Recall that a dynamic programming algorithm to solve a given problem $P$ involves the following elements:

(a) A definition of a polynomial number of subproblems that will be solved (and from whose solution we will compute the solution to $P$ — see (b) below).

(b) A recursive formula to compute the solution to each subproblem from the solutions to smaller subproblems. This induces a partial order on the subproblems defined in (a).

(c) A way to compute the solution to $P$ from the solutions to the subproblems computed in (b).

Proving the correctness of a dynamic programming algorithm amounts to justifying (i) why the recursive formula in step (b) correctly computes the subproblems defined in step (a), and (ii) why the computation in (c) yields a solution to the given problem. Part (ii) is often immediate from the definition of the subproblems.

Question 1. (20 marks) Let $a_1 b_1 a_2 b_2 a_3 \ldots a_{n-1} b_{n-1} a_n$ be a sequence where each $a_i$, $1 \leq i \leq n$, is an integer, and each $b_i$, $1 \leq i \leq n - 1$, is either $+$ or $-$. Thus, this sequence represents an unparenthesized
arithmetic expression such as $3 - 5 - 7 + 9$. The actual value of such an expression depends on the order in which the operations $+$ and $-$ are applied, i.e., on how the expression is parenthesized. For instance the above example can be parenthesized in five ways (check them out!), yielding four different values.

Give a **polynomial-time dynamic programming algorithm** that finds the **largest possible value** of a given unparenthesized expression, over all possible orders of performing the operations — i.e., over all possible parenthesizations. For instance in the above example your algorithm should return 14. More precisely, your algorithm should solve the following problem:

**Input:** Sequences $a_1, \ldots, a_n$ and $b_1, \ldots, b_{n-1}$, where $a_i$ is a positive integer for each $1 \leq i \leq n$ and $b_i \in \{+, -\}$ for each $1 \leq i \leq n - 1$.

**Output:** The maximum value that can be achieved by parenthesizing the expression $a_1 b_1 a_2 \ldots a_{n-1} b_{n-1} a_n$.

Justify the correctness of your algorithm and analyze its running time.

**Hint.** A parenthesization of the expression can be viewed as a tree, where the leaves are the $a_i$’s (integers) and the internal nodes are the $b_i$’s ($+$ or $-$).

**Question 2.** (20 marks) [Honour among thieves.] Two thieves break into a store and steal $n$ items, labeled 1 to $n$. Item $i$ has value $v_i \in \mathbb{N}$. The two thieves want to divide up the stolen goods so that each of them receives exactly half of the items and these items account for exactly half of the total value of the stolen goods $V = \sum_{i=1}^{n} v_i$. Give a dynamic programming algorithm that takes as input the sequence $v_1, \ldots, v_n$, and determines if such a division is possible. The running time of your algorithm should be polynomial in $n$ and $V$.

**Note:** Such an algorithm is not a polynomial-time one; make sure you understand why. If you do come up with a polynomial-time algorithm for this problem, don’t keep it secret: You may have just won the one-million dollar (US) [Clay Institute prize](#) for settling the P vs. NP question! (Those of you who have taken CSCC63 should be able to see an easy polynomial-time reduction of the Partition problem to this problem.)

**Question 3.** (15 marks) Let $G = (V, E)$ be a directed graph and $\text{wt} : E \to \mathbb{Z}^+$ be an edge-weight function (note that edges have positive weights). The **width** of an $i \to j$ path in $G$ is the **minimum** weight of any edge on that path; $+\infty$ if $i = j$ (i.e., there is no edge on that path; and 0 if there is no $i \to j$ path). If we think of the weight of an edge as its width, the width of a path is the width of the narrowest edge, i.e., the “bottleneck” of the path.

Modify the Floyd-Warshall algorithm to obtain an $O(n^3)$ algorithm that, given $G$ and $\text{wt}$, returns a pair $(W, \text{pre})$ of two-dimensional arrays, where $W[u, v]$ is the maximum width of a $u \to v$ path (0 if no such path exists) and $\text{pre}[u, v]$ is the predecessor of $v$ on a maximum width $u \to v$ path ($\bot$ if no such path exists). Justify the correctness of your algorithm. (You don’t need to analyze its running time, but it should be $O(n^3)$.)