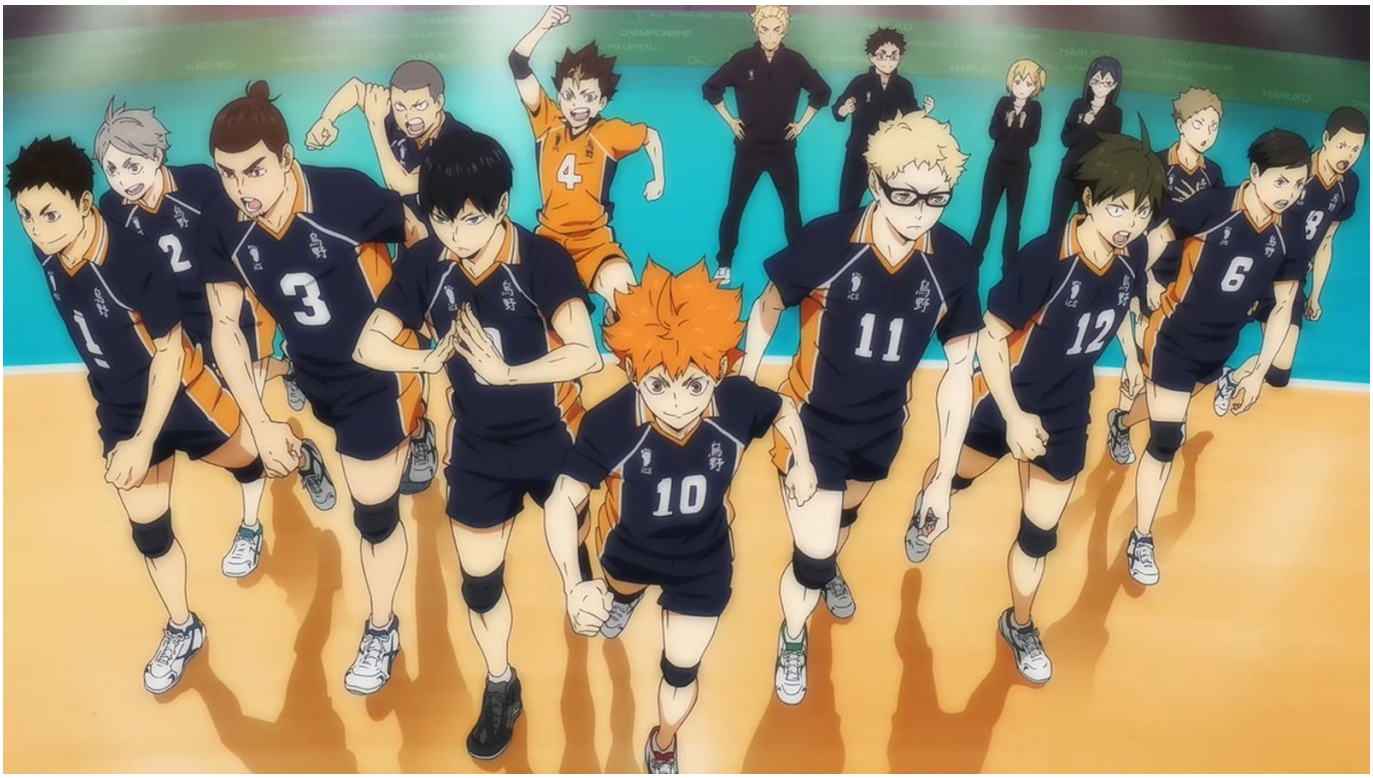
Project 2: Let's Sort It Out

Instructor: Sadab Hafiz

Due: July 04, 2024



Introduction

We saw how different sorting algorithms work and how each have their pros and cons. It is now time for you to get your hands dirty and implement the algorithms. In this project, you will implement an abstract class `SortingAlgo` which will serve as a blueprint for the five sorting algorithms we talked about in class. Furthermore, you will track the number of swaps and comparisons for each algorithm. The goal of this project is to familiarize students with abstract classes and sorting algorithms. Furthermore, it introduces the idea of a comparator function that can be used to sort in both ascending and descending order.

Submitting through GitHub

Assuming you did project 1, you should have a GitHub account. Furthermore, you should now be comfortable submitting assignments on Gradescope from GitHub. If not, watch this video which shows the entire process, from accepting an assignment to submitting it on Gradescope. **The link to accept the GitHub Classroom assignment can be found on Blackboard**.

Documentation and Design

Documentation and design is worth 20% of each project like spring semester. The requirements are:

Documentation Requirements (15%)

- All files should start with block comments with at least your name, date, and a brief description of the code implemented in that file (a rough idea of what this document is).
- All functions in the `.hpp` files should have a comment describing the function:

```
/**
 * @brief: describe the function in general and what it does
 *
 * @pre: describes any precondition
 * @param: one for each parameter the function takes
 * @return: describes the return type
 * @post: describes any postconditions
 * @throws: describe any exceptions that the function may throw
 */
```

  For the required functions in each project, these will be provided to you in the instructions. However, you have to write them for any additional helper functions that you add and implement.

- While function comments are descriptive, they are intended to let users of the class know how to use the functions. Thus, they don't go into the details of the implementation. In order to explain the implementation, you should add inline comments where necessary (loops, conditionals, etc.):

```
// comment explaining what is happening in the for-loop
for (DataType x: some_arr_){
    // comment explaining the if-else statements
    if (...x meets some condition...){
        ...some code in here...
    }
    else{
        ...some code in here...
    }
}
```

  If something feels necessary to be explained to someone looking at your code, you should leave an inline comment explaining it.

- Avoid commenting every single line. Things that are self-explanatory should not be commented.

Design Requirements (5%)

- You must follow the naming convention discussed in class.

```
string my_variable;
class MyClass{};
MyClass class_instance;
string my_data_member_;
void myFunction();
const int MY_CONSTANT;
```

- Any decisions you make, make sure to be consistent. This applies to the choice of include guards (`#pragma` or `#ifndef`), the order of things in classes, comments, code formatting, etc.
- Avoid making decision choices that violate OOP principles. For example, `public` data members and global variables are to be avoided unless explicitly mentioned. Things should be `protected` only if the class will have other classes inheriting from it.
- For any helper functions, the keyword `const` and `&` should be used where appropriate. For example, if your function doesn't modify the private data members, the functions should be defined as `const`.
- Do not include `using namespace std;` in the header files. Avoid using it altogether.
- Include `.cpp` files at the end of class template headers. Libraries should be included in the `.hpp` files.
- As long as you follow the standard practices that are demonstrated in class, you will not lose points.

Helpful Tips

If you are using vscode, there are extensions that can help you. The documentation generated in the header files I share use an extension that generate comments compatible with Doxygen. Doxygen is a tool that provides robust support for documenting `C++` code. Here are the extensions that I highly recommend you to use to automate some documentation generation:

- Doxygen Documentation Generator : Generate Doxygen compatible comment templates
- `C/C++` for VSCode : VSCode language support for `C/C++` editing and debugging

If you don't have a working `C/C++` environment, contact me as soon as possible. The starter files show you how you can document your files. Make sure your final submission is documented and formatted properly.

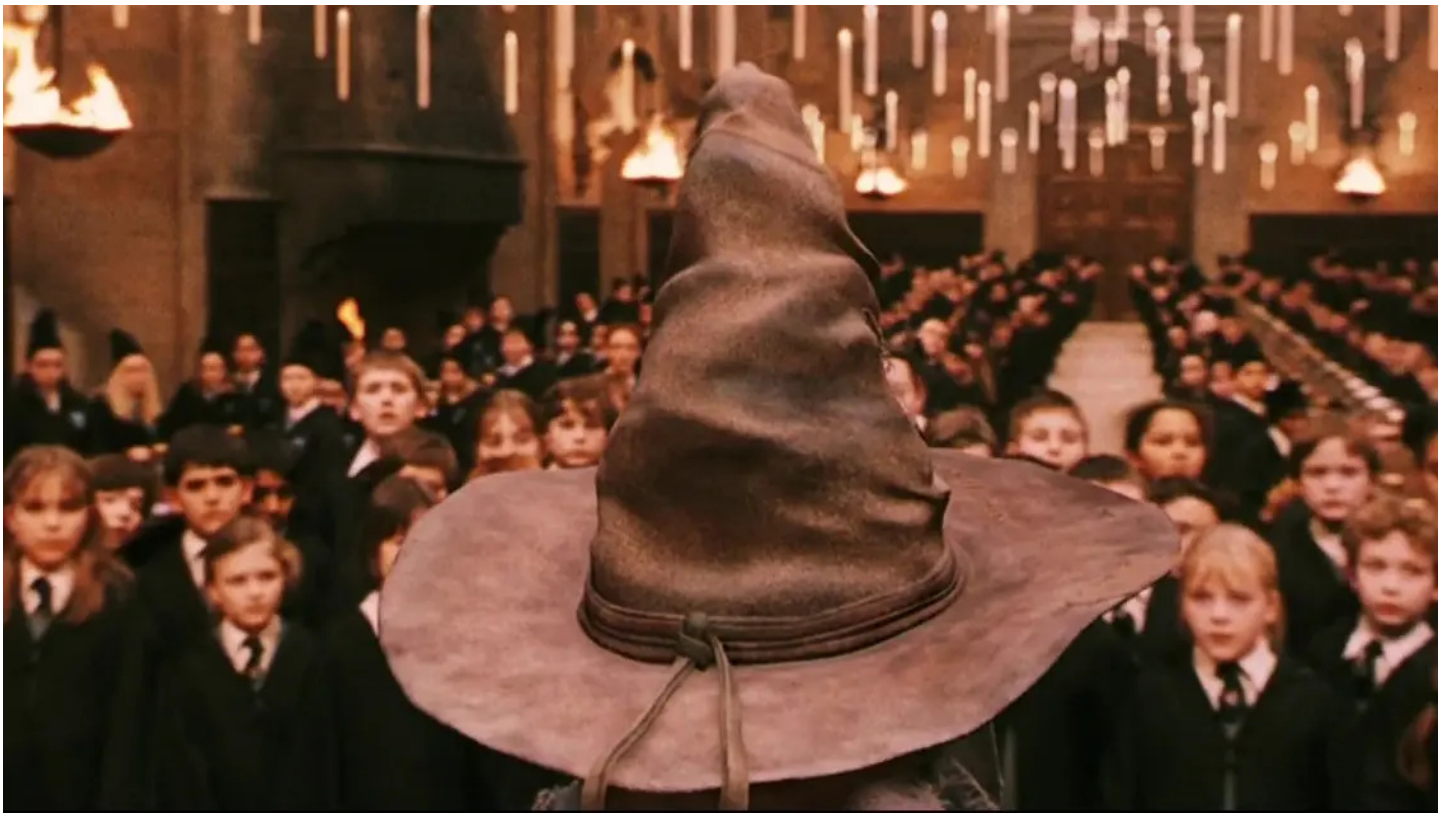
Starter Code Tips

In the provided files, make sure you address each `TODO` comment and delete the comments after addressing them. If you plan to modify a function that is explicitly stated not to be modified, make a copy of the original. You can also look at the original starter files from GitHub commit history.

Plagiarism and Citing

Your code in all the `.cpp` files should not 100% match with another student. If that happens, I will assume it is a violation of academic integrity. It can happen with the use of generative AI, it can also happen if you both copy code from the same source. If you use tools or functions that are not mentioned in the slides, or I didn't teach in class, you have to cite them with an inline comment. You should also be able to explain such lines of code if I ask you to explain what you wrote.

SortingAlgo Class



Your first task is to implement a class called `SortingAlgo`. The header file is in the starter code, **which you can access through the GitHub classroom link in Blackboard**. Carefully go through each of the function comments to figure out how to implement them. Some functions can be used to implement the other ones, so I encourage you to go over the header entirely first before starting to code. The `.cpp` file is also provided with the implementation for the `comparator` function. Take a look at the comment of this function and understand what it is doing. You will need to get used to using this function if you want to complete this assignment successfully. **Do not add any helper functions to this class. Do not modify the function prototypes or data members of this class. Do not modify the comparator function's implementation. Doing so will result in a 0 in the assignment!** In the derived classes of `SortingAlgo`, **use the algorithms from the slides to make sure that your total swaps and comparisons will match with what Gradescope is expecting**.

SelectionSort

The `SelectionSort.hpp` and `SelectionSort.cpp` files are provided in the starter code. The implementation for `findMinMax` function is provided. `Do not modify this function`. This function gets the min if we are sorting in ascending order, and it gets max if we are sorting in descending order. Take a look at its implementation and understand how the comparator is being used. In addition to that, notice that `comparisons_` is being incremented for each comparison being made. In your implementation for `sort` function, make sure you increment the `comparisons_` and `swaps_` data members appropriately to track the total number of swaps and comparisons being made.

MergeSort and QuickSort

Similar to `SelectionSort`, the header and implementation files for `MergeSort` and `QuickSort` are provided in starter code. In addition to that, the code for `merge` and `partition` functions are provided. These functions are correct, and they increment `swaps_` and `comparisons_` as necessary. Implement the sorting algorithms as they are defined in the header file. **Do not modify the provided functions. Do not add additional helper functions. Do not change the function prototypes of these classes**.

BubbleSort and InsertionSort

After finishing the above tasks successfully, add four files:

- `BubbleSort.hpp` — `BubbleSort.cpp`
- `InsertionSort.hpp` — `InsertionSort.cpp`

Similar to the other three sorting algorithms, `BubbleSort` and `InsertionSort` are to inherit from abstract class `SortingAlgo` class and `override` the `sort` function to implement the respective algorithms. Make sure to have the parameterized constructor. You are allowed to add any helper functions as needed, as long as you override the `sort` and add a parameterized constructor similar to the other three algorithms (should take only one parameter `const bool& ascending`).

Submissions

You will submit your code to Gradescope through GitHub Classroom. The autograder will grade:

- `SortingAlgo.hpp` — `SortingAlgo.cpp`
- `SelectionSort.hpp` — `SelectionSort.cpp`
- `BubbleSort.hpp` — `BubbleSort.cpp`
- `InsertionSort.cpp` — `InsertionSort.hpp`
- `MergeSort.hpp` — `MergeSort.cpp`
- `QuickSort.hpp` — `QuickSort.cpp`

To prevent the abuse of autograder and to encourage testing, **the autograder output will be vague**. Your code needs to compile on Gradescope in order to get any feedback from the autograder. If it doesn't compile, make sure you fulfill all the requirements mentioned in the project specifications. The autograder will not run unless all the above files are provided and all of them contain the required functions. Once you pass the test cases on Gradescope, make sure to verify the documentation and design requirements before making the final submission.

Compiling and Testing

All the classes in this project are class templates. Therefore, you don't have to compile them separately. Compile using `g++ --std=c++17 main.cpp` . **I cannot stress enough that Gradescope is not a testing platform. Just because your code compiles doesn't mean it works properly. You need to write test cases and make sure your functions are doing what the instructions want**. Write test cases for all the functions you implement and all the edge cases. Testing is a very important skill that you need to learn by trial and error. If you get an error, look at the error message and find out which

line it is coming from. I recommend implementing each function and testing it before implementing the next one. When you have all functions implemented, the given main function should create output that matches the outputs in the given example output files. This is the exact same function that Gradescope will use to check if your code works correctly.

Tips and Hints

Here are some tips and hints for all the sorting algorithms. I will add more here if necessary, so keep an eye on this document. Each modification will be logged at the end of this file:

- For `SelectionSort`, you should swap only if the current index is not the smallest or largest.
- The Sorting slides showed how to implement each algorithm. The autograder assumes that you are using that implementation instead of plagiarizing off of some website or generative AI.
- I recommend you to create another `.cpp` file to test your code in addition to the `main.cpp`. If you create a new one, you can test one at a time without modifying the original.
- For `QuickSort`, no need to optimize by calling insertion sort like the slides. Simply `return` in the base case. I highly recommend using recursion for `QuickSort` and `MergeSort`.
- The `protected` data members and functions of base class `SortingAlgo` can be accessed by using `this ->`. You can see how the provided code in starter code does this.
- The given `printAlgorithms` function in `main.cpp` is exactly what the autograder in Gradescope will use to test your functions. If your code doesn't work with this function, you can assume that it won't work on Gradescope. Furthermore, you can use this function to find which one of your algorithms is not working. Match your output for the given lists in `main` function to the expected output files.
- The `comparator` only checks greater than and less than, depending on whether `ascending_` is true. It doesn't handle when two things are equal. While implementing the algorithms, make sure you are handling the equal comparisons appropriately. Count `<=` and `>=` as one comparison, similar to the `merge` function in starter code.
- You can use the `comparator` to implement the `isSorted` function in `SortingAlgo` class. This will ensure that the function works for both ascending and descending order.
- Make sure you are resetting the `swaps_` and `comparisons_` to 0 every time sort is called. This allows same object to be used to sort multiple lists without combining their swap and comparison counts.

Debugging and Gradescope

Debugging is a skill you need to pick up at this point of your Computer Science journey. Before asking me to debug your code, make sure you do the following:

- If Gradescope is not compiling, check if the provided `main.cpp` file works and it produced the expected output. If not, you can figure out which part is not working.
- If Gradescope is not compiling, check if your file names are exactly as stated in the submission instructions. Furthermore, check if you are using the same compiling command as stated in the instructions. If this fails, there is something wrong with your includes.
- Make sure all your functions are implemented. If not, it won't work on Gradescope.
- If you ask me to debug, give me the specific function, as opposed to "my code isn't working". The given `main.cpp` should be good enough for you to catch the function that might be causing errors.

Changelog

While any major changes are unlikely, changes made to this document will be listed here