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 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 functions described in part A.

3 Part A: Implementation of Quick-Sort and Heap-Sort (60 points)

Provide an implementation of a quick-sort algorithm with the following syntax and behavior. The `quick_sort` function is the one described in CLRS, the textbook. Do the same for heap-sort as described in CLRS. You are also required to implement `hoare_sort` which is described in Problem 7-1 of the textbook.

```
void quick_sort (void *keys, int n, int size, int (*compare) (const void *, const void * ) );
void hoare_sort (void *keys, int n, int size, int (*compare) (const void *, const void * ) );
void  heap_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 `quick_sort` and `heap_sort` are similar to those of the ANSI C standard library function `qsort` also cited in Handout 3.

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. $n = 4096$.
   2. $n = 16384$.
   3. $n = 100000$.
   4. $n = 1000000$.

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.

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. Then you should reconsider the recursive implementation of quicksort or heap-sort (eg. Heapify).