In this homework, we will study the problem of selecting the k largest numbers in an array of n integers. Three alternative algorithms that solve this problem are discussed below. Each algorithm has a different time complexity. The goal of this homework is to simulate the growth rates of all three algorithms using different inputs.

Algorithm 1 Finding the maximum number in an array can be done in $O(n)$ time. We can simply find the maximum value in the array, move it to the beginning of the array, and continue doing this k times until the k largest numbers are in the first k positions in the array. This algorithm runs on $O(kn)$ time. A pseudocode is given in the Wikipedia entry on selection algorithms (`http://en.wikipedia.org/wiki/Selection_algorithm`) under the section "Nonlinear general selection algorithm".

Algorithm 2 You can sort the input array in descending order, and select the first k numbers in the sorted array. The overall complexity of the algorithm will be dominated by the complexity of the sorting algorithm because k numbers can be selected in $O(k)$ time once the array is sorted. You can use an efficient sorting algorithm (such as Quicksort) that runs in $O(n \log n)$ average time so that the average time complexity of the overall algorithm is also $O(n \log n)$.

Algorithm 3 There is a linear time algorithm for finding the k largest numbers. Read the section titled "Finding the k'th smallest item of an array" in Chapter 2 of the Carrano book and the section titled "Partition-based general selection algorithm" in the Wikipedia entry on selection algorithms (`http://en.wikipedia.org/wiki/Selection_algorithm`) for the description of an algorithm called *Quickselect* that has an average case time complexity of $O(n \log n)$. Then, read the section titled "Linear general selection algorithm - Median of Medians algorithm" in the Wikipedia entry and the pseudocode and detailed analysis by Prof. David Eppstein at `http://www.ics.uci.edu/~eppstein/161/960130.html` for a linear ($O(n)$) worst case time algorithm.

Assignment:

1. Study all of the algorithms above using the given references, and understand how the upper bounds are found for the running time of each solution.

2. You are given the implementation of each algorithm (on the course web site) with the following prototypes

```
void solutionX( int input[], const int n, const int k, int output[] );
```

   where X is the algorithm number ($X = 1, \ldots, 3$). Each solution requires that the input array and the output array are allocated with sizes n and k, respectively, and the input array is filled with n integers by the calling function. The solution function finds and stores the k largest integers in the output array. Study all of these solutions.

3. Write a driver (`main`) function that generates an array that contains random numbers and calls each of these solutions with that array as input. Then, run each solution on a computer and record execution times when different input sizes are used. You can fix k (e.g., around 1000) and vary n within a large range. You are expected to try many different input sizes, both small inputs and very large inputs and observe the effects of different growth rates.

4. Use these results to generate a plot of running time (y-axis) versus the input size n (x-axis). Specifically, you are expected to produce a plot as in Figure 2.3 of the handout chapter on algorithm analysis. (Make sure that you give exactly the same array as input to each solution. Note that the contents of the non-constant parameters (both `input` and `output` arrays) are modified inside each solution function.)

5. Provide the specifications of the computer you used to obtain these execution times. You can use any computer with any operating system for this assignment.

6. Plot the expected growth rates obtained from the theoretical analysis (as given for each algorithm above) by using the same `k` and `n` values that you used in your simulations.

7. Compare the expected growth rates and the obtained results, and discuss your observations in a paragraph.

You can use the following code segments to compute the execution time of a code block:

```
// Store the starting time
double duration;
clock_t startTime = clock();

// Code block
...

// Compute the number of seconds that passed since the starting time
duration = 1000 * double( clock() - startTime ) / CLOCKS_PER_SEC;
cout << "Execution took " << duration << " milliseconds." << endl;
```

For these operations, you must include the `ctime` header file.

An alternative code segment is as follows:

```
// Declare necessary variables
std::chrono::time_point< std::chrono::system_clock > startTime;
std::chrono::duration< double, milli > elapsedTime;
// Store the starting time
startTime = std::chrono::system_clock::now();
// Code block
...
// Compute the number of milliseconds that passed since the starting time
elapsedTime = std::chrono::system_clock::now() - startTime;
cout << "Execution took " << elapsedTime.count() << " milliseconds." << endl;
```

For these operations, you must include the chrono header file.