Objectives:
In this lab, you will learn how to measure the running time of an algorithm and compare the growth of the running times of different algorithms.

Deliverables:
- (10%) You have to attend the lab on Monday.
- (20%) The final code after running all the steps.
- (50%) A final report that includes the answers to all the questions below, the filled-in table below, and the log-log plot that you create in step 13.
- (20%) There will be a relevant question during the lab on Monday 10/9 that you will be asked to do during the lab.

Groups:
- This lab should be done individually.

Due date:
- The deliverables are due on Tuesday 10/10 by 11:59 PM Pacific Time. However, you are highly encouraged to deliver it during the lab on Monday 10/9 to save your time.

Important Note:
- You will need to carry out this lab at home to be able to perform the task that will be given to you during the lab on Monday 10/9. The official due date is on Tuesday just in case you have questions for your TA during the lab.

Steps:
1. Get the code in the file ‘time-measure.cc’ and run it on your favorite IDE or from command line. What does this initial code do?
2. Collect the running times for generating a random list of sizes 10 up to 10,000,000 entries. Fill them in the table below. (Hint: You can copy this table to a spreadsheet application to help you with the following steps.)

<table>
<thead>
<tr>
<th>List size</th>
<th>Randomize</th>
<th>Std::find</th>
<th>Insertion Sort</th>
<th>Std::sort</th>
<th>Std::binary_search</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Move the ‘generate_random_numbers()’ call before the stopwatch block so that its time is not measured. Inside the stopwatch block, write a code that generates one number and searches for it in the list. (Hint: Use rand() to generate a number and std::find() to search for it.)
4. Run the code again for lists of sizes 10 to 10,000,000 and fill in the table above under std::find.
5. Do you think the `std::find` algorithm will most likely act in best-case, average-case, or worst-case scenario? Why?

6. Implement the insertion sort algorithm as shown in the class. Convert the pseudo code shown in the slides to C++. Do not perform any optimization techniques; just make a line-by-line translation.

7. Comment the lines you added in Step 3 and add a new line that sorts the list using your insertion sort algorithm.

8. Fill in the table above under insertion sort. (Hint: Since insertion sort takes a long time, you can fill in that column up-to a list size of 100,000.)

9. Replace your insertion sort algorithm with a call to the STL `std::sort` algorithm. Measure the running times and record the numbers above. (Hint: This time, you can actually run the sort algorithm up-to a list size of 10,000,000.)

10. Move the `std::sort` call before the stopwatch block to avoid measuring its time.

11. Inside the stopwatch block, add a code that generates a random number and searches for this number using `std::binary_search` algorithm. Measure the running times of up-to 10,000,000 entries in the list.

12. By looking at the numbers in the table, categorize these algorithms based on how the numbers grow with the list size. (Hint: There are four categories based on the growth rate of the algorithms.)

13. Draw a chart for all these algorithms where the x-axis represents the list size and the y-axis represents the running time. Plot one line for each algorithm in that chart. Make it a log-log plot so that you can better see the growth. How does the figure match with your categorization in Step 12.

14. What does the line ‘`srand(0)`’ do? Why do you think it can be useful to have this line?