Describe and analyze an algorithm to determine in $O(n)$ time whether an arbitrary array of numbers $A[1..n]$ contains more than $n/4$ copies of any value. You do not need to justify correctness of your algorithm. An $O(n \log n)$ time algorithm is worth a good amount of partial credit.

Solution: We use the procedure $\text{Select}(X[1..n],k)$ from class which returns the $k$th smallest element in an array $X[1..n]$. The below procedure $\text{FindCommonElement}(A[1..n])$ returns $\text{True}$ if and only if $A$ contains at least $n/4$ copies of any value.

```
\text{FindCommonElement}(A[1..n]):
\begin{align*}
R[1..4] &\leftarrow \{1,\lceil n/4 \rceil,\lfloor 2n/4 \rfloor,\lfloor 3n/4 \rfloor\} \\
\text{for } r \leftarrow 1 \text{ to } 4 \\
    a &\leftarrow \text{Select}(A[1..n],R[r]) \\
    \text{count } \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n \\
    \text{if } a = A[i] \\
        \text{count } \leftarrow \text{count } + 1 \\
\text{if } \text{count } > n/4 \\
    \text{return } \text{True} \\
\text{return } \text{False}
\end{align*}
```

The algorithm calls an $O(n)$ time procedure four times and does four for loops so it takes $O(n)$ time total.

Explanation: If an element appears more than $n/4$ times, then it must contain an element from one of the four ranks being searched in the procedure. We figure out the elements of these ranks and return $\text{True}$ if and only if one of them appears more than $n/4$ times.
A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACANALPANAMA. Note that a palindrome may have an odd number of characters.

(a) (5 out of 10) Let \( X[1..n] \) be an array of characters, and let \( \text{MaxPalSub}(i, j) \) be the length of the longest subsequence of \( X[i..j] \) that is also a palindrome. Fill in the blanks to complete the following recursive definition of \( \text{MaxPalSub}(i, j) \).

Solution:

\[
\text{MaxPalSub}(i, j) = \begin{cases} 
0 & \text{if } i > j \\
1 & \text{if } i = j \\
2 + \text{MaxPalSub}(i+1, j-1) & \text{if } i < j \text{ and } X[i] = X[j] \\
\max\{\text{MaxPalSub}(i, j-1), \text{MaxPalSub}(i+1, j)\} & \text{otherwise}
\end{cases}
\]

Explanation: If \( i = j \), then the entire substring is a palindrome of length 1. If \( i < j \) and \( X[i] = X[j] \), then we can safely include both the first and last character in our palindrome substring. Otherwise, we have to drop one of the first or last characters, and we drop the better of the two options.

(b) (5 out of 10) Use dynamic programming to write an iterative algorithm that returns the length of the longest palindrome subsequence in \( X[1..n] \) based on the above recurrence. What is the running time of your algorithm? You may assume you filled the blanks correctly in the previous part.

Solution: The procedure \( \text{MaxPalSubSequence}(X[1..n]) \) returns the length of the longest palindrome subsequence of \( X \).

\begin{verbatim}
MaxPalSubSequence(X[1..n]):
    for i <-- n down to 1
        for j <-- 1 to n
            if i > j
                MaxPalSub[i, j] <-- 0
            else if i = j
                MaxPalSub[i, j] <-- 1
            else if X[i] = X[j]
                MaxPalSub[i, j] <-- 2 + MaxPalSub[i + 1, j - 1]
            else
                MaxPalSub[i, j] <-- max{MaxPalSub[i, j - 1], MaxPalSub[i + 1, j]}
    return MaxPalSub[1, n]
\end{verbatim}

The algorithm has a doubly-nested for loop over \( n \) elements, so the running time is \( O(n^2) \).
**Explanation:** We need to return $MaxPalSub(1, n)$. The recurrence depends upon larger values of $i$ or smaller values of $j$, so we evaluate subproblems in decreasing order by $i$ and increasing order of $j$. 
Suppose we are given an undirected graph $G$ in which every vertex has a positive weight.

1. (3 out of 10) Describe and analyze an algorithm to find a spanning tree of $G$ with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.) You do not need to justify correctness of your algorithm.

   Solution: We run a breadth-first search from an arbitrary vertex to find a spanning tree of $G$ and return it. This algorithm takes $O(V + E)$ time.

   Explanation: Every spanning tree contains the same set of vertices, so they all have the same weight. A BFS tree is as good as any.

2. (5 out of 10) Describe and analyze an algorithm to find a path in $G$ from one given vertex $s$ to another given vertex $t$ with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.) You do not need to justify correctness of your algorithm.

   Solution: We modify $G = (V, E)$ to create a new graph $G' = (V, E')$ as follows: For each edge $uv \in E$, we create two edges $u \rightarrow v$ and $v \rightarrow u$ in $E'$. Then, we weight each edge $u \rightarrow v$ with the weight of $v$ in $G$.

   Finally, we run Dijkstra’s algorithm with Fibonacci heaps to find the shortest path from $s$ to $t$ in $G'$. We return this path. Creating $G'$ and running Dijkstra’s algorithm takes $O(E + V \log V)$ time total.

   Explanation: By construction, the total weight of edges in any path of $G'$ is the total weight of head vertices on the edges. Which means its the total weight of all vertices of $G$ on the path with the exception of $s$. Because $s$ appears in every $s$ to $t$ path exactly once, it's sufficient to find the shortest path considering all vertex weights except $s$. 
Describe and analyze a recursive or dynamic programming algorithm to compute the minimum number of rounds required to broadcast a message to all nodes in a binary tree. Your algorithm should run in $O(n)$ time. You do not need to justify correctness of your algorithm.

**Solution:** The procedure $\text{NumRounds}(v)$ computes the minimum number of rounds needed to broadcast the message throughout $v$'s subtree assuming we start with only $v$ knowing the message. We use $\text{left}(v)$ and $\text{right}(v)$ to denote $v$'s left and right children (if they exist).

```plaintext
\begin{algorithm}
\textbf{NumRounds}(v):
  if $v$ is a leaf
    return 0
  else if $v$ has one child $c$
    return 1 + \text{NumRounds}(c)
  else
    $\ell \leftarrow \text{NumRounds(\text{left}(v))}$
    $r \leftarrow \text{NumRounds(\text{right}(v))}$
    if $\ell = r$
      return $2 + \ell$
    else
      return $1 + \max\{\ell, r\}$
\end{algorithm}
```

The algorithm spends constant time at each node outside the recursive calls, so it runs in $O(n)$ time.

**Explanation:** If $v$ is a leaf, there is no need to broadcast the message. If $v$ has one child $c$, we spend 1 round sending the message to the child and then $\text{NumRounds}(c)$ rounds waiting for the child's subtree to receive the message. Otherwise, we can assume inductively that it takes $\ell$ rounds to send a message throughout the left subtree and $r$ rounds to send it throughout the right subtree. If $\ell = r$, it takes 2 rounds for both children to receive the message plus another $\ell$ rounds for the latter subtree to spread the message. Otherwise, we should first send the message to the worst of the two subtrees. By the time its done spreading, the better of the subtree can also receive the message and finish spreading it.
Suppose we are given a bipartite graph with vertex set \( L \cup R \) and edge set \( E \) such that every edge joins a vertex in \( L \) to a vertex in \( R \). Let \( n = |L| + |R| \) be the number of vertices and \( m = |E| \) be the number of edges. Describe and analyze an algorithm to find the size of the largest subset of edges such that every vertex in \( L \) is incident to at most 3 edges and every vertex in \( R \) is incident to at most 2 edges. You do not need to justify correctness of your algorithm. (Note we want a subset of edges, meaning each edge is included in the subset at most once.)

**Solution:** We build a flow network \( G' = (V', E') \) as follows. \( V' \leftarrow L \cup R \cup \{s, t\} \). We add an edge \( s \rightarrow \ell \) of capacity 3 to \( E' \) for all \( \ell \in L \). We add an edge \( \ell \rightarrow r \) of capacity 1 for all \( \ell r \in E \). We add an edge \( r \rightarrow t \) of capacity 2 for all \( r \in R \).

Now, we obtain an integer maximum flow \( f^* \) using the Ford-Fulkerson augmenting path algorithm. We return the set of edges \( \ell r \) such that \( f^*(\ell \rightarrow r) = 1 \). We cannot use more than 3\( n \) edges, so \( |f^*| = O(n) \). Building the graph, finding the maximum flow, and returning the subset of edges takes \( O(E|f^*|) = O(mn) \) time.

**Explanation:** If there's a subset of size \( k \), we can build a flow by assigning to each edge \( s \rightarrow \ell \) the number of edges incident to \( \ell \), to each edge \( \ell r \) in the subset a value of 1 and to each edge \( r \rightarrow t \) the number of edges incident to \( r \). Similarly, for the flow of value \( k \), we can use the flow paths to extract edges from \( L \) to \( R \) that don't include individual vertices too many times.
Both parts ask you to prove a problem is NP-hard. For both parts, you must argue that your
reduction is correct.

1. (5 out of 10) Prove the following problem is NP-hard using a reduction from 3SAT. Given
a boolean formula \( \Phi \) in conjunctive normal form with at most four literals per clause,
determine whether \( \Phi \) has a satisfying assignment.

**Solution:** Let \( \Phi \) be a 3CNF formula given as input to 3SAT. It has at most four literals per
clause, because \( 3 \leq 4 \). Therefore, a solution to the proposed problem will determine if the
formula can be satisfied, which is exactly the output desired by 3SAT. We spend constant
time doing the reduction, so it’s polynomial time.

2. (5 out of 10) Prove the following problem is NP-hard using a reduction from HAMILTON-
IANCYCLE in undirected graphs. Given a complete undirected graph \( K_n \) over \( n \) vertices
with non-negative weights on the edges, find a minimum weight cycle that includes every
vertex. (The weight of a cycle is the sum of its edge weights.)

**Solution:** Let \( G = (V, E) \) be an undirected graph given as input to HAMILTONIANCYCLE.
We create a new weighted complete undirected graph \( G' = (V, E') \) as follows. For each
edge \( uv \in E \), we add \( uv \) to \( E' \) with weight 1. For each edge \( uv \notin E \), we add \( uv \) to \( E' \) with
weight 2. Finally, we run the posed problem. If the minimum weight cycle has weight
\( |V| \), we report there is a Hamiltonian cycle in \( G \). Otherwise, we report there is not. The
reduction takes \( O(V^2) \) time so it is polynomial.

Suppose there is a Hamiltonian cycle in \( G \). The edges of the cycle each have weight 1
in \( G' \), so the whole cycle has weight \( |V| \). Every cycle has \( |V| \) edges, so they all have weight
at least \( |V| \), meaning \( |V| \) is the weight of the minimum weight cycle.

Suppose there is a cycle of weight \( |V| \) in \( G' \). We cannot have any edges of weight 2 in
the cycle or it would have total weight at least \( |V| + 1 \). Therefore, every edge of the cycle
appears in \( G \) and it is a Hamiltonian cycle.