Question 1 (25 points): In the reverse engineering of a computer application, an important task is to write the source code for a corresponding segment of assembly code. You are working for a security agency and you have been asked to provide source-level code for the function `enigma` whose assembly code is below. The code is printed with line numbers to facilitate referencing to instructions in your answer.

```
(1) enigma:
(2)       add  $t0, $zero, $zero
(3)       slt  $t1, $t0, $a2
(4)       beq  $$t1, $zero, label_one
(4') label_two:
(5)       sll  $t2, $t0, 2
(6)       add  $t3, $a1, $t2
(7)       lw   $t4, 0($t3)
(8)       sll  $t5, $t4, 2
(9)       add  $t6, $a0, $t5
(10)      lw   $t7, 0($t6)
(11)      lw   $t8, 4($t3)
(12)      sll  $t9, $t8, 2
(13)      add  $t1, $a0, $t9
(14)      sw   $t7, 0($t1)
(15)      addi $t0, $t0, 1
(15')     blt  $t0, $a2, label_two
(16) label_one:
(17)      jr $ra
```

a. (**10 points**) Assume that this function follows the MIPS procedure-calling conventions. How many parameters does the function `enigma` has? Give a name for each parameter. You you will use these names in your source code. Also, indicate the type of the parameter (is it an address or a value? In the case of a value, can you say anything about the number of bits?) Justify your answer.

The function accesses registers `$a0`, `$a1`, and `$a2`. Therefore it has three parameters.
`$a2` is compared with register `$t0` which was initialized to zero. Thus it must be a value. It could have any number of bits from 1 to 32. We will call it `N`.
`$a1` is added to `$t2` at line 6, and the result (`$t3`) is then used as the base for a load. Thus `$a1` is probably the base of an array, and thus is a pointer. We will call it `C`
`$a0` is added to `$t5` in line 9, and the result (`$t6`) is used as the base for a load. Thus `$a1` is also probably a memory address. We will call it `A`

b. (**10 points**) How many memory loads and how many memory stores are executed by `enigma`? Your answer can be in terms of one or more of the parameters of the function.

There is a loop in `enigma` and each iteration of the loop executes 3 loads and 1 store. The loop executes `$a2` times (we called it `N` above). Thus $3 \times$ `N` loads and `N` stores are executed.

c. (**5 points**) Write C-style source code that leads to the generation of the assembly code above for `enigma`.

The first step is to write comments in the assembly code. We have given the following names to the parameters: `$a0 = A, $a1 = C, $a2 = N`. It is easy to see that `$t0` is the index for a loop, thus we will call it `i`.

```
(1) enigma:
(2)       add  $t0, $zero, $zero       # i <-- 0
(3)       slt  $t1, $t0, $a2           # if (i < N) then $t1 <-- 1 else $t1 <-- 0
(4)       beq  $$t1, $zero, label_one  # if ($t1 == 0) goto label_one
(4') label_two:
(5)       sll  $t2, $t0, 2             # $t2 = 4*i
(6)       add  $t3, $a1, $t2           # $t3 = C+4*i
(7)       lw   $t4, 0($t3)             # $t4 = Mem[C+4*i]
(8)       sll  $t5, $t4, 2             # $t5 = 4 * Mem[C+4*i]
(9)       add  $t6, $a0, $t5           # $t6 = A + (4 * Mem[C+4*i])
(10)      lw   $t7, 0($t6)             # $t7 = Mem[A + (4 * Mem[C+4*i])]
(11)      lw   $t8, 4($t3)             # $t8 = Mem[C+4*i + 4]
(12)      sll  $t9, $t8, 2             # $t9 = Mem[C+4*i + 4] * 4
(13)      add  $t1, $a0, $t9           # $t1 = A + Mem[C+4*i + 4] * 4
(14)      sw   $t7, 0($t1)             # Mem[A + Mem[C+4*i + 4] * 4]
                                       #    <-- Mem[A + (4 * Mem[C+4*i])]
(15)      addi $t0, $t0, 1             # i <-- i+1
(15')     blt  $t0, $a2, label_two     # if(i < N) goto label_two
(16) label_one:
(17)      jr $ra
```

One of the first things that we can do to simplify this code is to recognize that a reference to the memory position `Mem[C+4*i]` is represented in the C language as `C[i]`. Similarly, a reference to `Mem[C+4*i + 4]` is written in the C language as `C[i+1]`. With these replacements, we can rewrite the comments above as follows:

```
(1) enigma:
(2)       add  $t0, $zero, $zero       # i <-- 0
(3)       slt  $t1, $t0, $a2           # if (i < N) then $t1 <-- 1 else $t1 <-- 0
(4)       beq  $$t1, $zero, label_one  # if ($t1 == 0) goto label_one
(4') label_two:
(5)       sll  $t2, $t0, 2             # $t2 = 4*i
(6)       add  $t3, $a1, $t2           # $t3 = C+4*i
(7)       lw   $t4, 0($t3)             # $t4 = C[i]
(8)       sll  $t5, $t4, 2             # $t5 = 4 * C[i]
(9)       add  $t6, $a0, $t5           # $t6 = A + 4*C[i]
(10)      lw   $t7, 0($t6)             # $t7 = Mem[A + (4*C[i])]
(11)      lw   $t8, 4($t3)             # $t8 = C[i+1]
(12)      sll  $t9, $t8, 2             # $t9 = C[i+1] * 4
(13)      add  $t1, $a0, $t9           # $t1 = A + C[i+1] * 4
(14)      sw   $t7, 0($t1)             # Mem[A + C[i+1] * 4]
```

```
                                        #    <-- Mem[A + (4*C[i])]
(15)      addi $t0, $t0, 1              # i <-- i+1
(15')     blt  $t0, $a2, label_two      # if(i < N) goto label_two
(16) label_one:
(17)      jr $ra
```

Now we should apply the same transformations to the reminder memory references to obtain the following pseudo code:

```
(1) enigma:
(2)       add  $t0, $zero, $zero       # i <-- 0
(3)       slt  $t1, $t0, $a2           # if (i < N) then $t1 <-- 1 else $t1 <-- 0
(4)       beq  $$t1, $zero, label_one  # if ($t1 == 0) goto label_one
(4') label_two:

          A[C[i+1]] <-- A[C[i]]

(15)      addi $t0, $t0, 1             # i <-- i+1
(15')     blt  $t0, $a2, label_two     # if(i < N) goto label_two
(16) label_one:
(17)      jr $ra
```

Thus the C code for `enigma` is (versions of the code that use more temporary variables for storage of intermediate results, or use slightly different types, are also acceptable):

```
void enigma(int *A, int *C, int N)
{
 int i;

 for(i=0 ; i < N ; i++)
    A[C[i+1]] <-- A[C[i]]
}
```