CSC373— Algorithm Design, Analysis, and Complexity — Spring 2018

Assignment 2: Dynamic Programming — Due: 9am, Thurs, Feb 15, 2018

Submit answers to the course MarkUs page, `https://markus.teach.cs.toronto.edu/csc373-2018-01`. Your solutions must be in one PDF file named A2.pdf. Your solutions can be neatly hand-written and scanned. You do not need to repeat the questions themselves, and you can make use of any material in the class's lecture notes or tutorials simply by citing them.

You are encouraged to work in groups of size at most five. If you do work in a group then you must, of course, submit this work to MarkUs as a group.

1. **[6pts] Sub-Palindromes.** A subsequence is **palindromic** if it is the same whether read from left to right or right to left. For instance, the sequence

   $$(A, C, G, T, G, T, A, T, G, C)$$

   has many palindromic subsequences. Examples of palindromic subsequences of this sequence include (C, G, T, A, T, G, C), (A, T, G, T, A), (A) and (T, T), which have lengths 7, 5, 1, and 2, respectively.

   Develop an algorithm that takes an input sequence $x = (x_1, x_2, \ldots, x_n)$, for some $n \geq 1$, and returns the length of the longest palindromic subsequence. Your algorithm should run in $O(n^2)$ time and require $O(n^2)$ space. You can refer to algorithms and notation used in the lecture notes without repeating them in your answer. (Hint: Here that is particularly helpful.)

2. **[10pts] Parsing Words.** We wish to parse a sequence of letters, say $y = (y_1, y_2, \ldots, y_n)$, into a sequence of words. Here all the white space and punctuation have been removed from y, and we assume it consists of only the lower-case letters 'a' through 'z'. For example, $y = 'meetateight'$, and this could be parsed as the sequence of three words $s = (s_1, s_2, s_3) = ('meet', 'at', 'eight')$.

   In order to help with this task, suppose we function $\texttt{dict}(x)$ which returns an integer representing the overall quality of the string x as an English word. Here you can assume the quality of correct English words are high, and incorrect (or uncommon) strings such as 'etat' have low (or even negative) quality. We define the quality of the empty string to be arbitrarily low, for example, $\texttt{dict}('') = -\infty$.

   Moreover, we know a constant $L > 0$ such that for any string x of length bigger than L, $\texttt{dict}(x) = -\infty$. That is, we know any word with quality larger than $-\infty$ must have length less than or equal to L.

   We define a **parse** of an input string y as a sequence $S = (s_1, s_2, \ldots, s_K)$ of non-empty substrings of y, with each $s_k = y(i(k) \ldots j(k))$ with $i(k) \leq j(k)$. (Here the notation $y(i \ldots j)$ refers to the substring of y starting at the i^{th} character, y_i, and ending at the j^{th} character, y_j.) Moreover, for S to be a parse we require that the concatenation of the substrings in S equals the original string, i.e., $y = \texttt{concat}(s_1, s_2, \ldots, s_K) \equiv \texttt{concat}(S)$. It follows that

   $$1 = i(1) \leq j(1) < j(1) + 1 = i(2) \leq j(2) \ldots < j(K-1) + 1 = i(K) \leq j(K) = n.$$

   For the example $y = 'meetateight'$, a possible parse is $S = (s_1, s_2, s_3) = ('meet', 'at', 'eight')$ (that is, $s_1 = 'meet'$, and so on).

   The total quality of a parse $S = (s_1, s_2, \ldots, s_K)$ is defined to be

   $$Q(s) = \sum_{k=1}^{K} \texttt{dict}(s_j) \tag{1}$$

   The problem is then, given a string of letters y, find a parse S of y with the maximum total quality.

(a) Clearly describe a dynamic programming approach for solving this problem. You can assume that the given string y has length at least one. You do not need to provide a correctness proof, but explain why you believe your algorithm is correct.

(b) What is the order of the runtime of your algorithm in terms of the length n of the input string y? Here assume the that each call to the dictionary, $\texttt{dict}(x)$, runs in $O(1)$ time. Briefly explain your result (you do not need to do a detailed runtime analysis).

(c) **Optional (not marked).** Implement your algorithm in Python using a library to define the function $\texttt{dict}(s)$. For example, you could use package wordfreq 1.6.1 and define

$$\texttt{dict(s)} = \begin{cases} -1 & \text{if } \texttt{s} \text{ is empty,} \\ \texttt{math.floor}((1.0\texttt{e}+6) * \texttt{zipf\_frequency}(\texttt{s},'\texttt{en}')) & \text{if } 0 < |\texttt{s}| \leq L, \\ -\infty & \text{if } |\texttt{s}| > L, \end{cases}$$

For test strings you could take any English text, preprocess it to lower-case, removing all non-letters except blanks, and leave only one blank between words. Call this the "ground truth" string `g`. Form the test string `x` by removing blanks from `g`. Run your program on `x` and compare this with `g`. A greedy algorithm can then be used to output only the mismatched substrings of `x` with the corresponding parts of `g` which are considered the "correct" parse.

A similar problem is common in human speech understanding (that is, from audio) since people often blend words together during natural speech.

3. **[10pts] Optimal Parse Trees.** Let y be a string of n characters, say $y = (y_1, y_2, \ldots, y_n)$ We reuse the notation $y(i \ldots j)$ above, which denotes the substring $(y_i, y_{i+1}, \ldots, y_j)$ which has length $j - i + 1$. Suppose we are also given a sorted list $d = [d_0, d_1, d_2, \ldots, d_K]$ of string break points, with $1 = d_0 < d_1 < d_2 < \ldots < d_K = n + 1$. For example, these breakpoints could specify the endpoints of the words $s_k = y(d_{k-1} \ldots (d_k - 1))$, for $k = 1, 2 \ldots, K$, that were found in the previous problem.

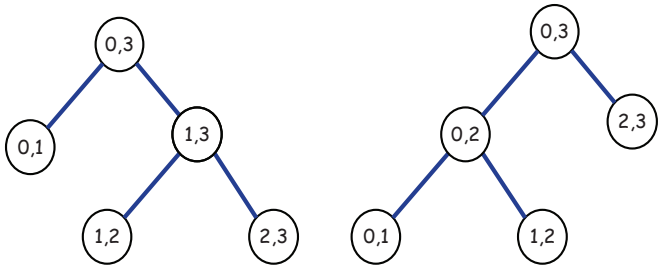
Here we are interested in the problem of computing an optimal parse tree for these words s_k. In general, this tree may assign words to phrases, phrases to sentence parts, and so on. For the purposes of this assignment we will simplify the form of the parse tree, but keep the essential elements required to illustrate a dynamic programming approach for computing an optimal parse tree.

We consider binary parse trees with each node v storing a data item, $v.data = (i, j)$, along with references to the left and right children, namely $v.left$ and $v.right$. These references to children may be null, i.e., $v.left = null$, representing that there is no such child. The data in each node v of the parse tree, say $v.data = (i, j)$, is a pair of integers i and j, with $0 \leq i < j \leq K$. These integers refer to the two string breakpoints d_i and d_j which together define the substring $y(d_i \ldots (d_j - 1))$.

Given the input string y and the sorted list of set of breakpoints $d = [d_0, d_1, d_2, \ldots, d_K]$, as described above, the binary tree T is said to be a **feasible parse tree** of y if and only if T is a binary tree (with nodes of the form v described above) which satisfies the following conditions:

- **Root node extends over y.** The root node r of T must have $r.data = (0, K)$. That is, the substring associated with r is $y(d_0 \ldots (d_K - 1)) = y(1 \ldots n) = y$.
- **Leaf nodes are words.** Every leaf node v of T (i.e., with $v.left = v.right = null$) must have data of the form $v.data = (k - 1, k)$ with $k = 1, 2, \ldots, K$. That is, leaf nodes are associated with single words $s_k = y(d_{k-1} \ldots (d_k - 1))$, as described above.
- **The children of non-leaf nodes correspond to left and right substrings.** For every non-leaf node v in the tree T, then both $v.left$ and $v.right$ are non-null. Moreover, for some integers i, k, j, with $0 \leq i < k < j \leq K$, $v.data = (i, j)$, $v.left.data = (i, k)$, and $v.right.data = (k, j)$. That is, the node v is associated with the substring $y(d_i \ldots (d_j - 1))$, and the left and right children are associated with the left and right parts of this string split at the intermediate breakpoint d_k. In particular, the left and right children are associated with the left and right substrings $y(d_i \ldots (d_k - 1))$ and $y(d_k \ldots (d_j - 1))$, respectively.

For a concrete example, suppose we are given a string y of length 50, and the sorted list of breakpoints $d = [d_0, d_1, d_2, d_3] = [1, 10, 30, 51]$. Then two feasible parse trees are shown for this problem in the figure below. The two integers in each tree node represent the data (i, j) for that node. These trees satisfy the conditions listed above, and are therefore feasible. Moreover, you can verify that these are the only two feasible parse trees for this problem (but don't hand that argument in).



We associate a cost with every feasible parse tree T, say $cost(T)$. The problem we wish to solve is then, given the input string y and the list of breakpoints d, find a minimal cost, feasible, parse tree.

Here we choose a simple form for this cost function. For any feasible parse tree T, define $cost(T)$ to be the sum of costs for all the nodes v in T, and define the cost of any node v to simply be $d_j - d_i$, where $v.data = (i, j)$. With this definition, the cost of any feasible parse tree T is simply the sum of the lengths of all the substrings represented by nodes in T. For example, for the parse trees above, the cost of the root node is $d_3 - d_0 = 51 - 1 = 50$, which equals the length of the input string y. The cost of the whole tree T_l on the left of this figure is (working from the root downwards) $cost(T_l) = 50 + 9 + 41 + 20 + 21 = 141$. Similarly, the cost of the tree on the right above is $cost(T_r) = 50 + 29 + 21 + 9 + 20 = 129$. Therefore, in this example, the right tree is the minimal cost tree.

	j=0	1	2	3
i=0		9	58	129
1			20	82
2				21
3				

Define $C(i, j)$ to be the minimum cost for any subtree rooted at a vertex v which has the data $v.data = (i, j)$. (The cost of any subtree is just the sum of the costs of every node in the subtree, as defined above.) Here i and j must satisfy $0 \leq i < j \leq K$. We can store these costs $C(i, j)$ in an $(K + 1) \times (K + 1)$ matrix, where we only need to consider the entries in this matrix for the column index j larger than the row index i. That is, we only need to consider the upper-right triangular portion of this cost matrix (see the above figure, which shows the cost table C for the previous example).

(a) Given a string y and a sorted list of break points $d = [d_0, \ldots, d_K]$, describe in detail a dynamic programming approach for computing this minimum-cost table C. Clearly explain why your approach is correct. (You do not need to provide a detailed proof of correctness.)

(b) What is the order of the runtime of your algorithm for computing this cost table C? Explain. (You need not provide a detailed proof of this runtime.)

(c) Explain in detail how a minimal cost, feasible, parse tree T can be computed for this problem given the cost table, C, computed in part (a). (You do not need to provide a detailed proof of the algorithm's correctness.)

4. **[10pts] Equal Thirds.** Given a list of $n > 0$ strictly positive integers, $X = (x_1, x_2, \ldots, x_n)$, we want to determine if it is possible to partition $I(n) = \{1, 2, \ldots, n\}$ into three mutually disjoint subsets, say $S_i \subset I(n)$ for $i = 1, 2, 3$, such that: a) $S_i \cap S_j = \emptyset$ for $i \neq j$; b) $I(n) = \cup_{i=1}^{3} S_i$; and

$$\sum_{k \in S_1} x_k = \sum_{k \in S_2} x_k = \sum_{k \in S_3} x_k = \left[\sum_{k \in I(n)} x_k \right] /3. \tag{2}$$

Moreover, if such a partitioning is possible, we wish to find suitable subsets S_i for $i = 1, 2, 3$.

For example, given $X = (1, 2, 3, 4, 4, 5, 8)$ we find that the answer is **yes** it is possible. Specifically, a suitable partition of $I(|X|)$ is $S_1 = \{1, 7\}$, $S_2 = \{5, 6\}$, and $S_3 = \{2, 3, 4\}$. Moreover, in this case, the corresponding sublists $X_i = (x_k \mid k \in S_i)$ are $X_1 = (1, 8)$, $X_2 = (4, 5)$ and $X_3 = (2, 3, 4)$. Note that these sublists X_i all sum to 9, which is one third of the sum over all X. Therefore equation (2) is satisfied.

Alternatively, if we were given $X = (2, 2, 3, 5)$, then $\sum_i x_i = 12$, but there is no way to partition this into three sublists which all sum to $12/3 = 4$. In this case the answer is **no**, it is not possible to find such a solution.

(a) Clearly describe a dynamic programming algorithm for returning "yes" or "no"', corresponding to whether or not such a partitioning is possible. Your algorithm must run in time $O(n\,[\sum_{i=1}^{n} x_i]^2)$. Explain why you believe the algorithm and your runtime estimate are correct, but you do not need to provide detailed proofs.

(b) Assuming your algorithm runs in time $\Omega(n\,[\sum_{i=1}^{n} x_i]^2)$, is this considered to be a polynomial time algorithm? Briefly explain.

(c) In situations where part (a) returns "yes", that is, when a solution exists, clearly explain how to compute suitable partition sets, namely S_i, $i = 1, 2, 3$. (You should build on your solution in part (a).)