1. The compiler for an object-oriented language has generated the following x86 code for a method:

```
_foo:
     push %ebp
     mov %esp, %ebp
     add $-4, %esp
     mov 12(%ebp), %eax
     add 16(%ebp), %eax
     mov %eax, -4(%ebp)
     cmp $2, -4(%ebp)
     jle L2
     mov -4(%ebp), %eax
     push %edx
     mov 8(%ebp), %edx
     sub 20(%ebp), %eax
     mov %eax, 8(%edx)
     pop %edx
     jmp L3
L2:  mov 8(%ebp), %ebx
     mov -4(%ebp), %eax
     add 20%(ebp), %eax
     mov %eax, 8(%ebx)
L3:  mov 8(%ebp), %eax
     mov %ebp, %esp
     pop %ebp
     ret
```

Assume that the language supports both static and virtual methods. Also assume that the stack grows downward.

(a) Show the layout for all of the pieces of memory that the execution of this method accesses. Leave blank entries for the pieces that cannot be inferred from this code alone.

(b) Show a possible input program that this code may have been generated from.

(c) Is it possible to identify whether this method is static or virtual by inspecting its generated code? Explain.

(d) Can the instruction "`add $-4, %esp`" be optimized away in this example? Precisely in what situations is it safe to discard this instruction?

2. Consider the following function which implements the insertion sort algorithm.

```
void sort(int[] v, int n) {
  int i = 1;
  while (i < n) {
    int k = v[i];
    int j = i-1;
    while(j >= 0 && v[j] > k) {
      v[j+1] = v[j];
      j = j-1;
    }
    v[j+1] = k;
    i = i+1;
  }
}
```

In this program, the test condition in the inner while loop uses a short-circuit "and" operation `&&`. At the end of the function, no variable is live.

(a) Write the three-address code (low IR) for this program. In the generated code, array indices must represent byte offsets (not element offsets). This means that you have to translate statements such as `k = v[i]` into sequences `t = i*4; k = v[t]`. Also, when short-circuit boolean expressions occur in the conditions of while loops, your code must efficiently combine the control-flow in these two constructs as discussed in class.

(b) Construct the control flow graph from the low IR code.

(c) Show the dominator tree, the back edges, and the natural loops for the program in part (b).

(d) Optimize the program. Perform common subexpression elimination, dead code elimination, strength reduction, and elimination of induction variables. Indicate the changes in the code for each optimization and show the final, optimized program.

3. We want to design a dataflow analysis which compute ranges for integer variables in the program. For this, we extend the set $\mathbb{N}$ of integer numbers with plus and minus infinity: $\mathbb{N}^* = \mathbb{N} \cup \{+\infty, -\infty\}$, such that $-\infty < n$, and $n < +\infty$ for any integer number n. We then use a lattice over the set $L = \{[l, u] \mid l, u \in \mathbb{N}^* \wedge l \leq u\} \cup \{\top\}$.

   (a) Explain what does the element $\top$ represent and why we need it. Define the partial order and the meet operator for elements in this lattice (including $\top$).

   (b) Using this lattice to compute ranges of variables will fail. Explain why.

   (c) To solve the problems from part (b), we define a lattice $L' = \{[l, u] \mid l, u \in \{-\infty, -1, 0, 1, +\infty\} \wedge l <= u\} \cup \{\top\}$ (with the same partial order as before) and build a dataflow analysis that computes ranges in L'. Show the transfer functions for assignments of constants `x = n` and arithmetic operations `x = y + z` and `x = y * z`, where `x`, `y`, and `z` are program variables and `n` is an integer constant.

   Assuming programs that consist only of the three kinds of statements shown above, does this analysis from part (c) always yield the same result as the Meet-Over-Paths solution? If yes, show why. If not, show a counter-example program and indicate the MOP and dataflow solutions.