Algorithms and Data Structures 2020/21
Week 8 solutions

1. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 12, 5, 50, 6\rangle$.

   Answer:

   Basically this question is to show how to iterate the dynamic programming Matrix-chain algorithm given in lecture 9. We have 5 matrices $A_1, \ldots A_5$, hence we need a 5-by-5 table/array which we call m. Our first step is to set $m[i, i] = 0$ for every $1 \leq i \leq 5$ (also we black out the bottom left-hand half of the array, since cells in that part of the array represent sequences $A_i \ldots A_j$ for $i > j$, which doesn't make sense).

   In this solution, I don't actually draw out the s matrix. The entries of the s matrix only matter for sequences of ≥ 3 matrices (as there is only one possible parenthesisation for sequences of length 1 or 2). However, I do mention the values given to s in the description below for the cases of $\ell = 3$ ($A_1A_2A_3$, $A_2A_3A_4$ and $A_3A_4A_5$), of $\ell = 4$ ($A_1 \ldots A_4$ and $A_2 \ldots A_5$) and $\ell = 5$.

   Initialising the main matrix m, we get:

	1	2	3	4	5
1	0				
2	-	0			
3	-	-	0		
4	-	-	-	0	
5	-	-	-	-	0

   Now consider all "sequence windows" of length 2 ($\ell = 2$ in terms of line 4 of MatrixChainOrder). In this case there is only ever one possible split (taking one matrix on each side), hence there is no choice to be made - eg, for cell $[1, 2]$, we have $m[1, 2] = 5 * 10 * 12 = 600$.
   Doing the same operation for $m[2, 3]$, $m[3, 4]$, $m[4, 5]$, we get:

	1	2	3	4	5
1	0	600			
2	-	0	600		
3	-	-	0	3000	
4	-	-	-	0	1500
5	-	-	-	-	0

   Next we consider windows of length 3 ($\ell = 3$ in the Algorithm).
   We must fill-in $m[1, 3], m[2, 4], m[3, 5]$. I'll do $m[1, 3]$ in full:

For $m[1,3]$, we can choose $k = 1$ or $k = 2$ ($k \leftarrow i$ to $j-1$, line 8. of algorithm MatrixChainOrder). If we take $k = 1$, our cost is

$$m[1,1] + m[2,3] + p_0 p_1 p_3 = 0 + 600 + 5 * 10 * 5 = 850.$$

If we take $k = 2$, our cost is

$$m[1,2] + m[3,3] + p_0 p_2 p_3 = 600 + 0 + 5 * 12 * 5 = 900.$$

Hence we set $m[1,3] = 850$, $s[1,3] = 1$ (remember $s[i,j]$ stores the top-level split for the optimum parenthesization). After doing $m[2,4], m[3,5]$ similarly, we get the new table:

	1	2	3	4	5
1	0	600	850		
2	-	0	600	3100	
3	-	-	0	3000	1860
4	-	-	-	0	1500
5	-	-	-	-	0

We also have $s[2,4] = 3$ and $s[3,5] = 3$.

Next we do windows of length 4 - there are just two, $[1,4]$ and $[2,5]$. Doing those (I'm not giving details), we get

	1	2	3	4	5
1	0	600	850	2100	
2	-	0	600	3100	2400
3	-	-	0	3000	1860
4	-	-	-	0	1500
5	-	-	-	-	0

We also have $s[1,4] = 3$ and $s[2,5] = 3$.

Finally we must calculate $m[1,5]$. There are 4 possibilities for top-level parentheses, namely $k = 1, 2, 3, 4$. We have

$$\begin{aligned}
m[1,1] + m[2,5] + p_0 p_1 p_5 &= 0 + 2400 + 5 * 10 * 6 &&= 2700 \\
m[1,2] + m[3,5] + p_0 p_2 p_5 &= 600 + 1860 + 5 * 12 * 6 &&= 2820 \\
m[1,3] + m[4,5] + p_0 p_3 p_5 &= 850 + 1500 + 5 * 5 * 6 &&= 2500 \\
m[1,4] + m[5,5] + p_0 p_4 p_5 &= 2100 + 0 + 5 * 50 * 6 &&= 3600
\end{aligned}$$

Hence we have $m[1,5] = 2500$ and $s[1,5] = 3$.
You might want to trace back the s values to find the parenthesization.

2. Consider the following typesetting problem. The input is a sequence of n words containing $l_1, l_2, \ldots, l_n$ characters, respectively. Each line can hold at most P characters, the text is left-aligned, and words cannot be split between lines. If a line contains words from i to j (inclusive) then the number of spaces at the end of the line is $s = P - \sum_{k=i}^{j} l_k - (j-i)$ (because the words are separated by white spaces). We aim to typeset the text so as to avoid large white spaces at the end of lines, i.e., we would like to minimise the sum over all lines of the square of the number of white spaces at the end of the line.

   (a) Give an efficient algorithm for this problem.

   Answer: We use dynamic programming. Let $A[j]$ be the optimal cost (minimal sum of squares of white spaces at the end of the line over all lines) one can achieve by typesetting only the words $1, \ldots, j$ and ignoring the remaining words. Then

$$A[j] = \min_{i<j:(T[j]-T[i]-(j-(i+1))\leq P)} A[i] + (P - (T[j] - T[i]) - (j - (i+1))^2$$

   where $T[j] = \sum_{i=1}^{j} l_i$.

   I.e., we first optimally typeset words up to i and then place the remaining words from $(i+1)$ to j in the last line. (There are $(j-(i+1))$ white spaces in the last line that are not at the end of the line.)

   We can compute a table of the values $T[1 \ldots n]$, and use dynamic programming to to compute each value $A[j]$ in sequence. (Return fail if any word longer than P.)

   At the end $A[n]$ contains the *value* of the optimal solution. We can reconstruct the optimal solution itself by maintaining backpointers (as usual in dynamic programming) to record the optimal splitting of the words between lines. One can also use an auxiliary data structure to record that.

   (b) Formally prove the running time of the algorithm (matching upper and lower bounds).

   Answer: We can compute a table of the values $T[1 \ldots n]$ in $O(n)$ steps. Each of the n steps takes $O(n)$ time to compute (trying different values of $i < j$), and for $j \geq n/2$ also $\Omega(n)$, so we get $\Theta(n^2)$.

   Note: It is also possible to get a bound in terms of P, i.e., $O(n * \min(n, P))$, by using the fact that a line can hold at most $\lceil P/2 \rceil$ words (plus the separating spaces). However, generally, if one interprets n as the full size of the problem instance and encodes numbers in binary, then one could have $P = \Omega(2^n)$. One the other hand, if one explicitly considers P as a fixed constant, then the above would even yield $O(n * \min(n, P)) = O(n)$.