Practical 7 - Searching and Sorting

Due Monday by 23:59 **Points** 7 **Submitting** an external tool

General Instructions

All submissions for the practical assignments should be under version control. Review practical 1 for setting up your repository for this practical.

Submission is through Gradescope.

If you get stuck on one of the hidden test cases and really cannot resolve it yourself, please feel free to ask the practical tutors for guidance as to how to fully test your code. You will need to explain what testing you have already done (boundary cases, unit testing, etc)

We encourage you to finish your work before the practical session and take the session as consulting time.

Problem Description

Objective

This practical will test your knowledge of sorting and searching algorithms. In this practical, you will be implementing a number of algorithms. Each of these algorithms will be constructed in a different class. You should make sure that you know the complexity of the algorithms you implement.

Design

Think of how you are going to solve the problem and test your implementation with the test cases you designed based on the stages below.

Testing Hint: it's easier if you test things as a small piece of code, rather than building a giant lump of code that doesn't compile or run correctly. As part of your design, you should also sketch out how you are going to build and test the code.

Sorting

Implement a base class called **Sort**. All other sorting algorithms must be derived from this class. You should decide how you are going to store the data that is going to be sorted or worked with as part of your sorting process. It is up to you to decide the storage mechanism. The Sort class must have a function, called **sort**:

```
std::vector<int> sort(std::vector<int> list)
```

Step 1: Implement bubble sort in a class called **BubbleSort** that extends **Sort**
http://en.wikipedia.org/wiki/Bubble_sort

Step 2: Implement quick sort in a class called **QuickSort** that extends **Sort**
http://en.wikipedia.org/wiki/Quicksort

As a requirement, for the pivot selection, when the list is of length at least 3, please always chose the third value in the list (e.g., if the (sub)list is formed by 4 elements 2, 4, 9, 1, then the pivot is 9). Do not naively copy an algorithm from the wikipedia page. Consider how the requirement above affects what algorithm(s) you can use.

Optional challenge (students enrolled in 7103 MUST complete ONE of these challenges):
- Implement quick sort without using recursion
- Implement quick sort without using any extra lists

Searching

Implement recursive binary search in a class called **RecursiveBinarySearch.** In recursive binary search, the parameter list should include a sorted list of elements, an element to be searched, a starting point and an ending point. You need to consider how you will implement this approach while still providing the interface to the user that only requires the list and item to be searched for.

When we search, we examine the middle element of our array. If we find the element we are looking for, return the index of the element. If the element we are looking for has value less than the middle element, call binary search with the same starting point and an ending point one position less than the current middle index. If the element we are looking for has a value greater than the middle element, call binary search with the same ending point and a starting point one greater than the current middle index.

RecursiveBinarySearch must provide the function:

```
bool search(std::vector<int>, int);
```

Main function

The test script will compile your code using
g++ -o main.out -std=c++11 -O2 -Wall *.cpp

It is your responsibility to ensure that your code compiles on the university system. g++ has too many versions, so being able to compile on your laptop does not guarantee that it compiles on the university system. You are encouraged to debug your code on a lab computer (or use SSH).

Create a main function that takes in a list of integers (one line, separated by space) int1 int2 int3 ... intn.
In the main function, the list should be sorted in ascending order first using quick sort, then binary search is used to determine whether 1 belongs to the list or not. Your output should start with either “true” (1 belongs to the list) or “false”, and then followed by the sorted list. Please separate the results using space.

Sample input: 1 3 5 4 -5 100 7777 2014
Sample output: true -5 1 3 4 5 100 2014 7777

Sample input: 0 3 5 4 -5 100 7777 2014
Sample output: false -5 0 3 4 5 100 2014 7777

Files to be submitted:

Sort.h
BubbleSort.h
BubbleSort.cpp
QuickSort.h
QuickSort.cpp
RecursiveBinarySearch.h
RecursiveBinarySearch.cpp
main.cpp

Marking Scheme

- Functionality (7 marks):
  - Matches specified interface/Follows OO design principles (1 mark)
  - Passing public test cases (3 mark)
  - Passing hidden test cases (3 mark)

Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

[✓ GitHub] Bitbucket

REPOSITORY

Select a repository...

BRANCH

Select a branch...

Upload Cancel