CIS 435 Programming Exercise Module 1 (30points)

1 What to turn in

Follow the guidelines of Handout 10 dated January 23, 2004. Handout 10, can be found in the handout section of the Web-page. The source code cited in that handout can also be found in the handout section of the web-page, at the very bottom of it. 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 Insertion Sort (20 points)

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

```
void ins_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 (eg. 1), zero (eg. 0) or negative (eg. -1) number. The parameters of `ins_sort` are similar to those of the ANSI C standard library function `qsort` (Review the evaluation quiz as well). The following function describes a sorting algorithm; the arguments to `bubble_sort` are identical to those of `ins_sort`.

```
void bubble_sort(void *akeys, int n, int size, int (*compare) ())
{
        register int i,j,k;
        char    *x;
        char    *keys;

        keys= (char *) akeys;
        x = (char *) malloc(size*sizeof(char));
        for(i=1;i<n;i++) {
                k=i;
                memcpy(x,&keys[i*size],size);
                for(j=n-1;j>=i;j--) {
                        if (compare(&keys[(j-1)*size],&keys[j*size]) > 0) {
                                memcpy(x,&keys[(j-1)*size],size);
                                memcpy(&keys[(j-1)*size],&keys[j*size],size);
                                memcpy(&keys[j*size],x,size);
                        }
                }
        }
        free((char *)x);
}
```

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 4 problem sizes.

1. Use the following problem sizes:

    1. $n = 500$.
    2. $n = 2000$.
    3. $n = 8000$.
    4. $n = 32000$.

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.

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.