Problem 3.59 Solution:

This problem gives students practice analyzing disassembled code. The `switch` statement contains all the features one can imagine—cases with multiple labels, holes in the range of possible case values, and cases that fall through. The main trick is to use the jump table to identify the different entry points, and then analyze each block of code separately.

```
1  int switch_prob(int x, int n)
2  {
3      int result = x;
4
5      switch(n) {
6      case 50:
7      case 52:
8          result <<= 2;
9          break;
10     case 53:
11         result >>= 2;
12         break;
13     case 54:
14         result *= 3;
15         /* Fall through */
16     case 55:
17         result *= result;
18         /* Fall through */
19     default:
20         result += 10;
21     }
22
23     return result;
24 }
```

Case 50 is the default because sub 50
if edx is not above 5, default, towards the end
Jump table first one is "array" index 50
Jump table second one would have to be case 51, but it jumps to address of the default

Problem 3.64 Solution:

This exercise both introduces new material and makes students spend time examining the IA32 stack structure and how it is accessed by machine code.

A. We can see that the value at offset 8 from `%ebp` is a pointer to the structure where the function is to fill in its results. The values at offsets 12 and 16 correspond to the fields `s1.a` and `s1.p`, respectively.

B. Starting from the top of stack, we can see that the first field points to the location allocated for the returned structure. The next two fields correspond to argument values `s1.a` and `s1.p`. The final two fields are where the result of `word_sum` is stored, and so are the values of `s2.sum` and `s2.diff`.

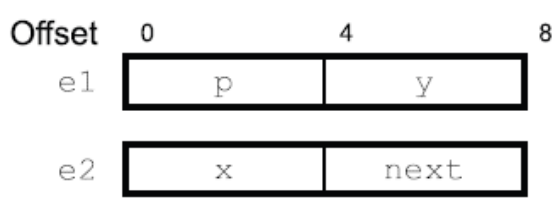
C. The general strategy is to pass the argument structure on the stack, just as any argument is passed. The callee then accesses the fields of its argument by offsets relative to `%ebp`.

D. The general strategy is for the caller to allocate space in its own stack from for the result structure, and then it passes a pointer to this structure as a hidden first argument to the function.

Problem 3.67 Solution:

This is a very tricky problem. It raises the need for puzzle-solving skills as part of reverse engineering to new heights. It shows very clearly that unions are simply a way to associate multiple names (and types) with a single storage location.

A. The layout of the union is shown in the table that follows. As the table illustrates, the union can have either its "`e1`" interpretation (having fields `e1.p` and `e1.y`), or it can have its "`e2`" interpretation (having fields `e2.x` and `e2.next`).



B. It uses 8 bytes.

C. As always, we start by annotating the assembly code. In our annotations, we show multiple possible interpretations for some of the instructions, and then indicate which interpretation later gets discarded. For example, line 2 could be interpreted as either getting element `e1.y` or `e2.next`. In line 3, we see that the value gets used in an indirect memory reference, for which only the second interpretation of line 2 is possible.

```
1    movl    8(%ebp), %edx      Get up
2    movl    4(%edx), %ecx      up->e1.y (no) or up->e2.next
3    movl    (%ecx), %eax       up->e2.next->e1.p or up->e2.next->e2.x (no)
4    movl    (%eax), %eax       *(up->e2.next->e1.p)
5    subl    (%edx), %eax       *(up->e2.next->e1.p) - up->e2.x
6    movl    %eax, 4(%ecx)      Store in up->e2.next->e1.y
```

From this, we can generate C code as follows:

```
1 void proc (union ele *up)
2 {
3     up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
4 }
```

Problem 3.70 Solution:

This problem combines the topics of recursive functions, conditional moves, data structures, and x86-64 code.

A. Here is the code

```
1  /* This looks for the max value in a tree */
2  long traverse(tree_ptr tp) {
3      if (!tp)
4          return LONG_MIN;
5      else {
6          long val = tp->val;
7          long lval, rval;
8          lval = traverse(tp->left);
9          if (lval > val)
10             val = lval;
11         rval = traverse(tp->right);
12         if (rval > val)
13             val = rval;
14         return val;
15     }
16 }
```

B. This code finds the maximum value in a tree. It returns $TMin_{64}$ for an empty tree.