Susmitha Shailesh
I pledge my honor that I have abided by the Stevens Honor System.

Complication in converting from C to assembly:
This was the pseudocode I based my assembly code off of:

```
MERGESORT(A, lo, hi)
 1. if lo < hi then
 2.     mid = lo + (hi - lo) / 2
 3.     MERGESORT(A, lo, mid)
 4.     MERGESORT(A, mid+1, hi)
 5.     L = lo
 6.     H = mid + 1
 7.     for k <- lo to hi
 8.         do if L <= mid and (H > hi or A[L] <= A[H])
 9.             then scratch[k] = A[L]
10.                 L <- L + 1
11.             else scratch[k] = A[H]
12.                 H <- H + 1
13.     for k <- lo to hi
14.         do A[k] <- scratch[k]
```

Writing this pseudocode in C was not difficult. However, converting it to assembly code was very challenging. When C code compiles into assembly, it does not do it efficiently. It adds a lot of its own security features, etc. that makes the code long and unrecognizable. The assembly version of the compiled C code is much less readable than just manually coding in assembly. Compiling C would also use the stack in a different manner than how I used it. The biggest challenge I had as I was manually converting from C to assembly was the inability to do function calls with parameters. In order to do the function with different values than before, you would have to change those register values in memory, causing you to lose the old values. This is an issue for recursion. Another challenge I had was figuring out how the stack works. Although similar for the most part to stacks in C, minor differences made them tricky.

How the merge sort recursion work:

The idea behind implementing mergesort recursively is to split the array to be sorted in half and call mergesort on each half of the array until the array is broken into many single-element arrays. Then, as the smaller arrays are put back together, they are sorted so that once the original array is re-formed, it is in sorted order. My mergesort recursion uses two main functions, sort and merge, to accomplish this task. The sort function performs lines 1 to 4 of the pseudocode. It begins by calculating the value of mid and pushing the indices (lo and hi) of the original array, as well as the values of mid and the current link register, onto the stack. Then, it checks the base case: if lo >= hi, it ends that branch of recursion, because that means that there is only a single element left in the array. It then changes the hi value to equal mid, or in other words changing the indices to the indices of the left sub-array. It calls sort on these values. Afterwards, it loads the original index values of the parent array and changes the lo value to equal mid+1, or in other words changing the indices to the indices of the right sub-array. It repeats the aforementioned process for the right sub-array. After completing the sort function, the program moves on the merge portion. The merge portion performs lines 5 to 14 of the pseudocode. Merge sorts the array into the scratch array. It iterates through the array with a for loop. At the end, when the scratch array is copied to the original array for the last time, the numbers in the original

array should be sorted. This implementation of recursive mergesort, with an array and a scratch array, is very memory-efficient.

How the stack has been used in your program:

In my program, the stack stores two pairs of values: the lo and hi indices for each recursive call of mergesort as well as the mid value and the link register value for that call. These values are always pushed on in a pair of pairs, and always popped off in a pair of pairs. For every time a recursive call is pushed onto the stack, it is popped off later in the code when the values in that recursive call are needed to perform the next recursive call, i.e. the indices of the parent array are popped off the stack and used to make the next recursive call, which is on the right-half of the parent array.

How you are using every register (please mention what registers you are not using):

X0: lo index, stores string in print	X1: hi index, stores number in print	X2: mid index
X3: unused	X4: unused	X5: unused
X6: unused	X7: unused	X8: unused
X9: stores temporary values used for calculations, loads, and stores	X10: stores temporary values used for calculations, loads, and stores	X11: k, iterator for various for loops
X12: L in merge portion	X13: H in merge portion	X14: first element of scratch
X15: first element of array	X16: unused	X17: unused
X18: unused	X19: unused	X20: unused
X21: iterator for for loop in print	X22: unused	X23: unused
X24: unused	X25: unused	X26: unused
X27: unused	X28: unused	X29: stores temporary value used for a load in print

X30 is used as the link register.