1 Problem 4a

In this case, the light bulbs will blow up because of varying propagation delays in the circuit. The bits will not all reach the decoder at the same time, so intermediate states and delays will be reflected in the output of the decoder. For example, when going from 0011 to 0100 it is possible that any of 0101, 0110 and 0111 will be lit briefly. This can happen to any of the bulbs.

2 Problem 4b

Adding a register between the counter and the decoder will not solve this problem. The wire delays in the register still vary, so the same problem from part (a) applies. In order to fix this problem with registers it is necessary to add a one-bit register on each output of the decoder. This assures that only one light is on at each cycle and that the lights do not flicker.

3 Problem 4c

Using a Gray-coded counter will solve all the problems with light bulbs blowing up. The main problem with the original circuit is that it is possible for a large number of bits to change between cycles. Using a Gray-coded counter insures that only one bit can change at each cycle. No matter what the propagation delay, that one bit is the only thing changing, so there are no flickers.

4 Problem 4d

The circuit from part (b) doesn’t work because the delay through each of the registers could be different. If the delay through the register for the last light is slower than that of the register for the current light, they will overlap for a time. The circuit for part (c) is much worse, as there are clearly delays through
the decoder that will cause lights to overlap. The problem can be fixed using the same process we used for creating a priority encoder. Taking advantage of the sequential nature of the circuit, you add an AND gate in front of each bulb that computes \( \text{in}(i) \land \neg \text{out}(i - 1) \) where \( \text{in} \) is the output from the decoder and \( \text{out} \) is the output from the AND gate for the \( i \)'th bulb. This insures that the last bulb is off before the current one goes on.

5 Problem 5a

![Figure 1: The tree for problem 5a.](image)

6 Problem 5b

Given a single variable \( a \), there are only four possible functions: 0, 1, \( \bar{a} \) and \( a \). Using a fully simplified tree representation, there is only one way to compute each of these functions.

7 Problem 5c

We need to show that factoring two functions with the same variable ordering produces two identical trees if and only if the functions are also identical. We can do this with a straightforward induction on the number of variables. The base case is for one variable, which we have already shown in part (b). We assume that the all \( n \) variable functions are identical if and only if their tree representation is also identical (using the same order in factoring). We need to prove that this holds for \( n + 1 \) variable functions. Define four \( n \) variable functions, \( f_1, f_2, g_1 \) and \( g_2 \). Let \( f_1 = g_1 \) and \( f_2 = g_2 \). From our assumption we know that \( T_{f_1} \), the tree for \( f_1 \) is equivalent to \( T_{g_1} \), and similarly for \( f_2 \) and \( g_2 \). Now build two \( n + 1 \) variable functions, \( F \) and \( G \) as follows. Define
a new variable and make it the root of $F$. When this variable is zero, select the result of $f_1$, when the variable is one, select the result of $f_2$. Make $G$ the same way, using $g_1$ and $g_2$. Now, if you decompose $F$ and $G$ the same way, their roots will clearly be the same. Furthermore, their left and right subtrees will be equivalent, as $T_{f_1} = T_{g_1}$ and similarly for the right tree. Now, because this is the case for any set of four $n$ variable functions, we have shown that all $n+1$ variable functions are identical if and only if their tree representations are identical (using the same factoring).

8 Problem 5d

You can use the result from problem 5c to check for function equality. Simply decompose the functions using the same variable ordering and check whether their trees are identical. This can be done using any sort of coordinated recursive search of the two trees.

Checking the satisfiability of a function is as simple as checking the leaf nodes. If any of the leaf nodes is a one, the function is satisfiable. If they are all zeroes, the function is not satisfiable.

9 Problem 6a

To answer this question, it is useful to look at the exact cost for each of the implementations.

1. **ROM.** For a function with nine inputs, there will be $2^9$ min terms. This means that we will need nine NOT gates, $2^9$ AND gates (with nine inputs each) and four OR gates (with $2^9$ inputs each).

2. **PLA.** We will at least need nine NOT gates and four OR gates. The number of AND gates depends on the number of terms in the function. In this case, as we are essentially doing a sequence of additions, we will need to use every min term except the zero term. So we need essentially the same number of AND gates as a ROM.

3. **PAL.** The same analysis applies here, as in PLA’s, but in this case the functions can’t share terms, so there will be a lot of repetition, creating even more AND gates.

4. **Mux.** To do this using multiplexors would require four $2^8$ input multiplexors. That means $2^{10}$ AND gates, which discounts this implementation immediately.

5. **Decoder.** Using a decoder is exactly as expensive as using a ROM. It requires a decoder with $2^9$ outputs, which uses $2^9$ AND gates, four OR gates and nine inverters.
The ROM and decoder are the best solutions, given this information. We don't want the overhead of programming the PLA, and the other two are just more expensive.

10 Problem 6b

See the analysis above.

11 Problem 6c

One way to approach this problem is the following. Implement a three bit adder with two bit output using two muxes. At the first level of logic, use three of these adders to add three bits each. Now you have three two-bit numbers. At the second level of logic, use two adders: one to add the low order bits and one to add the high order bits. The low order bit of the first adder is the low order bit of the result. Now, at the last level of logic, use three muxes to combine the high order bit from the first adder and both bits from the second adder (you are basically adding a two bit number to a one bit number here). This will give you three output bits, which are the top three bits of the result. This requires three levels of logic with six muxes on the first, four on the second and three on the last, for a total of thirteen.

12 Problem 6d