Algorithms and Data Structures 2014/15  
Solutions/marking-scheme for Coursework 2

Sample implementations for the coding parts of this coursework are given in LineFormatSol.java.

1. (10 marks in total)

The question asked students to implement the greedy algorithm presented in the specification. One example of an implementation is given in LineFormatSol.java.

**marking:** The testing of correctness is done via a series of tests which are the initial section of my maintainests file:

- `greedyLF` should return *either* the empty array [], *or* an array with one negative entry (probably [-1]) when called with the empty array;
- `greedyLF(lens2)` should return the array [0] if lens2 is [20] (or any array containing a single value (less than 80));
- `greedyLF(lens3)` should return [8, 11] when lens3 is the array [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8];
- `greedyLF(lens4)` should return [7, 11] when lens4 is [8, 8, 8, 8, 8, 8, 8, 9, 8, 8, 8, 8];
- `greedyLF(lens5)` should return [7, 12] when lens5 is [10, 11, 8, 12, 9, 8, 8, 6, 12, 9, 9, 9, 11];
- `greedyLF(lens6)` should return [7, 14, 21] when lens6 is [10, 11, 8, 12, 9, 8, 8, 6, 12, 9, 9, 9, 11, 7, 10, 11, 12, 6, 4, 8, 2, 11];

Apart from checking the test results, you should examine the code to make sure it is a reasonably faithful implementation of the greedy pseudocode.

- Answers that are totally wrong may be due to indexing confusion, so if that is the case, see if the alternate tests pass (maintests2). I will not penalize for this *unless the student uses DIFFERENT indexing conventions for the different implemented methods* (-3 overall penalty if this happens, you'll only know after marking the later implementations).
- Give 10 marks if all tests are passed and the implementation is faithful.
- Give 9 marks if all tests except lens1 (the empty array) are passed, and the implementation is faithful.
- Give 8 marks if the student has a faithful implementation which fails to append the n-1 at the end, or the n if the student is using a different indexing, (of course the score will be only 7 if the test on lens1 also fails).
2. (15 marks in total)

The $\Theta(\ell)$ algorithm for optimal line formatting on lines of length $L$ with respect to the measure $ss$ is based on the following recurrence, given in the coursework specification, where $\ell_1, \ldots, \ell_n$ is the given list of word lengths:

$$
vopt(\ell) = \begin{cases} e_\ell(1,n)^2 & \text{if } e_\ell(1,n) \geq 0 \\ \min_{k} \{ e_\ell(k,n)^2 + e_\ell(k+1,n)^2 \} & \text{if } e_\ell(1,n) < 0 \end{cases}, \quad (1)
$$

and where $e_\ell(i,j)$ is defined as

$$
e_\ell(i,j) = L - \left( \sum_{j=i}^{i'} \ell_j \right) - (i' - i),
$$

where $\ell = (\ell_1, \ldots, \ell_n)$.

We will first describe the $\Theta(\ell)$ dynamic programming algorithm based on (1), and afterwards will mention how it can be adjusted to be $\Theta(n \cdot L)$ (note that either solution is fine for your submission).

For the $\Theta(\ell)$ algorithm, our algorithm will begin by computing a table of the $e_\ell(i,j)$ values for all $i, j, i \leq j$ (this step taking $\Theta(n^2)$ time). Then a table (array) $vopt$ of length $n$ is defined, and $vopt[i]$ is computed in increasing order of $i$, according to the following recursive formula:

$$
vopt[i] = \begin{cases} e_\ell(1,i)^2 & \text{if } e_\ell(1,i) \geq 0 \\ \min_{k<i} \{ e_\ell(k,i) + e_\ell(k+1,i)^2 \} & \text{otherwise} \end{cases} \quad (2)
$$

The $vopt[i]$ table is one of the main tables of the $\Theta(\ell)$ algorithm; we will also maintain a table $break$ of length $n$ in parallel with $vopt$, with $break[i]$ marking the “last-but-one breakpoint” $k$ which minimizes the sum-of-squares for $w_1, \ldots, w_i$.

• Students were asked to discuss the collection of subproblems that will be solved.

The collection of sub-problems that will be solved are $vopt(\ell[k])$ for every $i = 1 \ldots n$ (recall that $\ell[k]$ is the prefix of word lengths $\ell_1, \ldots, \ell_k$).

**marking (2 marks):** 2 marks for either stating “we will solve the subproblems ...” (with correct details given) or writing very clear pseudocode where the subproblems being solved are clear/explicit.
Students were asked to give details of the tables used by their algorithm.

**answer:** The algorithm will have two main tables $v_{opt}$ and $\text{break}$, each of length $n$. $v_{opt}[k]$ will store the value of the optimal formatting of $\ell[k]$ once that is computed using (1), while $\text{break}[k]$ will store the “top-level breakpoint” for the/a line formatting which realises that optimum value.

My algorithm will also have a two-dimensional table called $e$ of size $n \times n$, which will be filled with the values of $e_{\ell}(i, j)$ for all $1 \leq i \leq j \leq n$. This table is not necessary, and would not appear in an $\Theta(n \cdot L)$ algorithm.

**marking (2 marks):** 2 marks if the details of the tables, and what values are stored in the cells of those tables, are explained clearly. We actually want something written down here, it’s not enough just to see the table names inside the pseudocode for this.

Students were asked to explain how their algorithm fills the tables and constructs the solution.

**answer:** Below is the pseudocode for the $\Theta(n^2)$ solution. Note that students are perfectly welcome to just explain their algorithms in sentences, assuming the explanation is reasonably detailed. My algorithm implements (1) fairly directly - only extra detail to note is that in the interesting case of $e_{\ell}(1, i) < 0$, that in considering the elements of the set $\{k < i, e_{\ell}(k + 1, i) \geq 0\}$, that in order to have $e_{\ell}(k + 1, i) \geq 0$, we require at a minimum that $(\sum_{j=k+1}^{i} \ell_j) + (i - (k + 1)) \geq L$, and hence require (by $\ell_j \geq 1$ for all $j$) that $2(i - (k + 1)) + 1 < L$, which is the case only if $k + 1 \geq i + (L - 1)/2$, which is true only if $k \leq i - (L + 1)/2$. This observation justifies the bounds set on $d$ on line (n) below.

There may be some submissions which give $\Theta(n \cdot L)$ algorithms - those probably won’t have a table to store the $e_{\ell}(i, j)$ values, and will be missing the section in lines (b)-(e). Instead a single variable (ee, say) will probably be initialised to $L$ just after entering the loop at (g), and this will be decreased as $d$ is increased, with the condition of the for-loop at (n) testing $ee$ against 0.
Algorithm DynamicLFall(ℓ)

(a) n ← ℓ.length
(b) for i ← 1 to n do
(c) e[i, i] ← L − ℓi
(d) for j ← i + 1 to n do
(e) e[i, j] ← e[i, j − 1] − ℓj − 1
(f) vopt[0] ← 0
(g) for i ← 1 to n do
(h) if e[1, i] ≥ 0
(i) vopt[i] ← e[1, i]2
(j) break[i] ← nil
(k) else
(l) vopt[i] ← ∞
(m) break[i] ← nil
(n) for d ← 1 to min(⌈L/2⌉, i) do
(o) k ← i − d
(p) q ← (e[k + 1, i]2 + vopt[k])
(q) if ((e[k + 1, i] ≥ 0) and (q < vopt[i])) then
(r) vopt[i] ← q
(s) break[i] ← k
(t) fi
(u) fi
(v) return vopt[n];

marking (5 marks): The 5 marks go for the details of filling the main tables vopt and break if correct, to include initialisation. If their inner loop only examines an O(L) number of different ks like mine, they must justify this in discussion like I did (1 mark deleted if not). If they have extra tables like my e[i, j] table, of course they must explain how that is filled.

Partial marks go for sketchy answers or algorithms with bugs (though 1 small error/typo is allowed before deleting any marks).

Some students may have given a algorithm which would have time Θ(n³) - these occur if the algorithm uses a different recurrence to (1), where the right-hand side for the recursive case is instead \( \min_{k:1 \leq k < n} \text{vopt}[1, k] + \text{vopt}[k + 1, n] \). While correct, this resulting algorithm will be slower than what I asked for. If they have described the \( \Theta(n^3) \) algorithm instead of the one asked for, HERE is the place to delete marks for it; work out of 3 marks rather than 5. In the other parts, just mark as already described (full marks for a correct description of the tables they used, etc). The only other place extra marks might be to be deleted would be if they were erroneously trying to show such an algorithm was \( O(n^2) \).

- Give details of how the actual breakpoint sequence \( \text{opt}(\ell) \) itself is computed, in addition to the value \( \text{vopt}(\ell) \).
answer: This can be easily computed in a recursive fashion working from the array `break` which was built during `DynamicLFall` (note that `DynamicLFBreak2` accepts this array as its input):

Algorithm `DynamicLFBreak(ℓ)`

(a) Compute the `break` array as in `DynamicLFall`
(b) return `DynamicLFBreak2(break, n, n)`

Algorithm `DynamicLFBreak2(break, n)`

(a) if `n == 0`
(b) return `nil`
(c) else
(d) `k ← break[n]`
(e) return `DynamicLFBreak2(break, k, k)`
(f) fi

marking: 2 marks for any decent answer which shows how to take the array of “top-level breaks” from the main algorithm and turn it into the `opt` sequence for the line formatting. They should remember to add the `n` at the end, else a penalty of 0.5-1 marks off.

The response of just returning the `break` array directly should be given 0 marks as this is far from the correct answer.

• (4 marks) Detailed justification of the running time of the algorithm (to be either \(\Theta(n \cdot L)\) or \(\Theta(n^2)\).

answer: For Algorithm `DynamicLFall` above, the running time is \(\Theta(n^2)\).

Line (a) clearly takes \(\Theta(1)\) or \(\Theta(n)\) time, depending on implementation; lines (f) and (v) take \(O(1)\) time. Next consider the declaring/building of the `e` table; lines (b)-(e) contain a pair of nested loops with overall running time \(\sum_{i=1}^{n}(\Theta(1)+\sum_{j=i+1}^{n}\Theta(1))\), ie, \(\Theta(1)(\sum_{i=1}^{n}\sum_{j=i}^{n}1) = \Theta(1)(\sum_{i=1}^{n}(n+1-i))\), which is \(\Theta(n^2)\).

The rest of the algorithm consists of the for-loop in lines (g)-(u), which contains a nested for-loop in lines (n)-(t). The loop between (n)-(t) will run at most \(\lceil L/2 \rceil\) times for any \(i, 1 \leq i \leq n\). Also all of the operations on lines (n)-(t) take \(\Theta(1)\) time, regardless of \(i\) or \(j\)’s values. Hence, the loop on lines (n)-(t) is \(O(1)\) and \(O(L)\) always, and will be \(\Theta(L)\) for any \(j \geq L/2\). To analyze the overall running time for the loop in lines (g)-(u), observe that lines (h)-(m) are each \(\Theta(1)\). Putting this together with the asymptotics for the inner loop, the for-loop from (g)-(u) takes \(O(n \cdot L)\) time. Also, if \(n\) is sufficiently large (\(n \geq 2L\) (say)), then the interior loop is \(\Omega(L)\) for at least \(n/2\) values of \(i\), hence we have a corresponding \(\Omega(n \cdot L)\) lower bound for lines (g)-(u) for sufficiently large \(n\).

Note that the algorithm `DynamicLFBreak2` (which would be included in a full implementation of the DP algorithm) is \(O(n)\) algorithm; hence the generation of the
breakpoint sequence opt from the DP array does not affect the overall running-time.

**marking:**
- Any solution as detailed as mine above would get 4 marks. They must justify the $\Omega(\cdot)$ as well as the $O(\cdot)$ bound;
- If they skip the $\Omega(\cdot)$ but do the $O(\cdot)$ they get between 2-3 marks (and only get 3 if their $O(\cdot)$ analysis is very good).
- If they give arguments which are casual with justification (e.g., analyse only one part of their algorithm, and ignore other parts of lower complexity without justification), they will lose at least 1 mark for that (on top of $O(\cdot)$, $\Omega(\cdot)$ considerations).
- **mistakes:** if they claim $\Theta(n \cdot L)$ but their algorithm is $\Omega(n^2)$ (this would be if the test for the inner for-loop at (m) ranged over $\Omega(n)$ indices), or if they gave the $\Theta(n^3)$ algorithm but are claiming better, then they get at most 2 marks and probably less.
- If they just state complexities without referring to any details of steps/loops of their algorithm, at most 1 mark.
- If something is mostly wrong, then at most 1 mark.

Note they don’t need to actually do the analysis of the two loops in as detailed a way as I did. If they were trying to show $\Theta(n \cdot L)$ they’d only need to show the (b)-(e) section was $\Theta(n^2)$ and the later section $O(n \cdot L)$ - I’ve put in the $\Omega(n \cdot L)$ argument for the main loop, to set things up for discussing the alternative solution (see below).

**alternative solution:** They would also get up-to-4-marks for a argument that their algorithms is $\Omega(n \cdot L)$, if that is the case.

These algorithms will probably not have the $e$ array of the (b)-(e) loop (they could set this up as a $n \times L$ or $n \times \lceil L/2 \rceil$ array I guess, but indexing would be messy). They will probably have a variable $ee$ (say) within (f)-(t) being initialised/updated to hold the current value of $e(i,j)$ as the for-loop of (m)-(s) iterates - for a $O(n \cdot L)$ solutions, this will have to be updated for the new $j$ in $\Theta(1)$ time, so check this. The test of the inner loop at (f)-(t) will be a test of $ee$ being non-negative. Then (assuming the update to the “leftover space” variable $ee$ is done in $O(1)$ time), my argument above the “marking” section shows that the entire algorithm is $\Theta(n \cdot L)$.

Marks for the analysis of a $\Theta(n \cdot L)$ solution should be allocated in a similar way as for the $\Theta(n^2)$ version.
3. Next task was to design a $\Theta(n^2)$ algorithm for the optimal line formatting of a list of words into lines of length $L$ with respect to the measure $ss^*$, where the square of leftover-space for the final line is omitted.

**answer:** To develop the algorithm for this, it helps to make an observation regarding $vopt^*(\ell)$ and $opt^*(\ell)$ and their relationship to $vopt, opt$: that the value $ss^*(\ell, i)$ of any feasible formatting $i = (i_1, \ldots, i_m)$ (recall we always have $i_m = n$) equals the value $ss(\ell[i_{m-1}], (i_1, \ldots, i_{m-1}))$. Suppose we are considering the/a list of breakpoints $i = (i_1, \ldots, i_m)$ which is the optimum solution $opt^*(\ell)$ for the word-length sequence $\ell$. Then note that by definition $vopt^*(\ell) = ss(\ell[i_{m-1}]; (i_1, \ldots, i_{m-1}))$. However, even more than that is true - it must be the case that $ss(\ell[i_{m-1}]; (i_1, \ldots, i_{m-1})) = vopt(\ell[i_{m-1}])$. To see this, note that if there was a different sequence $j$ of breakpoints for $\ell[i_{m-1}]$ with a higher value under $ss$ than $(i_1, \ldots, i_{m-1})$, then this would imply $i$ was not optimal for $ss^*$ for $\ell$.

Hence, if $i_{m-1}$ is the penultimate breakpoint of the formatting which realises $vopt^*(\ell)$, then $vopt^*(\ell) = vopt(\ell[i_{m-1}])$, where $vopt$ is the array computed by DynamicLFall for the sequence $\ell$.

Of course, given a line length sequence $\ell$, we will not know a-priori the particular index $i_{m-1}$ which corresponds to an optimum penultimate breakpoint for $vopt^*, opt^*$; however, we do know that it must satisfy $e_2(i_{m-1} + 1, n) \geq 0$. By similar reasons to those given at the start of our (b) solution, this implies that we must have $i_{m-1} \geq n - \lceil L/2 \rceil, i_{m-1} < n$. Hence there are only $\lceil L/2 \rceil$ different options for $i_{m-1}$. Also, the value of $vopt^*(\ell)$ will be equal to the value $vopt[i_{m-1}]$ already computed as part of the call to $\text{DynamicLFall}(\ell)$. Hence to find $vopt^*(\ell), opt^*(\ell)$ it is sufficient to iterate over all (at most $\lceil L/2 \rceil$) indices $i_{m-1} < n$, looking-up and comparing the precomputed values of $vopt(\ell[i_{m-1}])$ from the pre-computed table.

**Algorithm** $\text{DynamicLFallButLast}(\ell)$

(a) Compute the $vopt, break$ arrays as in $\text{DynamicLFall}$
(b) $n \leftarrow \ell.\text{length}$
(c) $k^* \leftarrow n$
(d) $k \leftarrow n - 1$
(e) $ee \leftarrow L - \ell_n$
(f) $\textbf{while } ((k > n - \lceil L/2 \rceil) \text{ and } (ee \geq 0))$
(g) $\textbf{if } (vopt[k] < vopt[k^*])$
(h) $k^* \leftarrow k$
(i) $ee \leftarrow ee - (\ell_k + 1)$
(j) $k \leftarrow k - 1$
(k) $\textbf{return } vopt[k^*]$

If we want to instead compute the breakpoint sequence which realises the value $vopt^*(\ell)$, then we would change the final line to become $\text{DynamicLFBreak2}(\text{break}[k^*], k^*, n)$.

The algorithm takes $\Theta(n^2)$ or $\Theta(n \cdot L)$ time depending on which implementation for $\text{DynamicLFall}$ (called in line (a)) was used. The work done on lines (b)-(e) is $\Theta(1)$, and the
while-loop from (f)-(j) iterates at most $\lceil L/2 \rceil$ times, doing $\Theta(1)$ work on each iteration - so $O(L)$ time in total. If we do have a call to DynamicLFBreak2 on the final line this is $O(n)$ as mentioned above.

**marking:** 3 marks for the details of the algorithm, 1 mark for justifying the top-level step (how we can find the value of $v_{opt}^*$ by examining a limited number of $v_{opt}[k]$ entries for $k$ near $n$). The final mark goes for *either* doing a very good job of the above, *or* for giving some details of running-time.

4. (10 marks) This part of the coursework asked the student to implement their algorithms from parts 2 and 3 as the following methods in LineFormat.java:

```java
public static int[] dynamicLFall(int[] wordLengths)
public static int[] dynamicLFallbutlast (int[] wordLengths)
```

The array of integers returned should be the breakpoint sequence for an optimal line formatting (ie, the sequences returned by DynamicLFBreak and by the adjusted version of DynamicLFallbutlast). This part of the coursework will be assessed by a mixture of tests and examination of pseudocode.

**marking:** The testing of correctness is done via a series of tests which use the same arrays as the greedyLF tests. The code for testing is in maintests (though you may need to use maintests2 or maintests3 for some submissions). Those arrays were set up carefully to

- test “base cases” (arrays lens1 and lens2)
- test the very exact details of adding-the-spaces for greedyLF (this also gets tested for dynamicLFallbutlast); this is seen by the difference between the result given for lens3 and lens4, where a single (crucial) cell entry differs by 1;
- show examples of differences between greedyLF and dynamicLFall (arrays lens3, lens4, lens5);
- show that when working with dynamicLFallbutlast, that results of 2-line inputs are close to that for greedyLF (lens3, lens4, lens5), but when we go to 3 lines dynamicLFallbutlast we may get an answer different from greedyLF and dynamicLFall (this is the test with lens6).

To mark, examine the results maintests, and also look in the student’s file to see the details of their implementations for the two functions.

- You may have previously noticed (back when marking greedyLF) that the student is using a different indexing convention; hence you may be working with maintests2. Do not penalize for this *unless the student is using DIFFERENT indexing conventions for the different implemented methods* (-3 overall penalty if this happens).
- 7 marks are going for dynamicLFall:
– Check the code to see if it looks like a faithful approximation to the Algorithm they developed if it is, and if all tests pass, award 7.
– If code looks faithful, and all tests except one/both of lens1, lens2 pass, 6 marks.
– If code looks faithful, and all tests except one of lens3, lens4 pass, and it looks like the counting-spaces-error, award 6 marks.
– If code looks faithful, and error is that the n-1 is not getting appended to the list of breakpoints, 6 marks.
– If code looks faithful, and two or three of the issues above happen, 5 or 4 marks.
– If more than this is wrong, look again at their code.
  If the code is maybe doing all the table-building correctly, but can’t get the array-of-breakpoints right, then work out of 5 or 4 (4 if they don’t even try to do opt, or just copy the breaks array for this). There could be extra deductions for bugs mentioned above, but don’t give less than 2-3 if they have done the table building.
  For anything that isn’t doing the table-building right, give anything between 0 (maybe they just duplicated “greedy” or similar when coding) and 3 (got some part of the way there).

• 3 marks are going for dynamicLFallbutlast.
  – For this one I’m going to just focus on the lens5, lens6 results, and will ignore the issue of the appended n-1 (unless it’s new for this method, and hasn’t previously been penalized).
  – If tests lens5, lens6 pass, give 3 marks. Still have a look in code as you might want to make a comment about their implementation if it’s nice (or maybe if it could be shortened)
  – If tests lens5, lens6 don’t both pass, check the code to see if a decent attempt is being made (eg, they could just be using greedy in which case they should get 0 here). If a decent attempt is being made, give 2 (note lens6 is a bit “brittle” with counting the spaces, that could explain a false result).

5. (10 marks) The final section asked students to write a report discussing experimental analysis of their implementations. The task was described as shown below, so please allocate 10 marks in a sensible manner in correspondence to that. For generating their experimental data, I’d expect word-length sequences either based on paragraphs scraped from the web/real-life, or a sensible random model of natural sequences (as discussed at the end).

To get the full 10 marks they’d need to study at least two of the possible comparisons suggested below, and do thorough experiments reported well. Preferably the differences in the performance of the algorithms would be described using relative values (ratios).

One extra thing they might have done would be comparing running-times of different algorithms; that is fine if they’ve done that, but it would require quite long word-length sequences to be significant . . . . Please give detailed feedback on graphs purporting to show
\(\Theta(\cdot)\) bounds wrt practical data, and mention the issues with that approach (unknown c etc).
Experimental running-time analysis would contribute at most 5 marks, and would need to be very good.

You should consider a collection of randomly generated sequences of word lengths, and compare the outputs generated by your different algorithms. Some the results of interest would be:

- The typical/average/max difference between \(ss(\ell; i)\) when \(i\) is computed by greedyLF, in comparison to when it is computed (for the same \(\ell\)) by your dynamic programming method dynamicLFall.
- The typical/average/max difference between \(ss^*(\ell; i)\) when \(i\) is computed by greedyLF, in comparison to when it is computed (for the same \(\ell\)) by your dynamic programming method dynamicLFallbutlast.
- The frequency with which \(opt(\ell)\) describes a different breakpoint sequence to \(opt^*(\ell)\).

To set up your experiments, you should define a sensible random method which generates sequences of word lengths which model the English language well (with word lengths between 2 and 7 being common, lengths 1 and 8 being fairly common, and lengths larger than 8 less common). You should base your experiments on reasonably-long sequences of words (which will require 10 or more lines of length 80).

Mary Cryan