Name:_____

Problem 1: (40 points) Number Representations

1. **Floating Point (10 points)**
   a. **(5 points)** Translate the decimal number 43.125 to a 32 bit IEEE 754 single precision floating-point number. Your answer should be a hexadecimal number.

   b. **(5 points)** Translate `0x12345678` (a 32-bit IEEE 754 floating point number) to a binary number in normalized scientific notation form. E.g. I want the answer in the form $1.100010001 \times 2^{345}$

2. **(15 Points) Hexadecimal to MIPS Conversion**
   a. **(5 points)** Translate the hexadecimal number `0x0004082A` to a MIPS instruction using register names, e.g. $s0, $t0, …

   b. **(5 points)** Translate the hexadecimal number `0x14200001` to a MIPS instruction using register names, e.g. $s0, $t0, …

   c. **(5 points)** The instructions in 2.a followed by the instruction in 2.b can be combined to form what pseudoinstruction?

Name:_____

3. **MIPS to Hexadecimal Conversion (15 points)**

The following code is taken from the first quiz. It slightly modified to eliminate pseudoinstructions and do some redundant jumping at the end (so that I can test you on j instruction). By now, you should know what it computes; however, that is not important for this problem.

```
      addi $t0, $0, 0       # originally li $t0, 0
      addi $v0, $0, 0       # originally li $v0, 0
moral:
      addi $at, $0, 4       # originally mul $t1, $t0, 4
      mult $t0, $at         # originally mul $t1, $t0, 4
      mflo $t1              # originally mul $t1, $t0, 4
      add $t1, $a0, $t1
      lw $t1, 0($t1)
      slt $at, $t1, $0      # originally blt $t1, $0, orel
      bne $at, $0, orel     # originally blt $t1, $0, orel
      addi $at, $0, 4       # originally mul $t2, $v0, 4
      mult $v0, $at         # originally mul $t2, $v0, 4
      mflo $t2              # originally mul $t2, $v0, 4
      add $t2, $a1, $t2
      sw $t1, 0($t2)
      addi $v0, $v0, 1
orel:
      addi $t0, $t0, 1
      slt $at, $t0, $a2     # originally blt $t0, $a2, moral
      bne $at, $0, short_jump
      j return
short_jump:
      j moral
return:
```

Convert the following instructions from the code above into 32 bit hexadecimal number. Assume that the address of the first instruction (`addi $t0, $0, 0`) is located at address `0x00400024`.

a. **(5 points)** `lw $t1, 0($t1)`

b. **(5 points)** `bne $at, $0, orel`

c. **(5 points)** `j moral`

Name:_____

Problem 2: (25 points) Arrays and Pointers: Bubble Sort

Bubble sort, also known as exchange sort, is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time, swapping these two items if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. head) of the list via the swaps.

```
void bubbleSort(int * numbers, int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (*(numbers + j - 1) > *(numbers + j))
      {
        temp = *(numbers + j - 1);
        *(numbers + j - 1) = *(numbers + j);
        *(numbers + j) = temp;
      }
    }
  }
}
```

Write the MIPS code corresponding to these 3 lines of code:

```
temp = *(numbers + j - 1);
*(numbers + j - 1) = *(numbers + j);
*(numbers + j) = temp;
```

Assume that *j* is stored at 0($sp) and temp must be in register $s0. You must follow all MIPS function call standards, and must only use real MIPS instructions (no pseudoinstructions)

Name:_____

Problem 3: (35 points) Recursion: Ackermann Function

In the theory of computation, the *Ackermann function* or *Ackermann-Péter function* is a simple example of a recursive function that is not primitively recursive. It takes two natural numbers as arguments and yields another natural number. Its value grows extremely quickly; even for small inputs, e.g. for Ackermann(4,3), the values of the Ackermann function become so large that they cannot be feasibly computed, and in fact their decimal expansions require more digits than there are particles in the entire visible universe.

The C/Java pseudocode for the Ackermann function is:

```
int Ackermann(int m, int n) {
      if (m == 0)
            return n + 1;
      else if (m > 0 && n == 0)
            return Ackermann(m-1,1);
      else if (m > 0 && n > 0)
            return Ackermann(m-1, Ackermann(m, n-1));
}
```

a. **(25 points)** Write the MIPS code, which directly translates the previous Ackermann function. You must follow all MIPS function call standards, and must only use real MIPS instructions (no pseudoinstructions)

Name:_____

b. **(2 points)** What is the result of Ackermann(1,2)?

c. **(4 points)** How many times does the Ackermann function get called with these input values (do not include the initial call i.e. Ackermann(1,2))?

d. **(4 points)** What is the maximum number of Ackermann frames on the stack at any point during the execution of Ackermann(1,2)?