Analysis of Algorithms

Homework 4

Thomas Schollenberger (tss2344)

November 11, 2022

Question: 1

a. Consider the following chain of six matrices: $A_0, A_1, A_2, A_3, A_4, and A_5$, where A_0 is 5×10, A_1 is 10×3, A_2 is 3×12, A_3 is 12×5, A_4 is 5×50, and A_5 is 50×6. Find an optimal parenthesization of this matrix-chain. Show both the table containing the optimal number of scalar operations for all slices and the choice table.
b. Prove using the strong form of induction that for any $n \in \mathcal{N}$, if $n \geqslant 1$ then a full parenthesization of an n-element expression has $n - 1$ pairs of parentheses.

Answer to 1a: The optimal ordering is $((A_1 \times A_2) \times (A_3 \times A_4) \times (A_5))$.

i/j	0	1	2	3	4	5
0	0	150	330	405	1655	2010
1		0	360	330	2430	1950
2			0	180	930	1770
3				0	3000	1860
4					0	1500
5						0

i/j	0	1	2	3	4	5
0		0	1	1	3	1
1			1	1	1	1
2				2	3	3
3					3	3
4						4
5						

□

Proof of 1b: Suppose that a full parenthesization of an n-element matrix multiplication has $n - 1$ parentheses.
Basis:
Let $n = 2$. The optimal parenthesization, by default, is $A_1 \times A_2$

Inductive Step:
If we take $k + 1$ matricies, then the optimal parenthesization of these matricies is:
$(A_1 \times A_2 ... A_j)(A_{j+1} \times A_{j+2} ... A_{k+1})$
The two sides are implicitly parenthesized correctly. By the inductive hypothesis, we know that the first side has $j - 1$ parentheses, and the second has $(k + 1 - j + 1) - 1$. This means that the total amount of parentheses are $(j - 1) + (k + 1 - j + 1) - 1$, which simplifies into k, proving that the full parenthesization of a $k + 1$-element matrix has k parentheses or that a n-element matrix has $n - 1$ parentheses. □

Question: 2

a. CLRS 14.3-1
b. Draw the recursion tree for the merge-sort algorithm on an input sequence of length 16. Explain why memoization fails to speed up a good divide-and-conquer algorithm like merge-sort
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 the number of scalar multiplications. Does this problem exhibit optimal substructure?

Answer to 2a: Recursive algorithm. Although it does duplicate computation, it also solves the problem faster, as enumerating all ways to parenthesize the product and computing the number of multiplications for each takes much longer than just running the recursive algorithm. □

Answer to 2b: Recursive tree:
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 |
1 2 3 4 5 6 7 8		9 10 11 12 13 14 15 16
1 2 3 4		5 6 7 8		9 10 11 12		13 14 15 16
1 2		3 4		5 6		7 8		9 10		11 12		13 14		15 16

1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16
1 2		3 4		5 6		7 8		9 10		11 12		13 14		15 16
1 2 3 4		5 6 7 8		9 10 11 12		13 14 15 16
1 2 3 4 5 6 7 8		9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Memoization is the idea that we store the results of computations that occur. However, in sorting, the same computation will not occur twice, thus making it actually take slower to look up if the computation exists. □

Answer to 2c: Yes it does. Just like finding the smallest amount of scalar multiplications, it would also exhibit the optimal substructure. □

Question: 3

(**project**) Recall F_n the recurrence that defines the Fibonacci numbers. Write a function fibDyn that computes Fibonacci numbers which implements the naive recurrence via dynamic programming.

Question: 4

Consider the 0-1 knapsack problem in CLRS chapter 15 (or elsewhere).

a. Write functional pseudo-code for a recursive solution to the variation on the 0-1 knapsack problem that computes the maximum value that can be placed in the knapsack. The first parameter is a sequence of items (i.e., value-weight pairs), and the second parameter is the knapsack weight capacity. Note that weights are natural numbers.
b. (**project**) Give a dynamic programming solution to the 0-1 knapsack problem that is based on the previous problem; this algorithm should take and return the same values as the functional pseudo-code above. Implement this algorithm and call it knapsack.
c. (**project**) Implement a function called knapsackContents that takes the same values but returns a sequence of items that maximize the knapsack's value

Answer to 4a:

$$\text{ks((w, v) :: pairs, capacity)} = \begin{cases} \text{ks(pairs, capacity)} & \text{w} > \text{capacity} \\ \max(\text{v} + \text{ks(pairs, capacity - w)}, \text{ks(pairs, capacity)}) & \text{otherwise} \end{cases}$$

□

Question: 5

Consider the problem of neatly printing a paragraph on the screen (or on a printer). For the project portion, put all functions in the file printPar. The input text is a sequence S of n words (represented as strings) of lengths $l_1, . . . , l_n$ (measured in characters). The input bound M is the maximum number of characters a line can hold. The key to neatly printing a paragraph is to identify in the text sequence the lines of the paragraph so that new-lines can be placed at the end of each line. We can formalize the notion of the "badness" of a line as the number of extra space characters at the end of the line or ∞ if the bound M is exceeded. We can formalize the notion of the "badness" of a paragraph as the badness of the worst (i.e., maximum) line of the paragraph not including the last line. Thus to identify the lines for a neat paragraph, we seek to minimize the badness of the paragraph.

a. If a given line contains words i through j, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$. Write functional pseudo-code for the function $e(S, M, i, j)$ that computes the number of extra space characters at the end of a line.
b. (**project**) Write a procedure extraSpace that implements e.
c. Use the function e to write functional pseudo-code for the function $bl(S, M, i, j)$ that computes line badness.
d. (**project**) Write a procedure badnessLine that implements bl. e. Write functional pseudo-code for the

recursive function $mb(S, M)$ that computes the minimum paragraph badness (using slicing). The base case must be that the sequence S is the last line (and *not* that S is empty).
f. Write functional pseudo-code for the recursive function $mb'(S, M, \mathrm{i})$ where $mb'(S, M, \mathrm{i}) = mb(S[i:], M)$. The base case must be that the sequence characterized by S and i is the last line (and *not* that S is empty).
g. (**project**) Write a procedure minBadness that implements mb'. It should take three parameters: S, M, and i a slicing index.
h. (**project**) Write a procedure minBadnessDynamic that implements the function mb' using dynamic programming. It should take only two parameters: S and M.
i. (**project**) Write a procedure minBadDynamicChoice that implements the function mb' using dynamic programming. In addition to returning $mb(S,M)$, it should also return the choices made. Then write a procedure printParagraph which takes two parameters: S and M that displays the words in S on the screen using the choices of minBadDynamicChoice. What is the asymptotic running time of your procedure?

Answer to 5a:

$$\mathrm{e(S, M, i, j) = M - j + i} - \sum_{k=i}^{j} \mathrm{l}_k$$

□

Answer to 5c:

$$\mathrm{bl(S, M, i, j)} = \begin{cases} \infty & \mathrm{e(S, M, i, j)} < 0 \\ \mathrm{e(S, M, i, j)} & \text{otherwise} \end{cases}$$

□

Answer to 5e:

$$\mathrm{mb(s :: [], M) = bl(s, M, 0, length(s))}$$
$$\mathrm{mb(s :: S, M) = bl(s, M, i, j) + mb}(S[j+1:], \mathrm{M})$$
$$\text{where: } \mathrm{i} = 0, \mathrm{j} = \mathrm{length}(s)$$

□

Answer to 5f:

$$\mathrm{mb'(S, M, i)} = \begin{cases} \mathrm{bl(s, M, 0, j)} & \mathrm{if\ i + 1 = length(S)} \\ \mathrm{bl(s, M, 0, j) + mb(S, M,} \ i+1) & \text{otherwise} \end{cases}$$

$$\text{where: } \mathrm{s} = S[i], \mathrm{j} = \mathrm{length}(s)$$

□

Question: 6

CLRS 22.1-1

Answer to 6: Vertez z as the source:
d values:

s	t	x	y	z
∞	∞	∞	∞	0
2	∞	7	∞	0
2	5	7	9	0
2	5	6	9	0
2	5	6	9	0

π values:

s	t	x	y	z
nul	nul	nul	nul	nul
z	nul	z	nul	nul
z	x	z	s	nul
z	x	y	s	nul
z	x	y	s	nul

edge$(z,\, x)$ = 4

d values:

s	t	x	y	z
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

π values:

s	t	x	y	z
nul	nul	nul	nul	nul
nul	s	nul	s	nul
nul	s	y	s	t
nul	x	y	s	t
nul	x	y	s	t

$\square$