1 What to turn in

Follow the guidelines of Handout 2. Handout 2, can be found in the handout section of the Web-page. The source code cited in that handout or below can also be found in the handout section of the web-page. Submissions that deviate from these guidelines will be assigned 0 points.

2 What to implement

Implementation is required for the function(s) described in part A.

3 Part A: Implementations of Insertion Sort and Merge Sort (60 points)

Provide an implementation of an insertion sort sort algorithm with the following syntax and behavior. Provide an implementation of a generic (and possibly recursive) merge-sort algorithm. Also provide a variant of that merge-sort algorithm that sorts subsequences of length less than a threshold `THRESHOLD` using `ins_sort` or `bubble_sort` instead.

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

#define THRESHOLD 12

void mrg_sort (void *keys, int n, int size, int (*compare) (const void *, const void * ) );
void mrgT_sort (void *keys, int n, int size, int (*compare) (const void *, const void * ) );
```

`keys` : is a pointer to the input array.
`n` : is the dimension (number of elements) of the array.
`size` : is the length in bytes of the datatype of each element of the array.
`compare` : is a function/pointer to a function that returns an integer. Its two arguments are pointers `const void *` as well. Depending on whether the first argument of `compare` is greater, equal, or less than the second, `compare` returns a positive (e.g. 1), zero (e.g. 0) or negative (e.g. -1) number.

The parameters of `ins_sort , mrg_sort, mrgT_sort` are similar to those of the ANSI C standard library function `qsort` also cited in Handout 3 (pages 2 and 3). Your algorithm for `mrg_sort` 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 and does not abnormally stop because of space limitations. For `mrgT_sort`, if the keys to be sorted in a recursive (or otherwise) call are less than `THRESHOLD` use `ins_sort` (or `bubble_sort` if you skip PA1) to sort them, otherwise use merge sort. In other words, `mrgT_sort` will be a slight variant of `mrg_sort`; it saves times to do `bubble_sort` or `ins_sort` to sort 2 or 3 keys instead of calling the merge-sort algorithm recursively. A representative value for `THRESHOLD` is given as a C/C++ preprocessing directive. In testing we might change the value to something smaller or slightly larger (eg. 8 or 16).

The following function describes a sorting algorithm; the arguments to `bubble_sort` are identical to those of the three functions above.

```
void bubble_sort(void *akeys, int n, int size, int (*compare) (const void *, const void *))
{
        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 (20 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. $n = 8192$.
   2. $n = 16384$.
   3. $n = 32768$.
   4. $n = 65536$.
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, $i = 1, \ldots, n$.
   3. A reverse sorted array of integers where the i-th element of the array is $n - i + 1$, $i = 1, \ldots, n$.
   4. An array whose elements are randomly chosen using function `random` (see `sortg.c` on how to setup such an array).

3. 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 beginning of the submitted source code file as comment, after your name.

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 from previous ones and indicate in the compiled table that the corresponding figure is an estimate and not an actual figure by putting a $*$ one the left of it. **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. You may wish to implement a non-recursive (i.e. iterative) mergesort instead.**