4005-800 Algorithms

Homework 5

Christopher Wood

May 1, 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:

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

Since the set of matrices was relatively small, we could have easily found this result using trial and error, but using the algorithm shows the inefficiency of this approach when dealing with larger matrix chains.

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

Solution.

By definition, an n-element expression is fully parenthesized if it is either a single element or the product of two fully parenthesized elements, surrounded by parentheses. We can now prove the fact that a fully parenthesized n-element expression has exactly $n-1$ pairs of parentheses using induction on the number of elements in the expression.

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 n-element expression has exactly $(n-1)$ pairs of parentheses, and let $PE = (A_1, A_2, ..., A_n)$ be such a n-element expression that is fully parenthesized. Now, assume we add an additional element A_{n+1} to PE, yielding $(A_1, A_2, ..., A_n)A_{n+1}$, which is a $(n+1)$-element expression. Since A_{n+1} is a single element, we know that it is already fully parenthesized, and since PE is fully parenthesized as well, we can make the new expression fully parenthesized by surrounding the product of these two terms with a single pair of parentheses, resulting in $PE' = ((A_1, A_2, ..., A_n)A_{n+1})$. Thus, since $PE = (A_1, A_2, ..., A_n)$ has $(n-1)$ pairs of parentheses, and we only added one more pair of parentheses to form PE', the resulting $(n+1)$-element expression PE' has $(n-1) + 1 = n$ pairs of parentheses.

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:

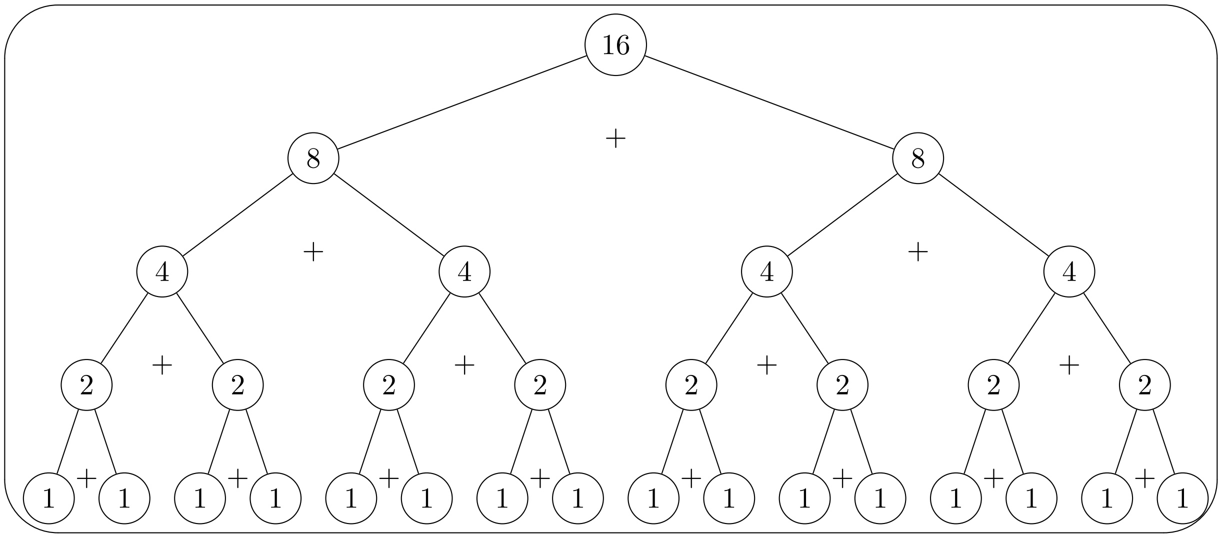
$$\begin{aligned}
T(n) &= 1 + 3T(n-1) \\
&= 1 + 3(1 + 3T(n-2)) \\
&= 1 + 3(1 + 3(1 + 3T(n-3))) \\
&= ... \\
&= \sum_{k=0}^{n-3} 3^k + 3^{n-2}T(2) \\
&= \frac{3^{n-2}-1}{2} + 3^{n-2}(3) \\
&= \frac{1}{2}(3^{n-2} + 2(3^{n-1}) - 1) \\
&< \frac{1}{2}(3^{n-2} + n(3^{n-1}) - 1) \\
&= O(n3^{n-1})
\end{aligned}$$

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 (due to the fact that the base of the exponential term is 3 for the RECURSIVE-MATRIX-CHAIN algorithm and 4 for the enumeration approach).

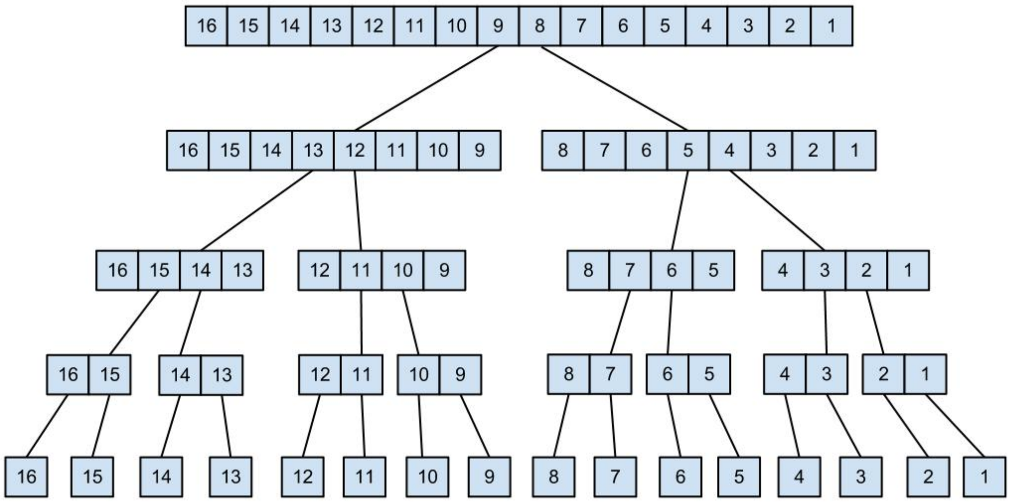
PROBLEM 2-b. *Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.*

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 no 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 designs because there are rarely cases of 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. However, as shown by the merge-sort recursion tree above, there are no overlapping subproblems (or identical sub-arrays that are passed into MERGE-SORT) for this specific input array, 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, rather 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 partitioning a matrix chain $A_iA_{i+1}...A_j$ into two separate chains $A_iA_{i+1}...A_k$ and $A_{k+1}A_{k+2}...A_j$, where k is the optimal partition index that results in the maximum number of scalar multiplications among the two smaller matrix chains. Now, by the "copy-and-paste" argument described in the textbook, we know that the solution to both partitions must be optimal as well. Consider the case where one of the partitions was not optimal. In such a case, it must be possible to find another parenthesization of such partition, "cut" it out of that partition, and "paste" it in the original matrix chain $A_iA_{i+1}...A_j$, which would yield a new optimum solution. This contradicts the fact that our k-index split was the optimum choice.

Now, using the fact that the we can solve for the optimal parenthesization for the two smaller matrix chains, we can combine these together to yield an optimal result for the original matrix chain $A_iA_{i+1}...A_j$ (i.e. utilize the optimal substructure). In other words, the optimal parenthesization for $A_iA_{i+1}...A_j$ is the result of two optimal subproblems. This implies that solving for the optimal parenthesization of $A_iA_{i+1}...A_j$ means we have to divide the matrix chain into subproblems, which can be solved to find subproblem optimal solutions, and then combine these optimal values together to yield the optimal value for the original problem.

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 partition 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.

Thus, because we must solve for optimal values of two subproblems in order solve for the optimal value of a given matrix chain, and because the solutions to these subproblems are independent by the properties of matrix multiplication, we can conclude that this problem variant does indeed exhibit 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 $O(nM)$ 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 (where i and only starts at valid positions that will allow words to fit on a line - thus at most M calculations for every word). 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) + $O(nM)$ (DP algorithm) + $O(n)$ (format words) and we conclude that printParagraph thus runs in $O(nM)$ time.