Consider a modification to the `Select` algorithm from early in the semester, which partitions the input array into \( \lceil n/6 \rceil \) blocks of size 6, instead of \( \lceil n/5 \rceil \) blocks of size 5, but is otherwise identical. 

(a) State a recurrence for the running time of \( \text{Mom}_6\text{Select} \), assuming that \( \text{MedianOf6} \) runs in \( O(1) \) time.

**Solution:** The top level call spends \( O(n) \) time running the `Partition` procedure. It recursively calls \( \text{Mom}_6\text{Select}(M[1..m]) \) which takes \( T(m) \approx T(n/6) \) time. All that remains is to figure out the running time for the other recursive call at the bottom of the procedure.

Observe there are at least \( n/12 \) blocks of six contiguous elements from \( A \), each of which has a median that is less than or equal to \( \text{mom}_6 \). Within each of these blocks, there are three elements less than or equal to that block’s median. Therefore, there are at least \( (n/12) \cdot 3 = n/4 \) elements less than or equal to \( \text{mom} \), meaning there are at most \( 3n/4 \) elements strictly greater than \( \text{mom}_6 \). Similarly, there are \( n/12 \) blocks with medians greater than or equal to \( \text{mom} \) with three elements per block strictly greater than that block’s median, meaning there are at least \( n/4 \) elements greater than or equal to \( \text{mom}_6 \). Again, there are at most \( 3n/4 \) elements strictly smaller than \( \text{mom}_6 \).

Either recursive call at the end runs in at most \( T(3n/4) \) time. The running time of \( \text{Mom}_6\text{Select} \) follows the recurrence

\[
T(n) \leq O(n) + T(n/6) + T(3n/4).
\]

**Rubric:** 7 points total: 5 points for the recurrence; 2 points for the proof.

(b) What is the running time of the algorithm obtained by solving your recurrence?

**Solution:** We can use recursion trees to solve \( T(n) \). For simplicity, let’s assume the constant on the big-oh is 1. The value of the nodes at depth \( i \) sum to at most \( (11/12)^i \cdot n \). Summing over all the rows gives a decreasing geometric series, bounded from above by a constant times its largest term, \( n \). The running is \( T(n) = O(n) \).

**Rubric:** 3 points total: 2 points for the correct time bound; 1 point for an argument.
In the **Backpack** problem, you are given two arrays $C[1..n]$ and $W[1..n]$ of positive integers where $C[i]$ is the cost of book $i$ in dollars and $W[i]$ is the weight of book $i$ in pounds. You are also given a positive integer $M$ which is the maximum load you can carry in pounds. Your goal is to compute the maximum total cost of any subset of books you can carry from books 1 through $n$.

(a) For any integers $i$ and $m$ with $0 \leq i \leq n$ and $0 \leq m \leq M$, let $MaxCost(i, m)$ be the maximum cost of any subset of books 1 through $i$ that have a total weight of at most $m$. Give a recurrence definition for $MaxCost(i, m)$. Don’t forget the base cases!

**Solution:** We can use the following recursive definition:

$$MaxCost(i, m) = \begin{cases} 
0 & \text{if } i = 0 \\
0 & \text{if } m = 0 \\
MaxCost(i-1, m) & \text{if } W[i] > m \\
\max\{MaxCost(i-1, m), C[i] + MaxCost(i-1, m-W[i])\} & \text{otherwise}
\end{cases}$$

Indeed, any subset of books 1 through 0 contains no books and has total cost 0. If $m = 0$, then we cannot take even a single book, because they all have positive weight. If $W[i] > m$, then we cannot take book $i$ without becoming overloaded, so we have to consider only books 1 through $i-1$. In all other cases, we may choose to take book $i$. If we don’t, then we want the maximum cost subset from books 1 through $i-1$ but have all $m$ pounds to work with still. If we do take book $i$, we get $C[i]$ cost from it, but then we only have $m-W[i]$ pounds left to work with while picking a subset from the remaining books 1 through $i-1$. ■

**Rubric:** 6 points total: 4 points for the recurrence; 2 points for the proof.

(b) Design and analyze a dynamic programming algorithm for solving the **Backpack** problem based on your recurrence from part (a). You should get a running time of $O(nM)$.

**Solution:** We store the solutions to $MaxCost$ in a two-dimensional array $MaxCost[0..n][0..M]$. The bases cases can be filled in $O(n + M)$ time. Every other entry depends upon only those entries that have smaller $i$ index and possibly smaller $m$ index. Therefore, we can fill it in row-major order, by increasing $i$ index for the outer loop and increasing $m$ index in the inner loop. After filling the array, the solution to the **Backpack** problem lies at position $MaxCost[n][M]$.

There are $O(nM)$ entries to fill in, and each takes constant time to fill given its dependencies are already computed. The running time of the algorithm is $O(nM)$ as desired. ■

**Rubric:** 4 points total: 2 points for filling the data structure; 1 point for saying where the solution lies; 1 point for the time analysis.

---

1
Consider the following greedy strategy for computing a smallest set \( P \) of points that stabs \( X \): Let \( x \) be the interval whose right endpoint comes furthest to the left. We add the right endpoint of \( x \) to \( P \), remove all intervals containing \( p \) from \( X \), and recursively add a smallest stabbing set on the remaining intervals to \( P \).

Prove that this strategy does compute a smallest stabbing set for \( X \).

**Solution:** We begin by proving there is some smallest stabbing set that uses the first point considered by our algorithm. Let \( P \) be some smallest stabbing set, and let \( p \) be the leftmost point of \( P \). Let \( p' \) be the first point chosen by our greedy algorithm, and let \( P' = P - p + p' \). We see \( |P'| = |P| \). Since \( p \) is the leftmost point of \( P \), no interval lies completely to the left of \( p \). Therefore, \( p' \) lies to the right of \( p \). Any interval stabbed by \( p \) must then continue on to at least \( p' \), so it is stabbed by \( p' \) as well. We have that \( P' \) is a smallest stabbing set containing \( p' \).

Given an empty set of intervals, the strategy will correctly return an empty stabbing set. We have that the algorithm picks a good first point for the smallest stabbing set of all intervals, and then by induction, it correctly finds a smallest stabbing set for the remaining intervals.

**Rubric:** 10 points total. 7 for the exchange argument; 3 for finishing the proof.
Consider the weighted graph pictured below.

Solution:

1. Draw a depth-first spanning tree rooted at $s$.

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

4. Draw a minimum spanning tree.

Rubric: 10 points total: 2.5 points for each. -1 for one minor mistake, -2.5 for multiple minor mistakes.
Prove that it is \( \text{NP} \)-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

**Solution:** As suggested by the hint, we will do a reduction from 3SAT. Let \( \Phi \) be a 3CNF formula with \( v \) variables and \( c \) clauses for which we want to decide satisfiability. We build a \( c \times (v + 1) \) grid so that each clause has its own row and each variable has its own column (there is an additional last column for technical reasons). For each clause containing a variable and its negation, we add a blue stone to the last entry in that row. For every other clause, for each literal in that clause, we add a stone to the row, column pair representing that clause and the literal’s variable; if the literal is the (non-negated) variable itself, we use a blue stone. Otherwise, we use a red stone. Finally, we solve 3SAT on \( \Phi \) by return whether or not it is possible to solve the puzzle. The reduction takes \( O(cv) \) time in addition to the time spent deciding if the puzzle has a solution, so it is a polynomial time reduction.

We still need to prove that \( \Phi \) is satisfiable if and only if the puzzle has a solution. Suppose \( \Phi \) is satisfiable. For each true variable in the satisfying assignment, we remove the red stones in that variable’s column. For each false variable, we remove the blue stones. We remove no stones from the last column. Every column is left with stones of exactly one color. If a clause has a variable and its negation, then there is a blue stone in the last entry of its row. For every other clause, there is a stone still in its true literal’s variable’s entry in the clause’s column.

Finally, if the puzzle has a solution, then each column contains stones of exactly one color. If that color is blue or there are no stones in that column, we set the variable to true. Otherwise, we set the variable to false. If a row contains a stone in its last entry, then the clause for that row had a variable and its negation, so it is satisfied by any assignments to the variables. For other rows, there is at least one stone, and the color of that stone corresponds to a variable assignment that satisfies the clause.

We conclude that a polynomial time algorithm to decide if the puzzle can be solved implies a polynomial time algorithm for 3SAT, meaning \( P = \text{NP} \). Determining if the puzzle can be solved is \( \text{NP} \)-hard.

**Rubric:** 10 points extra credit total: 5 points for the reduction, 1 point for running time, 2 points each for both direction of the proof.