Computer Science 237
Assignment 5
Due before lab next week

This week I would like you translate an insertion sort into assembly. You can find the starter kit on the web site as `sortie.tar.gz`.

Steps to completing this lab include:

1. Untar the starter kit in the usual manner:

   ```
   tar xvfz sortie.tar.gz
   ```

   This creates a subdirectory, `sortie`, that contains all of the usual files, including `sortie.c` and `isort.s`

2. If you type `make` it will make a working version of `sortie`. This is because I have given you the `C` code associated with the insertion sort routine I would like you to ultimately write. I have also included a copy of the code at the end of this document.

3. The program, in its current state, works. If you type

   ```
   sortie
   ```

   it prints the numbers that appear on the input, after they've been sorted. On the other hand, if you

   ```
   sortie -c
   ```

   it sorts the characters that appear on each line of the input. You should play with it and make sure that you understand how it works.

4. Carefully translate the routine `insertionSort` into an assembly language routines called `isort`. Your source should be stored in the file `isort.s`. The only symbol that you must make `.globl` is `isort`. All of the other symbols will only be referenced locally. You may find the notes, below, useful in performing this translation.

5. When you believe you have a working version of your assembly language code, you should modify the line

   ```
   sorter *theSort = insertionSort;
   ```

   to read

   ```
   sorter *theSort = isort;
   ```

   This is a *pointer to a function* that governs how sorting occurs in this utility. When you change the pointer value to `isort`, any calls to `theSort` become calls to your code. You might find the calls to `theSort`—they look pretty much like any other call. That's because a function is simply represented by a pointer to its first instruction.

6. Run your code. You should be able to enter a number of values, sort them. You should start by sorting zero values, then one, then (gosh!) two, etc. Do the same with the `-c` version of `sortie`, which should also work. You can test this against the working code on very large data sets with the commands

   ```
   sortie <d | diff - ds
   sortie -c <w | diff - ws
   ```

   that print differences between your output the expected output on a collection of numbers (`d`) or words (`w`).

7. When you are satisfied with your solution, turn it in:

   ```
   turnin -c 237 isort.s
   ```

A few notes on pointers and functions.

1. The `sortie` program references several different types of pointers including pointers to integers and pointers to characters. We use `void*` to mean "any type of pointer", but you must explicitly cast the pointer to the type you expect to manipulate, before you use it. You can see this, for example, in `intCompare`, which takes two generic pointers and casts them to pointers to integers.

   In your assembly language, the casting is unimportant. All pointers are the same to you.

2. Pointer arithmetic requires you to know the size of the objects pointed to before the addition (or subtraction) can be performed. That's why the `sorter` objects need to know the `size` of your data. This is a subtle point. If you don't understand it, see me or a TA before progressing.

3. Whenever you mention the name of a function (e.g. `intCompare`) it is simply treated as a pointer. This is why we can "pass" a function to the `sorter` type. In assembly, however, in our `call` instruction we need to make explicit that we're calling a function pointed to by, say, `%rax`. We use the following syntax to perform that call:

   ```
   call    *%rax
   ```

   The asterisk tells the machine to "call a function at this absolute address" as opposed to an address that is specified as a *relative* offset from the current instruction pointer. This, too, is subtle.

The code you need to convert.

```
// A great sort for almost sorted data.
void insertionSort(void *data, int n, int size, compareFun *c)
{
  void *left, *right;
  int nm1 = n-1;
  if (n > 1) {
    insertionSort(data,nm1,size,c); // do most of sorting
    right = ((char*)data) + nm1*size;  // full, unsigned multiply
    left = ((char*)right) - size; // next to right
    while (left >= data && c(left,right) > 0) { // short circuit
      memswap(size,left,right);
      left = ((char*)left) - size;
      right = ((char*)right) - size;
    }
  }
}
```