tables for $l = 1$ (base case)

0	150	-	-	-	-
-	0	360	-	-	-
-	-	0	180	-	-
-	-	-	0	3000	-
-	-	-	-	0	1500
-	-	-	-	-	0

-	1	-	-	-	-
-	-	2	-	-	-
-	-	-	3	-	-
-	-	-	-	4	-
-	-	-	-	-	5
-	-	-	-	-	-

m and c tables for $l = 2$

0	150	330	-	-	-
-	0	360	330	-	-
-	-	0	180	930	-
-	-	-	0	3000	1860
-	-	-	-	0	1500
-	-	-	-	-	0

-	1	2	-	-	-
-	-	2	2	-	-
-	-	-	3	4	-
-	-	-	-	4	4
-	-	-	-	-	5
-	-	-	-	-	-

m and c tables for $l = 3$

0	150	330	405	-	-
-	0	360	330	2430	-
-	-	0	180	930	1770
-	-	-	0	3000	1860
-	-	-	-	0	1500
-	-	-	-	-	0

-	1	2	2	-	-
-	-	2	2	2	-
-	-	-	3	4	4
-	-	-	-	4	4
-	-	-	-	-	5
-	-	-	-	-	-

m and c tables for $l = 4$

0	150	330	405	1655	-
-	0	360	330	2430	1950
-	-	0	180	930	1770
-	-	-	0	3000	1860
-	-	-	-	0	1500
-	-	-	-	-	0

-	1	2	2	4	-
-	-	2	2	2	2
-	-	-	3	4	4
-	-	-	-	4	4
-	-	-	-	-	5
-	-	-	-	-	-

m and c tables for $l = 5$

0	150	330	405	1655	2010
-	0	360	330	2430	1950
-	-	0	180	930	1770
-	-	-	0	3000	1860
-	-	-	-	0	1500
-	-	-	-	-	0

-	1	2	2	4	2
-	-	2	2	2	2
-	-	-	3	4	4
-	-	-	-	4	4
-	-	-	-	-	5
-	-	-	-	-	-

m and c tables for $l = 6$

Now, using these values, we conclude that the minimum number of multiplications needed to evaluate this sequence of matrices is 2010. Since the sequence $< 5, 10, 3, 12, 5, 50, 6 >$ corresponds to the matrices A_1 (5 x 10), A_2 (10 x 3), A_3 (3 x 12), A_4 (12 x 5), A_5 (5 x 50), A_6 (50 x 6), respectively, we can now find the optimal parenthesization using the resulting c matrix, which yields the following result:

$((A_1A_2)(A_3A_4)(A_5A_6))$

PROBLEM 1-b. *Show that a fully parenthesization of an n-element expression has exactly $n - 1$ pairs of parentheses.*

Solution. We can prove this fact using induction on the number of matrices in a matrix chain.
Base #1: $n = 1$

By definition, an expression is fully parenthesized if it is a single element. Therefore, since $n = 1$ corresponds to an expression of a single element (a single matrix), then we know it is fully parenthesized with no parentheses. Thus, we have $(1 - 1) = 0$ parentheses for a $n = 1$ element expression.

Base #2: $n = 2$

By definition, a 2-element expression is fully parenthesized if it is the product of the two fully parenthesized elements surrounded by a single pair of parenthesis. Since single elements are fully parenthesized by themselves with no addition parenthesis, we know that an 2-element expression is fully parenthesized if we write it as the product of the two elements surrounded by parentheses. Thus, with this 2-element expression, we can make it fully parenthesized with $2 - 1 = 1$ pair of parentheses.

Induction: $n = 2$

Assume that a full parenthesization of a k-element expression has exactly $k - 1$ pairs of parentheses. Now, let $[A_1, A_2, ..., A_k]A_{k+1}$ be a $k + 1$-element expression. By the induction hypothesis, we know that $[A_1, A_2, ..., A_k]$ is fully parenthesized with $k - 1$ parentheses. Now, since A_{k+1} is a single matrix we know that it is also fully parenthesized with 0 parentheses, so we can make the full expression $[A_1, A_2, ..., A_k]A_{k+1}$ by rewriting it as $([A_1, A_2, ..., A_k]A_{k+1})$ (since it is the product of two fully parenthesized expressions). Now, since $[A_1, A_2, ..., A_k]$ contributed $k - 1$ parentheses and we have just added one more pair of parentheses, the total is now k, which is equal to exactly $(k+1)-1$.

Thus, we can see that a fully parenthesization of an n-element expression has exactly $n - 1$ pairs of parentheses. This can also be argued by observing that a pair of parentheses always wraps two operands with a single operator, and since there are $n - 1$ operators in an n-element expression, there must also be $n - 1$ parentheses if it is fully parenthesized.

PROBLEM 2-a. *Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running the recursive matrix chain algorithm?*

Solution. Running the RECURSIVE-MATRIX-CHAIN algorithm is more efficient than exhausting enumeration of the possible parenthesization of the matrix chain. Informally, it is easy to see that the recursion tree for the RECURSIVE-MATRIX-CHAIN algorithm does not contain all possible enumerations for the different parenthesization schemes for a given matrix chain. For each subproblem from index i to j that is solved by the RECURSIVE-MATRIX-CHAIN algorithm, we can see that the optimal midpoint in the chain from matrix i to matrix j is found recursively using two subproblem invocations of RECURSIVE-MATRIX-CHAIN on the left and right half of this midpoint, and then those results are simply combined to obtain the optimal result for matrices from index i to j. However, using the enumeration approach, we would calculate all parenthesization possibilies for all possible left and right halves of the matrix chain from i to j, and then we would

have to compute the result of all possible combinations of these left and right half possibilities to find the optimal value for the chain from i to j. Basically, the RECURSIVE-MATRIX-CHAIN algorithm only does one combination step after the left and right subproblems have been solved to find the optimal value, whereas the enumeration approach performs many combinations for left and right half partitions until it finds the optimal value. Thus, by this simple argument, we can see that the RECURSIVE-MATRIX-CHAIN algorithm is by far more efficient than the enumeration approach.

Formally, we know (by section 15.2 in the textbook) that the enumeration approach runs in $\Omega(\frac{4^n}{n^{3/2}})$ (similar to the *Catalan numbers*). To show that the RECURSIVE-MATRIX-CHAIN algorithm is computationally more efficient, we must establish an upper bound on its time complexity that is less than the enumeration approach. With this as motivation, we first identify a recurrence $T(n)$ for the time of the RECURSIVE-MATRIX-CHAIN algorithm as follows:

$$T(n) = \begin{cases} 1, & \text{if } n = 1, \\ 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1), & \text{if } n > 1. \end{cases}$$

This is from the observation that the comparison and multiplication operations in the RECURSIVE-MATRIX-CHAIN algorithm take constant time ($O(1)$), and at each step in the partition loop we are making two recursive calls, where the size of the index range for each recursive call adds up to n. We can make a further observation that, if the inner sum was split up, the $T(k)$ and $T(n-k)$ terms could be rewritten as $T(k)$ and $T(k)$, which implies that we can rewrite $T(n)$ as follows:

$$T(n) = \begin{cases} 1, & \text{if } n = 1, \\ 1 + \sum_{k=1}^{n-1}(2T(k) + 1), & \text{if } n > 1. \end{cases}$$

Evaluating this recurrence by treating it as a recurrence with full history, and assuming that $T(2) = 3$, we obtain the following:

$$T(n) - T(n-1) \quad = \quad (n-1) - (n-2) + 2T(n-1)$$

Simplifying and evaluation this expression yields the following:

$$\begin{aligned} T(n) &= 1 + 3T(n-1) \\ &= 1 + 3(1 + 3T(n-2)) \\ &= 1 + 3(1 + 3(1 + 3T(n-3))) \\ &= \ldots \\ &= \sum_{k=0}^{n-3} 3^k + 3^{n-2}T(2) \end{aligned}$$

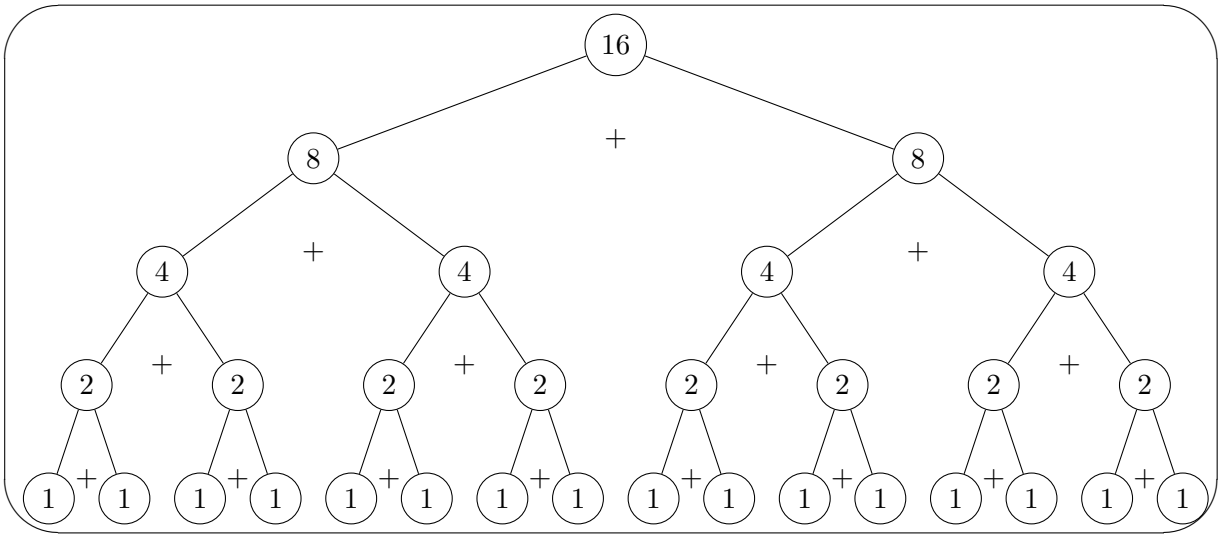
$$
\begin{aligned}
&= \frac{3^{n-2}-1}{2} + 3^{n-2}(3) \\
&= \frac{1}{2}(3^{n-2} + 2(3^{n-1}) - 1) \\
&< \frac{1}{2}(3^{n-2} + n(3^{n-1}) - 1) \\
&= O(n3^{n-1})
\end{aligned}
$$

Thus, since $T(n) = O(n3^{n-1})$ and the enumeration approach has a time complexity of $\Omega(\frac{4^n}{n^{3/2}})$, we can conclude that the RECURSIVE-MATRIX-CHAIN does indeed run more efficiently.

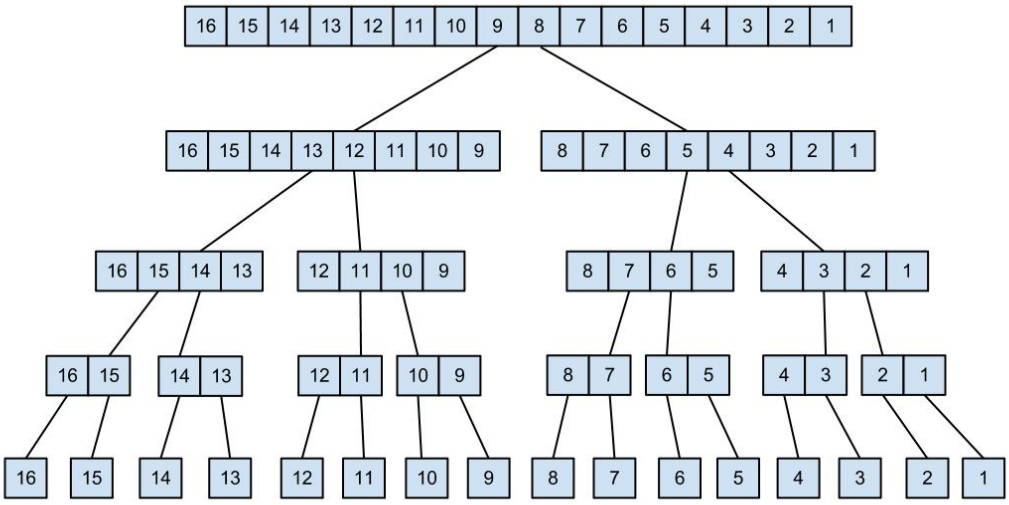
PROBLEM 2-b. *Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.*

Solution.

The recursion tree for MERGE-SORT can be shown in terms of the size n of the input sequence as follows (where each node in the tree represents the size of the input).



This same recursion tree can be shown explicitly using arrays instead, as depicted below.



One can see that there is no overlap in the sorting subproblems for this specific input array to mergesort, meaning that there is not an instance where two identitical sub-arrays are passed into mergesort (i.e. MERGE-SORT tries to sort the same sub-array twice).

Based on this observation and on the design of all design-and-conquer algorithms, memoization fails to speed up such algorithms because there are no overlapping subproblems. Memoization is most beneficial when the results of subproblems that are computed in a top-down approach can be saved in a table and then recalled when they are encountered at later recursive calls in the recursion tree. As shown by the merge-sort recursion tree above, there are no overlapping subproblems (or identical sub-arrays that are passed into MERGE-SORT), so we will never attempt to sort the same sub-arrays more than once. Thus, we would never make use of the sorted sub-arrays that have been stored, and will consequently not see any speedup. Other well-designed divide-and-conquer algorithms yield the same results, because the subproblems that are solved recursively are chosen such that they are maximally independent and overlap in as few places as possible.

PROBLEM 2-c. *Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rahter than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?*

Solution. Yes, this problem does exhibit optimal substructure. As with the minimization version of the problem, the optimal solution entails making a choice among two subproblems (the two partitions of the matrix-chain) as to which should be used to yield the maximum number of scalar multiplications. In addition, because matrix multiplication is not commutative, we are forced to

maintain the same order of the matrix chain, and thus both partitions that represent the two subproblems involved in the optimal calculation are independent. In other words, the optimal solution to the left partition of the matrix-chain (the optimal number of scalar multiplications) does not affect the optimal solution to the right partition of the matrix-chain. Therefore, since the optimal value is obtained by making a choice among two subproblems that are independent, we can conclude that this problem variant yields optimal substructure.

PROBLEM 3.

Solution. The printParagraph procedure is composed of two internal procedure calls. Namely, minLineSpaces and formatWords, which are implemented to minimize the maximum amount of white space on any one line (excluding the last), while recording the optimal newline indices, and then print the resulting paragraph based on these newline positions, respectively. Analyzing the minLineSpaces routine shows that it runs in $O(n^2)$ time, due to the fact that it considers all word sequences i through j starting from the beginning of S and terminating at the end. This time complexity is possible because we pre-compute the number of line endings for the sequence S by realizing that the equation for the number of extra white spaces is an arithmetic progression, and we can simply store the compounded result of this progression for all $i, i+1$ pairs of words in $O(n)$ time before the main work of the minLineSpaces algorithm starts.

Then, examining the formatWords routine indicates that it runs in $O(n)$ time, since it always makes one recursive call (in all cases except when the first word is reached) with a index decrement of at least 1 (meaning that in the worst case we will have one recursive call per word in S, which results in an upper bound of $O(n)$).

Now, putting the time complexity of these two routines together, we have the time T(n) for printParagraph equal to: $O(n)$ (line space precomputation) + $O(n^2)$ (DP algorithm) + $O(n)$ (format words) and we conclude that printParagraph thus runs in $O(n^2)$ time.