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 a Select function (60 points)

The purpose of Part A is to implement a `Select(i)` function for returning the i-th smallest of n keys in three different ways as described below.

- Sort the keys using quicksort and then return the i-th key of the sorted sequence.
- Use the Randomized Selection algorithm as described in the textbook.
- Use the worst case linear time selection algorithm described in the textbook (the one that splits keys into groups of five and so on).

Thus provide the implementation of the following three functions respectively.

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

int  randSel (void *keys, int n, int stat, int size, int (*compare)(const void *,const void *));

int  detlSel (void *keys, int n, int stat, 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 key pointed by 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.
`stat` : is the index between 1 and n inclusive of the statistic that will be returned by one of the three functions.
If the value returned by any of the three functions is t, this would imply that the requested statistic is stored at the address starting at location $\&keys[t * size]$. The order of the elements of `keys` might change during the execution of the function.

4 Part B: Experimental Results (20 points)

Run your implementations with the testing functions provided in `main()` of `sortg.c` in `testing.tar` on the following data-sets and problem sizes.
1. Use the following problem sizes:

    1. $n = 100000$.
    2. $n = 1000000$.
    3. $n = 10000000$.

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 implementations 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 implemented quicksort in a prior homework you could use one or the other of the two implementations. If you did not do that homework, you can use the Standard C library function `qsort` *whose description is available on AFS if you do a* `man qsort`*, or also available on the Visual C++ environment through the help system available there.*