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

```
_foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp

        movl    16(%ebp), %eax
        addl    12(%ebp), %eax
        movl    %eax, -4(%ebp)
.L2:
        movl    8(%ebp), %eax
        movl    8(%ebp), %edx
        movl    4(%eax), %ecx
        movl    8(%edx), %eax
        cmpl    %eax, %ecx
        jge     .L5
        movl    8(%ebp), %ecx
        movl    8(%ebp), %edx
        movl    -4(%ebp), %eax
        addl    4(%edx), %eax
        movl    %eax, 4(%ecx)
        jmp     .L2
.L5:
        movl    8(%ebp), %eax
        subl    $12, %esp
        movl    (%eax), %edx
        addl    $32, %edx

        pushl   %eax
        movl    (%edx), %eax
        call    *%eax
        addl    $16, %esp
        movl    8(%ebp), %eax

        movl    %ebp, %esp
        popl    %ebp
        ret
```

   (a) Assuming that the stack grows downwards, draw the memory layout during the execution of this method. The layout must contain all of the pieces of memory that the execution of this method accesses.

   (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) Would it be safe to remove the instruction "`addl \$16, \%esp`" at the end of the program? Explain.

2. Consider the following quicksort method:

```
int partition(int[] a, int low, int high) {
   int pivot = a[low];
   int i = low;
   int j = high;
   int v;

   while (true) {
       while (a[i] < pivot) i = i+1;
       while (a[j] > pivot) j = j-1;

       if (i >= j) break;

       v = a[i];
       a[i] = a[j];
       a[j] = v;
       i = i+1;
       j = j-1;
   }

   return j;
}
```

   (a) Write the three-address code for this program. You code should translate array accesses of the form `v = a[i]` into sequences `t = i*4; v = v[t]`.

   (b) Construct the control flow graph from the three-address code. Use basic blocks to make the graph smaller.

   (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.

   (d) 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.