Algorithms and Data Structures 2014/15
Week 6 solutions (Tues 21st - Fri 24th October)

1. Show how to sort n integers in the range $\{1, \ldots, n^2\}$ in $O(n)$ time.

   This question was on the tutorial sheet for week 5. However, I only covered the material for this question today (Friday of week 5) so I put it on the week 6 sheet also - probably most tutorial groups didn't cover it yet.

   answer: This is a simple application of the Radix Sort Theorem of lecture 8. The theorem states that if we have numbers represented by b bits, we can sort in time $\Theta(n\lceil b/\lg(n)\rceil)$ time. When our numbers are the integers between 1 and n^2, the numbers of bits needed for the representation is $b = \lceil 2\lg(n)\rceil$.

   Then $\lceil b/\lg(n)\rceil \leq 4$. So Radix sort (with bits taken in $\lceil \lg(n)\rceil$ size blocks) runs in $\Theta(4n) = \Theta(n)$.

2. 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

$$\begin{aligned} m[1,1] + m[2,5] + p_0p_1p_5 &= 0 + 2400 + 5 * 10 * 6 &= 2700 \\ m[1,2] + m[3,5] + p_0p_2p_5 &= 600 + 1860 + 5 * 12 * 6 &= 2820 \\ m[1,3] + m[4,5] + p_0p_3p_5 &= 850 + 1500 + 5 * 5 * 6 &= 2500 \\ m[1,4] + m[5,5] + p_0p_4p_5 &= 2100 + 0 + 5 * 50 * 6 &= 3600 \end{aligned}$$

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.

3. We have the recurrence

$$kp_{k,\widehat{C}} = \begin{cases} 0 & \text{if } k = 0 \\ kp_{k-1,\widehat{C}} & \text{if } k > 0 \text{ but } s_k > \widehat{C} \\ \max\{kp_{k-1,\widehat{C}}, kp_{k-1,\widehat{C}-s_k} + v_k\} & \text{otherwise} \end{cases},$$

where $kp_{k,\widehat{C}}$ denotes the maximum value solution for Knapsack considering the items $1, \ldots, k$ and with capacity $\widehat{C}$.

(a) Show the recurrence is correct.

answer:

(i) The first case, when $n = 0$ is obvious. We have no items to pack, so the optimal value is 0.

If $k \geq 1$, then we focus on the final item in $\{1, \ldots, k\}$. This have value v_i and size s_i.

(ii) In the case that $s_k > \widehat{C}$, no feasible packing for $k, \widehat{C}$ can contain item k. The optimal solution is the same as the optimal one on the first $k-1$ items with the same capacity bound.

(iii) In the case that $s_k \leq \widehat{C}$, the set of feasible packings can be partitioned depending on whether they contain item k, or do not contain item k.

By definition,

$$kp_{k,\widehat{C}} = \max_{S \subseteq \{1,\ldots,k\}} \left\{ \sum_{i \in S} v_i \ : \ \sum_{i \in S} s_i \leq \widehat{C} \right\}.$$

The set of all S to be considered can be partitioned according to whether $k \in S$ or $k \notin S$. Using this partitioning, we can rewrite $kp_{k,\widehat{C}}$ as

$$\max \left\{ \max_{S \subseteq \{1,\ldots,k-1\}} \left\{ \sum_{i \in S} v_i \ : \ \sum_{i \in S} s_i \leq \widehat{C} \right\}, \max_{S \subseteq \{1,\ldots,k-1\}} \left\{ v_k + \sum_{i \in S} v_i \ : \ v_k \sum_{i \in S} s_i \leq \widehat{C} \right\} \right\},$$

where the left internal max selects the optimum knapsack not containing item k, and the right internal max selects the optimum knapsack that does contain

item k. Now observe that by definition of $kp_{k-1,\widehat{C}}$, this implies

$$kp_{k,\widehat{C}} \;=\; \max\left\{kp_{k-1,\widehat{C}},\; \max_{S\subseteq\{1,\ldots,k-1\}}\left\{v_k+\sum_{i\in S}v_i \;:\; v_k\sum_{i\in S}s_i\le\widehat{C}\right\}\right\}.$$

Also note that we have $v_k\sum_{i\in S}s_i\le\widehat{C}$ if and only if $\sum_{i\in S}s_i\le\widehat{C}-s_k$, hence we have

$$\begin{aligned} kp_{k,\widehat{C}} &= \max\left\{kp_{k-1,\widehat{C}},\; \max_{S\subseteq\{1,\ldots,k-1\}}\left\{v_k+\sum_{i\in S}v_i \;:\; \sum_{i\in S}s_i\le\widehat{C}-s_k\right\}\right\} \\ &= \max\left\{kp_{k-1,\widehat{C}},\; v_k+\max_{S\subseteq\{1,\ldots,k-1\}}\left\{\sum_{i\in S}v_i \;:\; \sum_{i\in S}s_i\le\widehat{C}-s_k\right\}\right\} \\ &= \max\left\{kp_{k-1,\widehat{C}},\; v_k+kp_{k-1,\widehat{C}-s_k}\right\}, \end{aligned}$$

where the final step follows by definition of $kp_{k-1,\widehat{C}-s_k}$.

(b) Now we use the recurrence to design our algorithm.

answer: The main issues to be considered in solving are dp1(a) and dp1(b) (the collection of subproblems and the recurrence relating the problems), dp2 (the table(s) where the results will be stored) and dp3 (the order of filling in the table(s)). For dp3, the order of filling in the table has to ensure the subproblems on the rhs of the recurrence have *always* been solved and stored (hence available for lookup) in advance of the problem on the lhs.

Now we give our solution:

dp1 dp1(a) and dp2(b). These decisions are easily made by reference the recurrence above in 2(a). This recurrence contains $kp_{k,C'}$ terms on the right-hand side, for what seems like fairly changeable values of $C'\le C$. Hence we will decide to solve $kp_{k,C'}$ for all $0\le k\le n$ and all $C'\in\mathbb{N}, C'\le C$.

dp2 We define two tables of size $(n+1)\cdot(C+1)$ each, one called kp, the other called s. The kp table stores integers (the values of the "best" knapsacks) and the s table stores binary values. For any $0\le j\le n$, $0\le\widehat{C}\le C$, the entry $kp[j,\widehat{C}]$ will denote the value of the best knapsack solution from items $1,\ldots,j$ wrt capacity $\widehat{C}$ - that is the value of $kp_{j,\widehat{C}}$. The auxiliary table s is defined as follows - $s[j,\widehat{C}]$ will be 1 if an optimal solution *does* include item j and 0 otherwise.

Note that the space used by our algorithm is already $\Theta(n\cdot C)$.

dp3 The tables are filled in increasing order of j, and then in increasing order of $\widehat{C}$.

Initialize the 0th row and 0th column of kp and of s to contain all 0s. Note that this initialization of the 0th row takes care of all instances of the "first case" of our recurrence.

Next we consider each j from $1, \ldots, n$ in turn, and for a particular j also consider all $\widehat{C}$s in increasing order. For a specific j, $\widehat{C}$, test whether $s_i \leq \widehat{C}$ (this takes $\Theta(1)$ time), and depending on the result, either do a lookup of $kp[j-1, \widehat{C}]$ or of both $kp[j-1, \widehat{C}]$ and also $kp[j-1, \widehat{C} - s_i]$. Note that by $j - 1 < j$, we have *previously* visited these cells and filled them, hence these lookups are immediate, taking $\Theta(1)$ time. Then, with these values, compare $kp[j-1, \widehat{C}]$ with $kp[j-1, \widehat{C} - s_i] + v_i$ ($\Theta(1)$ time). Take the maximum and assign $kp[j, \widehat{C}]$ this value. If the first is larger, set $s[j, \widehat{C}]$ to be 0, if the second is larger, set $s[j, \widehat{C}]$ to be 1.
The two tables can be entirely completed in $\Theta(n \cdot C)$ time. To find the actual knapsack solution (rather than just its value), we finish by starting with $j = n, \widehat{C} = C$, and outputting 'j,' if and only if $s[j, \widehat{C}] = 1$, then recursing either on cell $[j-1, \widehat{C}]$ or $[j-1, \widehat{C} - s_i]$.

4. *Longest Common Subsequence* Formally, given $s = s_1 s_2 \ldots s_n$, we say that $r = r_1 \ldots r_k$ is a subsequence of s if there is a strictly increasing sequence $i_1, i_2, \ldots, i_k$ of indices such that for all $j = 1 \ldots k$ we have $r_j = s_{i_j}$. Given two sequences x and y we say that a sequence r is a *common subsequence* if r is a subsequence of both x and y. In the *longest common subsequence* problem, we are given two sequences $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_m$ and wish to find a maximum-length common subsequence of x and y.

   Give a $O(mn)$-time DP algorithm to solve longest common subsequence.

   answer: (Students *might* notice that it can be cast in terms of edit distance). Here is a sketch of a *direct* solution. We will write lcs to denote the Length of the lcs, rather than the actual sequence.

   To give a direct answer, the main observation is that we can have a concrete view of any *common subsequence* of x and y using the concept of an *alignment*, where the two sequences x and y have '-' characters inserted into them to make x' and y' such that x' and y' are the same length *and more importantly*, when x' is laid out above y' (two consecutive rows), the only indices where *both* $x'_i \neq$ - *and* $y'_j \neq$ - are those indices where $x'_i = y'_i$; also, that reading these matching characters from left to right gives the common sequence of interest. Take as an example, the two given sequences 'miserable' and 'amiable'. A common subsequence of these two words is 'iable'. An alignment which demonstrates this is shown below:

m	-	-	i	s	e	r	a	b	l	e
-	a	m	i	-	-	-	a	b	l	e

   (note the above is *an* alignment, not necessarily the best one). Observe that the length of the common sequence is given by the number of matching characters in the alignment - hence the *longest common subsequence* problem is equivalent to finding

the alignment with the maximum number of matches. Also notice that there are *three* possible options for the final column of the alignment:

- to place x_n aligned with y_m *if we have a match* between those characters (this adds 1 to the length of the common subsequence);
- to align x_n with '-' in the final column. The best alignment which ends in this way is equal to the best alignment of $x_1 \dots x_{n-1}$ with y.
- to align y_n underneath a '-' for the final column. The best alignment which ends in this way is equal to the best alignment of x with $y_1 \dots y_{m-1}$.

dp1(a) What is the generalization we look at?

For $\mathtt{lcs}$, we will generalize to the problem of finding $\mathtt{lcs}(x[1 \dots k], y[1 \dots \ell])$, for all $0 \le k \le n$, all $0 \le \ell \le m$.

You might want to mention that there's no justification for this, YET (we could have considered generalising to computing $\mathtt{lcs}(x[k' \dots k], y[\ell' \dots \ell])$ for all k', k, ℓ', ℓ, fortunately we'll see that's not necessary).

dp1(b) We need a recurrence to justify our choice of generalization (ie, why would it be possible to use 'small' solutions to build bigger ones).

The recurrence is

$\mathtt{lcs}(x[1 \dots k], y[1 \dots \ell])$

$$= \begin{cases} 1 + \mathtt{lcs}(x[1 \dots k-1], y[1 \dots \ell-1]) & \text{if } x_k = y_\ell \\ \max\{\mathtt{lcs}(x[1 \dots k-1], y[1 \dots \ell]), \mathtt{lcs}(x[1 \dots k], y[1 \dots \ell-1]\} & \text{otherwise} \end{cases}$$

dp2 What size table do we need to store our solutions?

We will need a table of size $(n+1)(m+1)$ (to store $\mathtt{lcs}$ for every k, ℓ).

dp3 What are the rules for filling-in the table?

For $k = 0$ we fill the values of this row directly, setting $\mathtt{lcs}(x[1 \dots 0], y[1 \dots \ell]) = 0$ for every $0 \le \ell \le m$. We also fill in column 0 the same way.

Then for $k \leftarrow 1$ to n (in increasing order) we fill in row k in one go as follows: we generate the values for $\mathtt{lcs}(x[1 \dots k], y[1 \dots \ell])$ in terms of increasing ℓ, using the recurrence above.

This method of doing the rows in increasing order of k, and within each row, in increasing order of ℓ, ensures that the $\mathtt{lcs}$ values from the right-hand side of the recurrence above are ALWAYS available in advance.

- Running time?

  $\Theta(nm)$.

Mary Cryan