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

Solutions for Assignment 2: Dynamic Programming

---

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.)

Soln Q1, Part 1: Use string matching algorithm from lecture notes. Consider the algorithm described on p.28 of `dynamicProgIntro.pdf`). Here we wish to use x as the original string, and y = reverse(x), which is just the left-right reversal of x. So both x and y have length n. Recall, this algorithm takes $O(n^2)$ time and space.

Soln, Q1, Part 2: Choose appropriate $\alpha_{p,q}$ and δ. We use $\alpha_{p,p} = 0$ for all characters p, $\alpha_{p,q} = 3$ for $p \neq q$, and $\delta = 1$. What's important here is that $\alpha_{p,p} < 2\delta < \alpha_{p,q}$.

Soln, Q1, Part 3: Characterization of optimal paths. With these values of $\alpha_{p,q}$ and δ, it follows that any diagonal edge in the matching graph that corresponds to mismatched items (i.e., costing $\alpha_{p,q} = 3$ since $p \neq q$) can always be replaced by two unmatched edges (e.g., p is unmatched in x and then q is unmatched in y), which together cost less (i.e, $2\delta < \alpha_{p,q} = 3$). Therefore, no optimal path (i.e., with the minimum matching cost) will use a diagonal edge with $p \neq q$).

Soln Q1, Part 4: Derive M as a function of $OPT(n, n)$. Let P be any optimal path in this string matching problem which has the minimum cost $OPT(n, n)$. From the previous paragraph, we know that any path with this optimal match value must use only the edges costing δ or $\alpha_{p,p} = 0$.

Let M be the number of times a diagonal edge costing $\alpha_{p,p} = 0$ is used on this path P. Then M characters in each string are paired with identical characters in the other string, while $n-M$ characters are mismatched in each string. As a result, we have

$$OPT(n, n) = 2(n - M)\delta + M\alpha_{p,p} = 2(n - M), \text{ since } \delta = 1 \text{ and } \alpha_{p,p} = 0. \tag{1}$$

Moreover, since P was assumed to have minimum possible cost, it follows that M is the maximum number of characters that can be matched. That is,

$$M = n - OPT(n, n)/(2\delta) = n - OPT(n, n)/2. \tag{2}$$

is the maximum length of any palindromic subsequence.

Therefore the maximum length M can be computed by first computing $OPT(n, n)$ using the algorithm described in the lecture notes (taking $O(n^2)$ time). Then M is given by (2), taking $O(1)$ more time. So the overall algorithm runs in $O(n^2)$ time.

Soln Q1, Part 5: Extract a longest palindromic subsequence. Optional and omitted. This can be done by working backwards through the table $OPT(i, j)$, and requires $O(n)$ time.

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 ith character, y_i, and ending at the jth 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

$$Q(s) = \sum_{k=1}^{K} \texttt{dict}(s_j) \tag{3}$$

   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.

   Soln 2a, Part 1: Explanation of cases and recurrence relation. Suppose we are given the string $(y(1), \ldots, y(n))$ with $n > 0$, and the function dict(s). For $j \in \{1, 2, \ldots, n\}$ define $OPT(j)$ to the optimum quality of possible parses for the prefix string $y(1 \ldots j) = (y(1), \ldots, y(j))$. Let S_j denote such a parse of quality $OPT(j)$. We wish to derive a recurrence relation for $OPT(j)$ in terms of the optimum values of parses of shorter prefix strings. It is convenient to define $OPT(0) = 0$.

   For each j with $1 \leq j \leq n$ there are two cases to consider. Either the optimal parse S_j consists of more than one word, or exactly one word. We consider these two cases below.

   Single Word Case: Here $S_j = (s_1)$, with $s_1 = y(1 \ldots j)$. So $OPT(j) = dict(y(1 \ldots j))$. Note that, since $dict(y(1 \ldots j)) = -\infty$ for $j > L$ this case is only relevant for $j \leq L$.

   Multiple Word Case: Here S_j must have the form $S_j = (s_1, \ldots, s_m)$, where $m > 1$ and the last word s_m satisfies $s_m = y(i \ldots j)$ for some i with $\max(1, j - L) < i \leq j$. In order for S_j to be an optimal parse which has the last word s_m it must be the case that $(s_1, \ldots s_{m-1})$ is an optimal parse of $y(1 \ldots (i-1))$. The optimum value of such a parse is defined to be $OPT(i-1)$. Hence, in this case we could achieve the score $OPT(i-1) + dict(y(i \ldots j))$. Moreover, in order to maximize the quality, we wish to choose $i \in \{j - L + 1, j\}$ in such a way to maximize these scores over these possible choices for i.

   Combining these two cases, we have

$$OPT(j) = \max_{\max(1, j-L+1) \leq i \leq j} \left[OPT(i-1) + \text{dict}(y(i \ldots j)) \right]. \tag{4}$$

Note that we use the definition $OPT(0) = 0$ when $i = 1$ in the above expression.

Soln 2a, Part 2: Correctness. The recurrence relation (4) is correct since it accounts for all possibilities that don't result in a score of $-\infty$. This follows from Q2a Part 1 above (details omitted).

Soln 2a, Part 3: Pseudo Code (Optional)

```
[q, p] = optSegQual(y, n)
    // Input: y = y(1...n)) the string to be parsed.
    // n − the length of the string (Precondition: n > 0)
    // Output: q an array of size n, with q(j) the maximum quality of any segmentation of y(1..j)
    // Output: p an array of size n, such that the optimal parse of the prefix string y(1...j) is
    // S_j = (s_1, ..., s_m) with the last word s_m = y((p(j)), ... j).
    // That is, when p(j) = 1 the prefix string y(1...j) is parsed as a single word.
    L = 45 // Set L to be the longest word in a major English dictionary.
    Allocate q, an array of length n. // q(j) will store OPT(j).
    Allocate p, an array of length n. // p(j) as above.

    for j = 1..n        // Loop over increasing lengths of the prefix string
        // Find the best quality of multi-word segmentations of y(1..j)
        for i = max(1, j-L+1)..j   // Check the case that the last word is y(i..j)
            if i == 1       // Single word segmentation
                q(j) = dict(y(1..j)
                p(j) = 1
            else            // Multiple word segmentation of y(1..j)
                // Find quality of best segmentation of y(1..j) with last word being y(i..j)
                tmpQ = q(i-1) + dict(y(i..j))
                if q(j) < tmpQ      // Update current max quality.
                    q(j) = tmpQ
                    p(j) = i        // Remember the start index of the best last word.
                end
            end
        end    // End loop over beginning index, i, of last word in y(1..j).
    end
    return q, p
```

Soln 2a, Part 4: Extracting the parse. If you only returned the table of optimum values of the parse, i.e., $OPT(j)$ (which is the $q(j)$ returned by the pseudo code above), then you need to use the table $OPT(\cdot)$ to find the beginning of each word, working backwards from the last word back to the first. That is, if you currently know the last word ends at j (initially $j = n$), then look backwards through i, where $\max(1, j - L + 1) \leq i \leq j$, for where $OPT(j) = OPT(i - 1) + dict(s(i..j))$. The first such i determines the start position for this last word, i.e., the last word is $y(i..j)$. (There must be such an i, by construction.) Iterate this until $i = 1$. We omit the details.

Instead, in the above pseudo code we have returned enough information to find an optimal sequence of words more directly, as follows:

```
\\ Given n and p(k) for 1 ≤ k ≤ n, as defined above
S = [ ]
k = n
while k ≠ 0
    S = prepend(y(p(k)...k), S)
    k = p(k)-1
end
```

(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).

Soln 2b, Part 1: The inner-most loop in optSegQual. The inner-most loop of `optSeqQuality` is executed at most L times and, by assumption `dict(s)` is O(1). The only other step in the inner-most loop is to extract the substring $y(i..j)$, and since we are only looking at substrings of length at most L, this is at most $O(L)$. Since L is a fixed constant, the runtime of this inner-most loop is $O(1)$.

Soln 2b, Part 2: Runtime of optSegQual. The inner-most loop is run n times, so this gives an overall runtime of $O(n)$.

Soln 2b, Part 3: Runtime for construction of optimal segmentation. The post-processing step involves copying disjoint substrings y, of total length n, so all this copying alone will cost $O(n)$. The loop for the segmentation construction runs M times, where M is the number of words in the optimal parse. Since each word has length at least one, we have $M \leq n$, and therefore the rest of the algorithm (not counting the string copying) also runs in $O(n)$ time. Therefore the segmentation construction step also runs in $O(n)$ time.

We could get the same order runtime, $O(n)$, if we did not return the array p containing the first character of words. Although this approach would reguire $O(LM)$ calls to the dictionary, and $O(LM)$ string copying costs, where $M \leq n$ is the number of words in the optimal parse.

(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

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

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.

Soln 2c. See Piazza note 130 for some intuition for the appropriate choice of `dict(s)` along with several refinements of the above choice.

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.

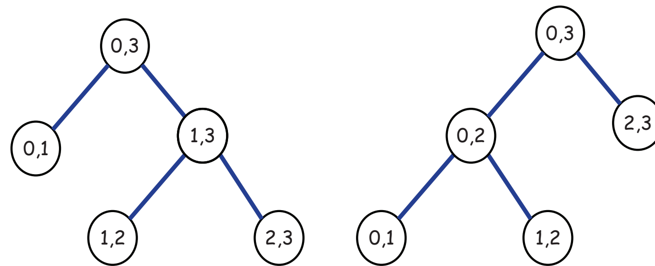
   Here we are interested in the problem of computing an optimal parse tree for these words s_k. In general, this tree may assign words to phrases, phrases to sentence parts, and so on. For the purposes of this assignment we will simplify the form of the parse tree, but keep the essential elements required 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 $0 \le i < j \le K$. 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 $0 \le i < k < j \le K$, $v.data = (i, j)$, $v.left.data = (i, k)$, 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 $0 \le i < j \le K$. 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.)

Soln 3a, Part 1: Recurrence Relation. Consider a subtree with the root (i, j) corresponding to the substring $y(d_i \ldots (d_j - 1))$. Here $0 \le i < j \le K$. The optimal cost of any subtree rooted at (i, j) is defined to be $C(i, j)$.

There are two cases.

Leaf Case. In the first case we have $j = i + 1$ and this root node (i, j) is a leaf. The optimum value of a leaf is defined to be $C(i, i + 1) = d_{i+1} - d_i$.

Non-Leaf Case. The other case is $j > i + 1$. By the definition of the form of the parse tree, this root node (i, j) has two children, (i, k) and (k, j) for some k with $i < k < j$.

In this second case, the cost for the parse tree is the cost of the root node (i, j), which is $d_j - d_i$ plus the costs for left and right subtrees rooted at, say, (i, k) and (k, j) for some k with $i < k < j$. In order for our parse tree to be optimal, we should choose the optimal values for the left and right subtrees. Thus, by the definition of C, the minimum costs of theses left and right subtrees are $C(i, k)$ and $C(k, j)$. Given these facts, the optimum cost of a subtree rooted at (i, j) with $j > i + 1$ is $C(i, j) = (d_j - d_i) + \min_{i<k<j}(C(i, k) + C(k, j))$.

As a consequence, we can compute $C(i, j)$ with $1 \le i < j \le K$ as

$$C(i, j) = \begin{cases} d_j - d_i, & \text{for } j = i + 1, \\ d_j - d_i + \min_{i<k<j}(C(i, k) + C(k, j)), & \text{for } j > i + 1. \end{cases} \tag{5}$$

In terms of the cost matrix $C(i, j)$ this computation can be done by first computing all the elements on the second-diagonal of this matrix, i.e., with $j = i + 1$. Then computing all the elements on the third-diagonal, $j = i + 2$, and so on.

Soln 3a, Part 2: Correctness. The recurrence relation (5) is correct since it accounts for all possibilities for any vertex in the parse tree, which is either a leaf or a vertex with two children. Moreover, when the vertex has two children, (5) considers all possibilities for its two children. These results follow from the argument in Q3a Part 1 above (details omitted).

(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.)

Soln 3b, Part 1. Each table entry. To compute one entry in the table, say $C(i, j)$, we may need to check $O(K)$ possible costs (i.e., the number of individual k's we are minimizing over in eqn (5)). Each of these k cost computations is requires $O(1)$ for the look-up of $C(i, k)$ and $C(k, j)$ from previously computed results. Therefore each table entry $C(i, j)$ can be computed in $O(K)$.

Soln 3b, Part 2. Runtime to compute table. Since there are $O(K^2)$ table entries, the overall runtime is $O(K^3)$.

(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.)

Soln 3c, Constructing the best parse tree. The root of the tree corresponds to $(0, K)$ with a cost of $C(0, K)$. We recursively compute the left and right subtrees of any node (i, j) as follows. If $j = i + 1$, then the node is a leaf. Otherwise $j > i + 1$ and, following equation (5), we need to find any k such that $k = \text{argmin}_{i<k<j}(C(i, k) + C(k, j))$. Given such a k, the left and right children of this node are (i, k) and (k, j).

Alternatively you might have kept a table, say $p(i, j)$, in part (a) for which the optimal next breakpoint for this table entry is $k = p(i, j)$, where k is as described in the previous paragraph.

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

$$\sum_{k \in S_1} x_k = \sum_{k \in S_2} x_k = \sum_{k \in S_3} x_k = \left[\sum_{k \in I(n)} x_k \right] /3. \tag{6}$$

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 (6) 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 $O(n \left[\sum_{i=1}^{n} x_i\right]^2)$. Explain why you believe the algorithm and your runtime estimate are correct, but you do not need to provide detailed proofs.

Soln 4a, Part 0. Ruling some things out. Note we cannot simply run Knapsack (with values equal weights) to find one subset that sums exactly to $A = \left[\sum_{1 \leq j \leq n} x_j\right] /3$, greedily remove those elements, and try again. For example, consider the case (provided to me by a CSC373 student)

$$X = (2, 2, 2, 3, 3, 4, 5, 6, 6).$$

This sums to 33, so $A = 11$. Note that, you could first choose the subsequence $(3, 3, 5)$, and you are then stuck (since you've used up all the odd elements of X). Or you could choose $(2, 2, 2, 5)$ as the first set, and you are again stuck (since there is no way to choose a subset of the remaining terms, namely $(3, 3, 4, 6, 6)$, to sum to 11). BTW, there is a solution, namely $S_1 = (2, 4, 5)$, and $S_2 = S_3 = (2, 3, 6)$.

You might expect such an approach to have serious problems in general since the number of subsets that sum to $A = \left[\sum_{1 \leq j \leq n} x_j\right] /3$ can be exponentially large (in terms of n). Successfully keeping track of all of these cases will end up with something similar to the solution presented next.

Soln 4a, Part 1. Sub-problems. Suppose we define the assignment function $\sigma(k) \in \{1, 2, 3\}$, for $1 \leq k \leq n$, which denotes that element x_k is to be assigned to sublist $\sigma(k)$. Then, given any assignment σ, we consider the sums of just the first i elements of x, namely

$$T_k(i, \sigma) = \sum_{\{j \ \mid \ 1 \leq j \leq i, \sigma(j) = k\}} x_j, \tag{7}$$

for $k = 1, 2, 3$. We then seek an assignment σ such that

$$A \equiv \left[\sum_{1 \leq j \leq n} x_j \right] /3 = T_k(n, \sigma), \text{ for } k = 1, 2, 3. \tag{8}$$

Soln 4a, Part 2. Two sums are enough. Note that since every element of X is assigned to one of the three sets, we have $\sum_{k=1}^{3} T_k(i, \sigma) = \sum_{j=1}^{i} x_j$, where the latter is just the sum of the first i values of X. Therefore,

$$T_3(i, \sigma) \ = \ \sum_{j=1}^{i} x_j - T_1(i, \sigma) - T_2(i, \sigma), \tag{9}$$

and so if we know T_1 and T_2 we can use (9) to work out T_3.

Soln 4a, Part 3. The possible states for prefix sets of length i. We are going to need to represent many possible values for the pair $(T_1(i,\sigma), T_2(i,\sigma))$ (i.e., for any feasible assignment σ). This set of pairs grows like $O(3^i)$ as i increases, since there are three choices for each $\sigma(i)$. Specifically, for a given assignment σ we have

$$\begin{pmatrix} T_1(i,\sigma) \\ T_2(i,\sigma) \end{pmatrix} = \begin{pmatrix} T_1(i-1,\sigma) \\ T_2(i-1,\sigma) \end{pmatrix} + \begin{pmatrix} x_i\delta(\sigma(i) == 1) \\ x_i\delta(\sigma(i) == 2) \end{pmatrix}. \quad (10)$$

where $\delta(true) = 1$ and $\delta(false) = 0$. Here we can take i to be any integer between 1 and n, and we define $T_k(0,\sigma) = 0$.

Soln 4a, Part 4. The relevant states for prefix sets of length i. For any given σ, note that $T_k(i,\sigma)$ is a non-decreasing function of i (since all the x_i are positive, see (10)). In particular, we do not need to consider any σ if we find an i such that $T_k(i,\sigma) > A$ (where A is as in (8)).

We therefore need to remember any (integer-valued) state $(T_1(i,\sigma), T_2(i,\sigma)) \in [0, A] \times [0, A]$.

(Note: An alternative is to represent the remaining sum instead, so $(A, A) - (T_1(i,\sigma), T_2(i,\sigma))$, which is also in $[0, A] \times [0, A]$, but here the goal state would be $(0, 0)$ instead of (A, A).)

Soln 4a, Part 5. A representation for the relevant states. There are many alternatives for representing the states. For example, we could use a hash table for each i, where the keys are the distinct integers $(A+1)*T_1(i,\sigma)+T_2(i,\sigma)$. That is, the ith hash table has this key iff the corresponding pair (T_1, T_2) can be formed by using only the first i elements of X for some choice of σ.

A more direct representation is to use a binary matrix $M_i(a, b)$, with $0 \leq a, b \leq A$. Here we choose

$$M_i(a, b) \text{ is true iff, for some assignment } \sigma,\ (T_1(i,\sigma), T_2(i,\sigma)) = (a, b). \quad (11)$$

Soln 4a, Part 6. Basic dynamic programming alg. For $i = 1$ we can assign x_1 to any one of the three sets. WLOG we could break some of the symmetry here and choose to assign x_1 to set 3 (i.e., $\sigma(1) = 3$). For this choice we have $M_1(a, b) = false$ for all (a, b) except $a = b = 0$, for which $M_1(0, 0) = true$.

The following algorithm then implements the recurrence relation in equation (10). The loop invariant is as in (11). It follows that the solution of the original problem is now simply given by $M_n(A, A)$. (The correctness of this algorithm follows from the preceeding argument.)

```
#00  [b, M] = threeSum(X)
#01      //Output: b = true iff the three sum problem has a solution for X.
#02      // M = (M_1, M_2, ..., M_n) are as in equation (11).
#02      Set n to be length of X, and denote the i^th element as x_i.
#04      Set all the elements in M_1 to be false, except M_1(0,0), which is true.
#05      For each i ∈ {2, ..., n}
#06          Initialize M_i = M_{i-1}      // Here we are using σ(i) = 3 in equation (10) above.
#07          Find each element (a,b) s.t. M_{i-1}(a,b) is true.
#08              // Apply the terms for σ(i) = 1 and σ(i) = 2 in eqn (10) above.
#09              If a + x_i ≤ A
#10                  Set M_i(a + x_i, b) = true
#11              If b + x_i ≤ A
#12                  Set M_i(a, b + x_i) = true
#13      return M_n(A,A) and list of M_i's // Note, a solution is possible iff M_n(A,A) is true.
```

Soln 4a, Part 7. Running time. Each of the matrices $M_i(a, b)$ is $(A + 1) \times (A + 1)$. Therefore

loop starting on line #07 could be executed $O(A^2)$ times, with the loop body executing in $O(1)$ time. Similarly the matrix copy on line #06 takes $O(A^2)$ time. Therefore, the loop starting on line #05 executes $n-1$ times and takes $O(A^2)$ time each iteration, for a total runtime of $O(nA^2)$. Line #04 takes no more than $O(A^2)$. Therefore the runtime of this algorithm is $O(nA^2)$ where A is as in equation (8).

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

Soln 4b, Part 1. No, not polytime. (Nor polyspace.)

Soln 4b, Part 2. Explanation. Suppose we consider problems where each integer x_i is represented in B bits, so the input size of X can be taken as nB bits. We are going to consider the limit at B goes to infinity. For input problems requiring nB bits, the quantity $\left[\sum_{k=1}^{3} x_k\right]$ could be as large as, say, any constant fraction $f < 1$ of $n(2^B - 1)$. That is, in the worst case, $\left[\sum_{i=1}^{n} x_i\right]^2 = \Omega(n^2 2^{2B})$. But this is not $O((nB)^q)$ as $B \to \infty$ for any constant $q > 0$ (see solved tutorial exercise Q3 for Feb 5). Therefore $\Omega(n^2 2^{2B})$ is exponential in the size of the input, nB, so this is not a polynomial time algorithm.

(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).)

Soln 4c. Part 1. How to work backwards. For the matricies $M_i(a, b)$ generated in (4a) Part 6, we know from (11) that $M_i(a, b)$ is true iff, for some assignment σ, $(T_1(i, \sigma), T_2(i, \sigma)) = (a, b)$. Also note that, by definition of $(T_1(i, \sigma), T_2(i, \sigma))$, these sums do not depend on $\sigma(k)$ for $k > i$. Therefore, we can work backwards from $i = n$, keeping track of a pair (a, b) for which there exists an assignment such that $(T_1(i, \sigma), T_2(i, \sigma)) = (a, b)$. We can starting this process at $(a, b) = (A, A)$, and $i = n$. In this way we can reveal $\sigma(k)$ for $k = n, n-1, \ldots, 1$.

For example, starting with $(a, b) = (A, A)$ and taking one step back from n,

- If $M_{n-1}(A, A)$ is true then we know there exists a solution with $\sigma(n) = 3$. Moreover, from (11) we know there is a solution to the subproblem $(T_1(n-1, \sigma), T_2(n-1, \sigma)) = (A, A)$. In this case we can keep $(a, b) = (A, A)$ for the next step.
- Otherwise, if $M_{n-1}(A, A - x_n)$ is true (being careful not to evaluate a negative index), then there must be a solution with $\sigma(n) = 2$ (see eqn (10)). Moreover, since $M_{n-1}(A, A - x_n)$ is true, we know from (11) there is a solution to the subproblem $(T_1(n-1, \sigma), T_2(n-1, \sigma)) = (A, A - x_n)$. In this case we can reset $(a, b) = (A, A - x_n)$ and continue.
- Otherwise, the only other option is $\sigma(n) = 1$ and we therefore know there must be a solution to the subproblem $(T_1(n-1, \sigma), T_2(n-1, \sigma)) = (A - x_n, A)$ (again, see eqn (10)). That is, $M_{n-1}(A - x_n, A)$ must be true. In this case we can reset $(a, b) = (A - x_n, A)$ and continue.

We can continue in this way to iterate this back to $i = 1$. Recall we set $\sigma(1) = 3$ in the first paragraph of Q4a Part 5. Indeed the only index at which $M_1(a, b)$ is true is $(a, b) = (0, 0)$.

Soln 4c, Part 2. (Optional) This motivates the following algorithm:

```
#00  [σ] = unpackThreeSum(M)
#01      // Given the list M_i(a,b), i = 1,...,n of matrices computed in part (a),
#02      // where M_n(A,A) is true, return a valid assignment σ.
#02      Allocate σ to be an array of length n (indexing of σ is 1-based).
#03      (a,b) = (A,A) // With A the max row/col index of M_i, see eqn (8).
#04      for i ∈ {n-1,...,1}
```

```
#05        Invariant: There exists a σ such that (T_1(i+1, σ), T_2(i+1, σ)) = (a, b).
#06        if M_i(a, b)
#07            σ(i+1) = 3
#08        elif (b − x_{i+1} ≥ 0) and M_i(a, b − x_{i+1}) // short-circuit "and" evaluation
#09            σ(i+1) = 2
#10            b = b − x_{i+1}
#11        else // a definition of confidence...
#12            σ(i+1) = 1
#13            a = a − x_{i+1}
#14    σ(1) = 3 // See first paragraph in (Q4a, Part 6) above.
#15    Assert (a, b) == (0, 0)
#16    return σ
```