Problem 1

There are many ways to solve this problem. One way is to use a DFS that is initiated from a designated root vertex. Each vertex should store the distance to the most distant leaf in its subtree. Then each vertex can pick the two children with the greatest distance values and use the sum of these two values plus 2 as a candidate diameter value. With one exception, the greatest of these candidate sums is the diameter. The exception occurs if the root has one child and the distance to its most distant leaf is larger than the candidate sums. This case is easy to include.

Another approach is to identify the vertices with just one neighbor in the undirected graph. These are the candidate endpoints of the diameter. Then a BFS-style topological sort can peel off the endpoints one level at a time until just one or two vertices remain. The diameter equals twice the number of peeling phases if just one vertex remains at the completion of the algorithm, and one more than this phase count if two vertices remain. The algorithm is as follows.

function $Diam((V, Adj[*])$:undirected tree);
for each v in V do
    $v.count \leftarrow |Adj[v]|$;
    if $v.count = 1$ then insert v into $Ready$ endif
endfor
$unprocessed \leftarrow |V|$;
$diam \leftarrow 0$;
while $unprocessed > 2$ do
    repeat
        $w \leftarrow RemoveFrom(Ready)$;
        $unprocessed \leftarrow unprocessed - 1$;
        for each vertex x in $Adj[w]$ do
            $x.count \leftarrow x.count - 1$;
            if $x.count = 1$ then insert x into $Waiting$ endif
        endfor;
    until $Ready$ empty;
    $diam \leftarrow diam + 2$;
    empty $Waiting$ into $Ready$
endwhile;
if $unprocessed = 2$ then $diam \leftarrow diam + 1$;
return(diam);
end-Diam.

Problem 2

First find the strong components (SC) of G, and create the reduced graph that has a vertex for each SC of G. There is an edge from u to v in the reduced graph if there is an edge in G from some vertex in the SC that corresponds to u to some vertex in the SC that corresponds to v. Clearly the reduce graph can be formed in $O(|V|+|E|)$ time. Moreover, the reduced DAG is also semiconnected. This graph is a DAG where every pair of vertices has a connecting path (in one direction). The graph must be a linear chain since any vertex with two outgoing edges has two immediate descendants that are unrelated, and likewise for incoming edges and predecessors.

But it is straightforward to test if the reduced graph is a chain in linear time.

Problem 3

The problem is clearly in NP. We simply guess a poor vertex cover, and count the number of edges covered.

To prove PVC is NPC, we reduce VC to PVC. So let $G = (V, E)$ be a graph with a target VC parameter k. Create a new graph $\widehat{G} = (\widehat{V}, \widehat{E})$ that contains G, and $999|E|$ edges that are just isolated edges which are connected to nothing else. Thus each such edge comprises two new vertices and the edge between them. Now a VC of k vertices for the original graph covers .1% of the edges in the new graph. Likewise, suppose that some set of k vertices cover .1% of the edges in $\widehat{G}$. To see that the original G must have a VC of k vertices, keep the portion of the PVC that has vertices in V. Each of the remaining vertices in the PVC covers just one edge. So replace these vertices one-by-one by a vertex that covers an as yet uncovered edge in G (if one still exists). When you are done, you must have either covered all of E or have covered a number of edges in E that equals .1% of $1000|E|$, which is to say that you covered all of E.

Problem 4

a. Let the string be stored in $Bit[1..n]$. Let $Word[i, j]$ be a Boolean function that is true if $Bit[i..j]$ is in L and false if not.

Then we can define the recognizer by the recursive function

$$Recog(j) = \begin{cases} True & \text{if } j = 0, \\ \bigvee_{0 \le i < j} (Recog(i) \wedge Word[i+1, j]) & \text{if } j > 0, \end{cases}$$

where $\bigvee_{0 \le i < j}$ is the j-way logical OR.

b. Same solution where $Word[i, j]$ is true if $String[i..j]$ is in L and false otherwise.

c. $O(|w|^{r+2})$. Reason: we compute the charge to evaluate $\bigvee_{0 \le i < j}(Recog(i) \wedge Word[i + 1, j])$ as $\sum_{0 \le i < j} (1 + (j - i)^r)$, where the 1 is for a table lookup of $Recog(i)$, since its Boolean evaluation should be computed just once for each i. Summing over j gives $\sum_{0 \le i \le j \le n} (j - i + 1)^r = O(n^{r+2})$.

Problem 5

a. Let $L = 0^{2n-1}1$. The language regular. However, $swap(L) = 0^n10^{n-1}$, which is clearly not regular. To see that it is not regular, we use the pumping lemma, which says, in this case, that for large enough words, a $w \in swap(L)$, if the language were regular, could be written as $w = xyz$ where $|y| \neq 0$ and xy^iz is also in the language for all $i \geq 0$. But then y must be a string of zeros only, whence xy^iz will have $(i-1)|y|$ more zeros on the y side of the 1 than is the case for xyz. This cannot be, since the number on the other side will remain the same.

b. Let D be a FSM that recognizes L. We build a nondeterministic PDA to recognize L. We ignore the issue of recognizing 1 (or not), since it is trivial. Let $w = yx$ with $|x| = |y|$. We build a PDA that on input w decides if D recognizes xy. To do so, the PDA must first decide if y is a correct second half of a word that D will recognize. To do so, the PDA first guesses the state σ that D will be in after processing x. It then processes the first part of w by emulating D starting from σ. It must also guess when the first half of w has been processed, and check to see if its simulation of D on y starting from σ is in an accepting state. If so, it then starts simulating D on (the guessed) second half of w, beginning from the start state of D. Once this second half is processed, it checks to see if its simulated state for D is σ as originally guessed.

The details: There are only a finite number of states for D, and hence the guessed state can be remembered in the finite control structure of the PDA. When processing the first half of w, a 1 is pushed onto the PDA for each symbol processed. Then the remaining characters that are processed can be counted to see if they have the same count. This is done by popping a 1 for each character processed during the recognition of x. The stack must be empty when w has bee processed, σ must be correct, and the acceptance state reached after y has been processed for w to be recognized.

The scheme is nondeterministic. So w is accepted only if a correct pair of guesses for σ and the location of the first half of w can be found, and D winds up accepting w.

c. We exploit the limited ability of PDA's to recall values. Let $L = a^{3m}b^m$, which is easily recognized by a PDA. $Swap(L) = a^mb^ma^{2m}$, which is just as difficult to recognize as $a^nb^na^n$.

Formally, we use the pumping lemma, which says that if $Swap(L)$ is CF, then for large enough n, a w in $Swap(L)$ can be written as $w = uvxyz$ where $|vy| > 0$, and uv^ixy^iz must also be in $Swap(L)$ for $i = 0, 1, 2, \ldots$. But if this is true, then v must be all of one kind of letter, and likewise for y. Since the number of a's grow no matter what, and the number of a's at the front is just half of the number of a's at the back, then both strings must be all a's. But this cannot be true, since the number of b's would then not grow. Hence the language is not CF.

Problem 6

a. Use a vanilla Dijkstra's algorithm with all B and and P vertices initialized to be sources (have distance zero from the source), and other vertices to have a distance that is infinite. Also initialize a $.security$ field in the B and P vertices to be of type $Insecure$ and $Secure$ respectively. The update lines in Dijkstra's algorithm reads:

```
v ← DeleteMin(Q);
for each neighbor w in Adj[v] do
    if sDist[w] > sDist[v] + EdgeCost[v, w] then
        sDist[w] ← sDist[v] + EdgeCost[v, w];
        w.security ← v.security
    endif
endfor;
```

b. Here the code is similar, but there are two adaptations. First, an edge cost will be treated as one third of its listed cost for a robber. Second, a little justification is needed for correctness. There can be locations that the robbers could reach first, but which will not be reached by them because every path to the location passes through a location that the police reach first. So we cannot run separate Dijkstra's algorithms for the robbers and the police, and just award each vertex to the agent who gets there first. Instead, we run the algorithms in parallel (as one algorithm), and thereby avoid spreading robbers from a location which the police reach first, because the vertex will be categorized as secure and removed from the queue once it is known that the police must be the first to arrive. So the initialization is as above, and we have two update loops:

```
v ← DeleteMin(Q);
if v.security = Secure then
    for each neighbor w in Adj[v] do
        if sDist[w] > sDist[v] + EdgeCost[v, w] then
            sDist[w] ← sDist[v] + EdgeCost[v, w];
            w.security ← Secure
        endif
    endfor
else
    for each neighbor w in Adj[v] do
        if sDist[w] > sDist[v] + EdgeCost[v, w]/3 then
        sDist[w] ← sDist[v] + EdgeCost[v, w]/3;
        w.security ← Insecure
        endif
    endfor
endif;
```