(a) In one sentence, describe what MERGESORT(A[1 .. n]) does to A if we remove the last line of
the algorithm so that we never call MERGE.

Solution: It does nothing; all steps that modify A are hidden in the MERGE procedure! ■

(b) Give a recurrence for the running time of STOOGESORT(A[1 .. n]). [Hint: Feel free to ignore
the ceiling.]

Solution:

\[ T(n) \leq 3T(2n/3) + O(1) \]

■

Explanation: We do three recursive calls on subproblems of size approximately 2n/3. All
other work takes constant time.

(c) What is the running time of STOOGESORT(A[1 .. n]) obtained by solving your recurrence?

Solution:

\[ T(n) = O(n^{\log_{3/2} 3}) \]

■

Explanation: The second case of the master theorem applies with \( a = 3, b = 3/2, d = 0. \)
Recall the BACKPACK problem described in the problem sheets. Let \( T = \sum_{i=1}^{n} C[i] \) be the total cost of all given books. Given integers \( i \) and \( c \) with \( 0 \leq i \leq n \) and \( 0 \leq c \leq T \), let \( \text{MinWeight}(i, c) \) denote the minimum total weight of any subset of books \( 1 \) through \( i \) that have total cost exactly \( c \). Let \( \text{MinWeight}(i, c) = \infty \) if there is no such subset of books.

(a) Give a recurrence definition for \( \text{MinWeight}(i, c) \). Don’t forget the base cases!

\[
\text{MinWeight}(i, c) = \begin{cases} 
0 & \text{if } c = 0 \\
\infty & \text{if } i = 0 \text{ and } c > 0 \\
\text{MinWeight}(i-1, c) & \text{if } i > 0 \text{ and } C[i] > c \\
\min\{\text{MinWeight}(i-1, c), \\
W[i] + \text{MinWeight}(i-1, c - C[i])\} & \text{otherwise}
\end{cases}
\]

**Explanation:** We can easily get a total cost of 0 by taking no books. If \( i = 0 \) and \( c > 0 \), then there are no books available to make a subset with positive cost, so \( \text{MinWeight}(i, c) = \infty \) by definition. If \( i > 0 \) but \( C[i] > c \), then we cannot include book \( i \) in our subset without overshooting our cost goal. We must instead work only with books \( 1 \) through \( i - 1 \) to reach total cost \( c \). Finally, if \( i > 0 \) and \( C[i] \leq c \), then we can choose whether or not we want to include book \( i \) in our subset. If we don’t, we still need to find a min weight subset from the remaining books \( 1 \) through \( i - 1 \) that have total cost \( c \). If we do include \( i \), then we need to count its weight \( W[i] \) and then try to find a min weight subset from the remaining books that has a cost \( c - C[i] \) so that book \( i \) and that subset together cost exactly \( c \).

(b) Design and analyze a dynamic programming algorithm to build a two-dimensional array \( \text{MinWeight}[0..n][0..T] \) where for each pair of indices \( i \) and \( c \), \( \text{MinWeight}[i][c] = \text{MinWeight}(i, c) \) as defined above. Your algorithm should have a running time of \( O(nT) \).

**Solution:** Each entry \( \text{MinWeight}[\cdot][0] \) is set to 0 as a base case. Similarly, each entry \( \text{MinWeight}[0][\cdot] \) where \( c > 0 \) is set to \( \infty \). Every other entry depends upon one or two entries to the left (with lower \( i \) index), so we fill the array in row-major order (low to high \( i \) index in the outer loop, low to high \( c \) index in the inner loop). There are \( O(nT) \) entries to fill in constant time each, so the running time is \( O(nT) \).
(c) Design and analyze an algorithm for the BACKPACK problem that runs in $O(nT)$ time. You may assume your solution to part (b) is correct and use it as a black box if you would like.

**Solution:** We build the array $MinWeight[0..n][0..T]$ as defined above in $O(nT)$ time using the algorithm from part (b). Then we search all entries $MinWeight[n][\cdot]$ in $O(n)$ time to find the largest $c$ for which $MinWeight[n][c] \leq M$. We return this value of $c$. By definition of $MinWeight$, we cannot find a larger cost subsets of books with smaller total weight than $M$. The total running time is $O(nT)$. ■
For each of the following alternative greedy algorithms for the class scheduling problem, write *Correct* if the algorithm always constructs an optimal schedule or write *Wrong* if there is some input for which the algorithm does not produce an optimal schedule.

1. Choose the course $x$ that ends last, discard classes that conflict with $x$, and recurse.

   **Solution:** *Wrong*

   **Explanation:** For a counterexample, consider one long class beginning before at all others and ending after all others. Let the others be disjoint. This greedy algorithm will take the one long course when the optimal schedule will take all the other courses.

2. Choose the course $x$ that starts last, discard all classes that conflict with $x$, and recurse.

   **Solution:** *Correct*

   **Explanation:** This is the lecture algorithm with start and finish flipped. Alternatively, given an optimal schedule $X$, replace its latest starting course with $x$. You’ll create no new conflicts, and induction guarantees recursing is correct.

3. Choose the course $x$ with shortest duration, discard all classes that conflict with $x$, and recurse.

   **Solution:** *Wrong*

   **Explanation:** Consider two long courses that don’t overlap but with a very small time gap between them. Finally, add a short third course that spans the gap. This greedy algorithm will take only that short course instead of the two long ones.
4. If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

**Solution:** *Wrong*

**Explanation:** Use the same counterexample from part (a).

5. If any course $x$ completely contains another course, discard $x$ and recurse. Otherwise, choose the course $y$ that *ends last*, discard all classes that conflict with $y$, and recurse.

**Solution:** *Correct*

**Explanation:** If any optimal schedule contains a course $x$ containing another course, we can replace $x$ with the course it contains and still have a good schedule of the same size. Now, suppose we've removed all the courses that completely contain other courses, and let $y$ be the course that ends last. Course $y$ must also start last; otherwise, it would completely contain the course that does start last. Once we're looking for courses that start last, we're running the algorithm from part (b).
Clearly indicate (draw) the following structures from the weighted graph pictured below. *This question has five parts.*

**Solution:**

1. A depth-first spanning tree rooted at $s$

2. A breadth-first spanning tree rooted at $s$
3. A shortest-path tree rooted at $s$

4. A minimum spanning tree

5. A minimum $(s,t)$-cut
Consider the solitaire game described in the problem sheets.

1. Prove that the obvious greedy strategy (always choose the smallest number) does not give the largest possible number of moves for every instance of the puzzle. Your proof should be a small counterexample and a description of the optimal sequence of moves that does better than the greedy strategy.

**Solution:** Consider the puzzle below:

![Puzzle Image]

The greedy strategy moves 1 step from the first block, and then is forced to move three steps and then one for three moves total. The optimal strategy is to move 2 steps from the first block and then take three more one step moves for four total.

2. Describe and analyze an efficient algorithm to find the largest possible number of legal moves for a given instance of the puzzle. Your algorithm should ideally run in $O(n)$ time.

**Solution:** Let $MaxMoves(i)$ denote the maximum number of moves we can make starting at the $i$th square from the left; if $i > n$, then let $MaxMoves(i) = 0$. We want to compute and return $MaxMoves(1)$. Let square $i$ contain the numbers $Square[i][1..4]$. Then,

$$MaxMoves(i) = \begin{cases} 
0 & \text{if } i > n \\
\max_{1 \leq j \leq 4} \{1 + MaxMoves(i + j)\} & \text{otherwise}
\end{cases}$$

Indeed, we are just trying all possible first moves, and trying to maximize the number of moves we do afterward given our choice.

We can compute all values of $MaxMoves$ by filling in an array $MaxMoves[1..n]$. Each subproblem depends on at most four subproblems to the right (with higher $i$ index), so we fill in the array right to left. There are $n$ subproblems taking constant time each to compute for $O(n)$ time total.

Alternatively, we could build a graph $G = (V, E)$. Each square is added to $V$ as a vertex along with a special terminal vertex $t$. We add an edge $u \rightarrow v$ if you can go from square $u$ to square $v$ in one move and an edge $u \rightarrow t$ if you can fall off the right end of the puzzle in one move. Note that no edge goes right-to-left, so there are no directed cycles in the graph. We give all edges a weight of 1, compute the longest paths from the leftmost square to $t$, and return the longest path length. Computing longest paths in a DAG takes $O(V + E) = O(n)$ time.
Describe and analyze an efficient algorithm to assign a room-time slot pair to each class (or report correctly that no such assignment is possible).

**Solution:** We will reduce the problem to finding a maximum matching in a bipartite graph $G = (C \cup P, E)$. Let $C$ be the set of $n$ students. Let $P$ be the set of $rt$ room-time slot pairs. Add an edge between vertex $i \in C$ and $(j, h) \in P$ if and only if $E[i] < S[j]$.

Now, compute a maximum cardinality matching $M$ in $G$. If $|M| < n$, then report no assignment is possible. Otherwise, assign each class $i$ to the room-time slot pair $(j, h)$ for which $\{i, (j, h)\} \in M$.

There are $O(nrt)$ edges in the graph, and the maximum matching has size at most $n$. Using Ford-Fulkerson's algorithm, it will take $O(n^2rt)$ time to compute the maximum matching (an $O(n^2r)$ time algorithm also exists by a more direct reduction to maximum flow).

**Explanation:** Suppose there is an assignment for the classes. Then, every class is assigned one room-time slot pair, and each room-time slot pair will be used exactly once. There will be a matching of size $n$ that contains an edge for each of these assignments.

Similarly, suppose there is a matching of size $n$. Then, each edge of the matching represents a class paired with a room-time slot pair. Each of these objects is used at most once, so we can use those edges as the unique assignments. There are $n$ classes, so the $n$ pairs will involve every class.