CIS 435 Programming Exercise Module 2 (30points)

1 What to turn in

Follow the guidelines of Handout 10 dated January 23, 2004. Submissions that deviate from these guidelines will be assigned 0 points.

2 What to implement

Implementation is required for the function described in part A.

3 Part A: Implementation of Merge Sort (20 points)

Provide an implementation of a merge sort algorithm with the following syntax and behavior.

```
void merge_sort(void *keys, int n, int size, int (*compare) ( ) );
```

`keys` is a pointer to the input array. Each element of the array is a datatype whose length in bytes is `size`. The length of the array (input size) is `n`. `compare` is a pointer to a function that returns an integer. Its two arguments are pointers to `void` as well. Depending on whether the first argument of `compare` is greater, equal, or less than the second, `compare` returns a positive, 0, or negative number. The parameters of `merge_sort` are similar to those of the ANSI C standard library function `qsort` (Review the evaluation quiz as well).

Your algorithm may be recursive, as the one presented in the textbook, or iterative. If it is recursive, make sure it works for the problem instances of part B.

4 Part B: Experimental Results (10 points)

Run your implementations with the testing functions provided in `main()` of `sortg.c` in `testing.tar` on 4 different data-sets and 5 problem sizes.

1. Use the following problem sizes:
   1. $n = 2000$.
   2. $n = 8000$.
   3. $n = 32000$.
   4. $n = 128000$.
   5. $n = 512000$.
2. The four different data sets consist of the following test instances.
   1. An array of integers where all the element are the same (say n).
   2. A sorted array of integers where the i-th element of the array is i.
   3. A reverse sorted array of integers where the i-th element of the array is $n - i$.
   4. An array whose elements are randomly chosen using function `random` (see `sortg.c` on how to setup such an array).

Describe in tabular form the running time of your implementation for each input instance. A timing of the execution of any function can be obtained similarly to the one provided in `sortg.c`

Remarks.

The table to be reported for part B should be included at the end of the submitted source code file as comment.

Make sure that your algorithm works for all requested problem instances. It is conceivable, if you use a purely recursive implementation, that you may run out of space. In this case you are allowed for small size subarrays (say of size less than 32) to use insertion sort to sort such small arrays of keys. Alternatively, you may replace recursion by iteration.

If you think the running time for a test instance takes more than a reasonable amount of time (say one or two minutes), try to extrapolate the running time for that problem size and input and indicate so in the compiled table.