Problem 1 [Optimal Prefix Code]

Since it is a prefix code, note that all the codewords have to sit in the leaves. Else, some codeword would have to be a prefix of some other codeword.

Suppose the binary tree is not full, i.e., there is some node which is not a leaf having only one child. We could as well do away with that single branch, and push up the child (and all it children) by one level. This would still remain a valid prefix code, each character having distinct binary codewords. Clearly this would be of less average length than the tree that we started with. So the initial tree could not have been that of an optimal prefix code.

Problem 2 [Ternary Huffman Code]

The algorithm is very similar to the Huffman code that we have seen in the class. Pick the smallest three frequencies, join them together and create a node with the frequency equal to the sum of the three. Repeat. However, notice that every contraction reduces the number of leaves by 2 – we remove 3 nodes and add 1 back. So to make sure that we end up with just one node, we have to have an odd number of nodes to start with. If not, add a dummy node with 0 frequency to start with.

Correctness : We shall show that any optimal tree has the lowest three frequencies at the lowest level. Suppose not. We could switch a leaf with a higher frequency from the lowest level with one of the lowest 3 leaves, and obtain a lower average length. Without any loss of generality, we can assume that all the three lowest frequencies are the children of the same node. (if they are at the same level, the average length does not change irrespective of where the frequencies are)

Now, observe that we can treat the contracted leaves as a new character with frequency equal to the sum of the frequencies of the three characters. By a similar reasoning to what was given in the class for the binary Huffman codes, we can see that the cost of the optimal tree is the sum of the tree with the three symbols contracted and the eliminated mini tree which had the nodes before contraction. Since it has been proved that the mini tree has to be present in the final optimal tree, we can optimize on the tree with the
contracted node.

Problem 3 [Fibonacci Numbers and Huffman Trees]

The optimal code for the given frequencies will be given by the following tree.

```
  54
   /|
  0 / | 1
   / |
 h-21 |-- 33
   /|
  0 / | 1
   / |
 g-13 |-- 20
   /|
  0 / | 1
   / |
 f-8 |-- 12
   /|
  0 / | 1
   / |
 e-5 |--  7
   /|
  0 / | 1
   / |
 d-3 |--  4
   /|
  0 / | 1
   / |
 c-2 |--  2
   /|
  0 / | 1
   / |
 b-1 |
 a-1
```

Verify that the above tree is correct (Exercise). To generalize for the
first \( n \) Fibonacci numbers, notice that the sum of the first \( n-2 \) Fibonacci numbers is \( F_{n-1} \) where \( F_n \) denotes the \( n \)th Fibonacci number. (Exercise: Prove that \( \sum_{i=0}^{n-2} F_i = F_n - 1 \) using induction).

So the tree for the first \( n \) Fibonacci numbers will be of the above form, one long branch from the root to the lowest leaf, and branches of length 1 hanging from it. This is because, after we contract the first \( n-2 \) numbers into a node, the total frequency of that node will be \( F_{n-1} \) which is more than \( F_{n-1} \) (which is the least remaining) and less than \( F_n \) (the second least remaining). So this node will combine with \( F_{n-1} \) and a similar argument will follow by induction for all \( n \).

**Problem 4 [2-SAT is in P]**

Consider the clauses \( x \lor y \). If \( x \) is FALSE, this implies that \( y \) has to be true. In other words, \( \neg x \Rightarrow y \). Similarly, \( \neg y \Rightarrow x \). So every clause \( x \lor y \) can be replaced by the above two implications.

Given a 2-SAT instance, we can construct the following graph. There will be \( 2n \) vertices, one each for each variable and its negation. Also, we can construct directed edges as follows. For every clause \( x \lor y \), we add two directed edges \( \neg x \rightarrow y \) and \( \neg y \rightarrow x \).

Notice that a path is a series of implications and a path from \( x \) to \( \neg x \) gives us the implication that if \( x \) is TRUE, \( x \) must be FALSE, which clearly contradict. Since we construct two directed edges per clause, one from \( \neg x \rightarrow y \) and one from \( \neg y \rightarrow x \), this means that there is a path from \( x \) to \( \neg x \) if and only if there is a path from \( \neg x \) to \( x \). The presence of these paths for any literal \( x \), means that \( x \) cannot be TRUE or FALSE in a satisfying assignment, hence there is no satisfying assignment.

If there is no such path for any literal, using the graph, we can construct a satisfying assignment. We can pick any unassigned vertex, and assign TRUE. If there is a chain of implications following from it, assign TRUE to all the vertices in paths starting from the chosen vertex. Also, assign all the corresponding negations to be FALSE. Notice that we will not hit any conflict in this case. We already know that there are no paths from \( x \) to \( \neg x \). So the only way we can hit a conflict is when there are two paths from \( x \), one leading to a literal \( y \) and another one leading to \( \neg y \). But if there is a path from \( x \) to \( \neg y \), there should be a path from \( y \) to \( \neg x \). This means that there is a path from \( x \) to \( \neg x \). Since this will never occur, we can be assured that we can safely assign TRUE or FALSE to all the vertices, and hence the literals.
for a satisfying assignment. So the 2-SAT has a satisfying assignment if and only if the constructed has no path from $x$ to $\neg x$ for any literal $x$.

For the running time, notice that we are working with a graph with $2n$ vertices. Checking if there is any path from $x$ to $\neg x$ can be done in polynomial time, as we have seen earlier in the course. Of course, constructing the graph can also be done in polynomial time. Hence we can solve 2-SAT in polynomial time.