# CMPSC122 In-Lab 13 – BST and Sorting

Submit the solution files for the exercises online before the due date/time. For exercise 1, write your answer in the docx file directly and submit. For exercise 2, upload the modified sorting.cpp. For exercise 3, upload the modified BSTSelect.cpp.

**Exercise 1: (Non-Programming Exercise.)**

Consider selecting pivots from arrays using the **median-of-three** rule. For each of the given arrays, determine which element will be chosen as the pivot.

(1) Array: `8 15 14 7 19 5 22 30 1`
Pivot should be: __**8**____

(2) Array: `10 12 6 2 18 3 9`
Pivot should be: __**9**____

(3) Array: `12 20 50 40 15 22 10`
Pivot should be: __**12**____

**Exercise 2: (Non-Programming Exercise.)**

In this task, you will conduct experiments to compare the performance of insertion sort and quicksort when used to sort different kinds of arrays.

Download the file **sorting.cpp** from Canvas. The file includes my implementation of the insertion sort and quicksort algorithms we have covered in the lecture. Note, I made some modifications to these algorithms so that they also count the number of element comparisons they made during the sorting process. When evaluating the running time of sorting algorithms, one of the commonly used standards is the number of array element comparisons made.

The main function of sorting.cpp file uses both insertion sort and quicksort to sort a same testing array and show how many elements comparisons are required for both algorithms. Such tests give us some idea of the performance of insertion sort and quicksort when used to sort different kind of arrays.

**Your task**:

1. Provide an example array **of size at least 20** where insertion sort takes less comparison to finish the sorting.

2. Provide another example array **of size at least 20**, where the number of comparisons of quicksort is **less than half** of the number of comparisons of insertion sort.

**Submission:** You have two example arrays at the end of the sorting.cpp file as a comment. Also, an execution sample for each of your example arrays.

**Exercise 3: Selection in BST**

Write a function

```
int BSTSelect(BinNode* root, int k);
```

This function should return the **k-th smallest key** value in a given BST. (When k is 1, the function returns the smallest key of the BST. When k is 2, it returns the second smallest, and so on). We assume that the keys in the BST are integers and `k` is always positive.

If `k` is greater than the size of the tree, return INT_MAX (which is a constant defined in the `<climits>` header).

Note that the BinNode class has a data field called `subTreeSize`, which tells the number of nodes in the subtree rooted at that node. **Make sure to use this information,** as it would greatly help design an efficient algorithm. A file for testing is provided on CANVAS named BSTSelect.cpp. Submit the filled-in file.