CS 7345 Sorting Library- Multi Threaded API Documentation:

a. **Overview:** This library allows for the creation of a user-specified vector and the running of several sorting algorithms on any vector of numbers passed into it. The vector can be specified to be in reverse order (from how it was created), completely random-shuffled (if the specified array was not already in that order), and duplicated (80% duplicated, which tests edge cases of many sorting algorithms and is thus also typically used as a benchmark).
   i. **User Input:** The array can be of any length (at least, any before the hardware/browser completely runs out of memory to supply) and **must contain only integers.**
   ii. **Sorting Algorithms:** Bubble, Merge, Insertion

b. **Endpoints:** All functions within the JS port of the library are called through a defined "Algo" object. This allows for single-file instantiation and readability of commands. The commands that can be called are divided into two families- *Algo.Vector*, which encompasses all functions that create and manipulate the target vector to be sorted- and *Algo.Sort*, which encompasses the previously mentioned sorting algorithms -Bubble, Merge, and Insertion. Each Sorting algorithm in *Algo.Sort* also contains a multi-threaded version, accessed under the namespace "Multi": for instance, a multi-threaded bubble sort can be called using *Algo.Sort.Multi.Bubble.*
   i. ***Algo.Vector:***
      1. *Algo.Vector.Init(int):* creates an empty VectorInt, the emscripten-bound equivalent of a C++ std::vector<int>, which can then be populated using the C++ push_back(data) method or other C++-centric means (with some changes implemented in Embind, such as VectorInt.get(i) instead of indexing std::vector<int>[i] and VectorInt.resize() instead of std::<vector>int.reserve()) and used within the library's other functions.
         a. The argument "size" passed into the method is optional. If none is supplied, an empty VectorInt is returned. If a size is supplied, a VectorInt populated with a default value of ascending numbers (ie: {1,2,3,… n}) is returned.
         b. **In order to use the other functions in this library, a VectorInt MUST first be created using this function, or the resulting object will not properly be bound/typed.**
            i. *Example Call:*
               1. `var arr = Algo.Vector.Init();`
               2. `var arr = Algo.Vector.Init(3);`
            ii. *Output:*
               1. Returns an empty VectorInt object (in this case, "arr" now contains an empty VectorInt)
               2. Returns a VectorInt object of size 3 populated with values {1,2,3}.
      2. *Algo.Vector.Shuffle(VectorInt):* shuffles the inputted vector with the Fisher-Yates Shuffle.

            a. *Example Call:*
                i. `var random = Algo.Vector.Shuffle(arr);`
            b. *Output:*
                i. If we assume that “arr” contains a VectorInt holding {1,2,3,4,5}, one possible output of the function is a VectorInt holding {1,4,3,2,5}.
        3. *Algo.Vector.Reverse(VectorInt):* reverses the order of every element in the inputted vector.
            a. *Example Call:*
                i. `var random = Algo.Vector.Reverse(arr);`
            b. *Output:*
                i. If we assume that “arr” contains a VectorInt holding {1,2,3,4,5}, this function will output a VectorInt holding {5,4,3,2,1}.
        4. *Algo.Vector.Dupe(VectorInt):* duplicates 20% of the contents of the vector across the entire vector, such that each item is repeated roughly vector.size()/5 times.
            a. *Example Call:*
                i. `var random = Algo.Vector.Dupe(arr);`
            b. *Output:*
                i. If we assume that “arr” contains a VectorInt holding {1,2,3,4,5,6,7,8,9,10}, this function will output a VectorInt holding {1,2,2,2,2,2,6,6,7,7} (or equivalent).
    ii. ***Algo.Sort:***
        1. *Algo.Sort.Bubble(VectorInt):* Runs the Bubble Sort algorithm on the inputted vector and returns the sorted result.
            a. *Example Call:*
                i. `var bubble = Algo.Sort.Bubble(arr);`
            b. *Output:*
                i. If we assume that “arr” contains a VectorInt holding {2,1,3,5,4}, this function will output a VectorInt holding {1,2,3,4,5}.
        2. *Algo.Sort.Merge(VectorInt):* Runs the Merge Sort algorithm on the inputted vector and returns the sorted result.
            a. *Example Call:*
                i. `var merge = Algo.Sort.Merge(arr);`
            b. *Output:*
                i. If we assume that “arr” contains a VectorInt holding {2,1,3,5,4}, this function will output a VectorInt holding {1,2,3,4,5}.
        3. *Algo.Sort.Insertion(VectorInt):* Runs the Insertion Sort algorithm on the inputted vector and returns the sorted result.
            a. *Example Call:*
                i. `var insertion = Algo.Sort.Insertion(arr);`

        b. *Output:*
            i. If we assume that “arr” contains a VectorInt holding {2,1,3,5,4}, this function will output a VectorInt holding {1,2,3,4,5}.
    4. *Algo.Sort.Multi.Bubble(VectorInt):* Runs the Bubble Sort algorithm using three threads on the inputted vector and returns the sorted result.
        a. *Example Call:*
            i. `var bubble = Algo.Sort.Multi.Bubble(arr);`
        b. *Output:*
            i. If we assume that “arr” contains a VectorInt holding {2,1,3,5,4}, this function will output a VectorInt holding {1,2,3,4,5}.
    5. *Algo.Sort.Multi.Merge(VectorInt):* Runs the Merge Sort algorithm using three threads on the inputted vector and returns the sorted result.
        a. *Example Call:*
            i. `var merge = Algo.Sort.Multi.Merge(arr);`
        b. *Output:*
            i. If we assume that “arr” contains a VectorInt holding {2,1,3,5,4}, this function will output a VectorInt holding {1,2,3,4,5}.
    6. *Algo.Sort.Multi.Insertion(VectorInt):* Runs the Insertion Sort algorithm using three threads on the inputted vector and returns the sorted result.
        a. *Example Call:*
            i. `var insertion = Algo.Sort.Multi.Insertion(arr);`
        b. *Output:*
            i. If we assume that “arr” contains a VectorInt holding {2,1,3,5,4}, this function will output a VectorInt holding {1,2,3,4,5}.

c. **Design Patterns:**
    i. **Strategy Design Pattern:** This library utilizes the Strategy Design pattern, allowing the main program, through the use of the “Algo” object, to seamlessly select the algorithm that is to be run at runtime (as opposed to hard-coding it). In C++, this “algorithm switching” (depending on desired input/output) is done through the use of function pointers that the Algo object keeps track of; while in the JS library, this same effect is achieved by allowing each function to be callable members fields of the Algo object.
        1. **Pros:**
            a. This strategy pattern allows the library to be easily expanded and extended (another emscripten-bound c++ sorting method could very easily extended to Javascript as a callable “Algo.Sort.myNewSortingFunction”), as all sorting algorithms

(strategies) take the same input (a single VectorInt) and yield the same output (a single, sorted VectorInt).

b. All algorithms are bound to the Algo object, so the only instantiations required once the library is loaded are the Algo object (which is automatically done in 'library.js') and the VectorInt object that will be sorted (which is also called through the Algo object).

2. Cons:

a. Like all Strategy patterns, the user must be aware of the entire definition of the Algo object to use it (to use a Strategy, the user must know it exists).

d. Multi-Threading:

i. Use of Threads:

1. In order to speed to execution time within slower algorithms when the data set becomes large (for instance, a Bubble Sort, running at $O(n^2)$, is quite slow when the data input size becomes large). Because an array can be broken into parts and later merged into one (similarly to the Merge Sort algorithm), this approach was used to sort more numbers at once.

a. As an example, if the input vector held 999 numbers, this would take approximately (999^2) time units, while with a multi-threaded approach, it would take $3*(333^2)$ time units to sort. This is a speedup by a factor of 3x, without inclusion of the thread creation overhead and eventual re-merge of the vector pieces, and it is this speedup that the library attempts to advantage of.

ii. Mutexes and Thread-safe Protocol:

1. Because this library operates with multiple threads by independently performing operations on completely separate and distinct vectors- and the calling thread will wait until all three threads have returned their sorted portion of the full vector before it is all stitched back together- there is no point where one thread's vector will interfere with another. Thus, because there are no "thread-unsafe" data pieces, there are no reasons for control structures such as mutexes.

iii. std::async vs std::thread:

1. Std::thread allows for explicit thread creation by wrapping objects- and each time it is called, a *new thread* is created. These thread objects are extremely expensive to create and destroy, creating significant overhead if they are needed to be created and destroyed multiple times.

2. Std::async, on the other hand, does not map the creation of a brand-new thread, every single time a task is created. The C++ std implementation of async automatically manages a small pool of threads and re-uses them for multiple tasks instead of deleting and re-creating them. This means that **std::async thread creation is significantly better for a large series of small, independent tasks-** and sorting multiple vectors by running multiple algorithms over multiple types and lengths of inputs can be classified as a series of small, independent (as each sorting is completely independent from another, and the runtime of each algorithm is deterministic, short (based on input size- the largest Big O in the library is Bubble Sort's O(n^2)), and, thus, predictable).

   a. However, because std::async does not delete threads until the end of the program, they cannot be mapped to an emscripten pthread-based (USE_PTHREADS) interface. **For this reason, std::thread was chosen over std::async**, despite the latter being a stronger fit for the type of problems a sorting algorithm library is likely to face**.**

iv. Why Three Threads?

1. To prevent threads from sharding each other and to better take advantage of multi-threading by not requiring constant re-scheduling of threads, it is recommended to run (number_of_computer_cores – 1) threads at any one time. The machine this library was tested on has 4 cores, and, thus, would run three threads.

v. *Pros of Multithreaded Approach:*

1. A constant number of threads using a constant method (either by std::async with an std::launch::async launch parameter or explicitly calling std::thread only during the method, itself, when threads are required, means that the user does not need to create extra threads/more than the tested machine can handle), will always spawn threads the same way- which leads to more predictable performance times. This is useful in JS and in the browser, where the many different activities and processes typically make performance significantly more unpredictable than a specialized C++ application.

2. A lack of thread-dangerous practices (and thus a lack of mandatory mutex structures) means no overhead is associated with creating and managing threads besides the initial creation and eventual re-joining of them.

vi. *Cons of Multithreaded Approach:*

1. This threaded approach is coded to, for each sorting algorithm, always use three threads- no more, and no less (because the testing machine had four cores, and more threads would increase the chance of sharding and skew timing results). As this number is hard-coded, the library would need to be updated to use more or less than this number of threads for a given task. As computers become more advanced, such a design decision will need to be adapted to better utilize a higher amount of cores.

2. The **SharedArrayBuffer,** which allows threads to communicate in the browser through Emscripten's Pthread-based interface, has been locked behind certain HTTP headers as a security countermeasure- which means that, while using the library in the browser, it must pass its own headers to these browsers to enable the buffer or simply be tested in its own web server where these headers can be manually created and edited.

3. The future/promise-based *std::async/std::future* utilizes a ".get()" function to return what the output of a thread. If called directly, this may will block the calling thread (even the main browser thread, which is very dangerous) to block idly until all spawned threads are obtained. Thus, runtime is restricted to calling these functions within a JavaScript WebWorker. ***However,*** **because WebWorkers are unable to parse entire custom classes as arguments, this library is thus better suited to being run through an interface such as Node than within a browser, as the blocking calls required to wait for threads to finish would inevitably block the main thread.**

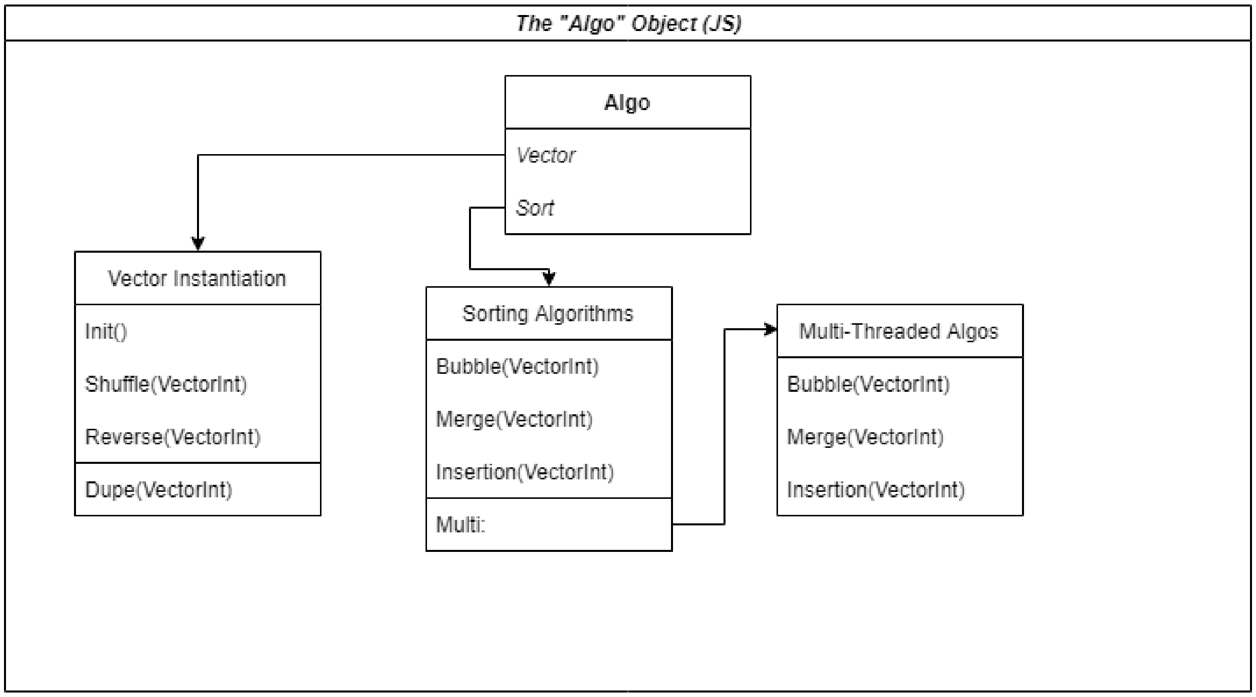
e. **Other Design Decisions:**

   i. **Use of Vectors:** Unlike arrays, which must be manually resized to change their size, vectors are able to be quickly imported to any language and are commonly used. Because the runtimes of all a vector's factory operations are universally understood, it can very easily serve as a base for numeric operations. As vectors are also a templated type, the library, itself, can also be very easily templated (even for completely-user-defined classes, provided operations such as the equality operator are also defined for those classes).

   ii. **Default Value in GetVector(size):**

      1. If size is supplied, a default ascending numeric scheme is supplied as a test case for each sorting method- testing how an algorithm will act if *the given vector doesn't need to be sorted.*

f. **Class Diagram/UML (JS):**



g. **Dependencies:**

i. *sorting_functions.cpp:* The file containing the C++ functions that will be bound by Embind to callable attributes that can be accessed using Javascript.

ii. *sorting_functions.js:* This file contains the generating bindings from running Embind on the "sorting_functions.cpp" file. It must be included for any calls to the Algo object. **This file must also be loaded before "library.js" is loaded, as it relies on the definitions found in this file.**

iii. *library.js:* This file contains the definition for the Algo object and acts as a wrapper from the sorting function library contained in sorting_functions.js to. These files were split (as opposed to simply adding the definition of the Algo object to sorting_functions.js) so that compilation using emcc's binding commands do not erase the Algo object's definition.

1. This also has a *"library_with_multi.js" variation* for the multi-threaded definitions. Either file can be used to run a single-threaded WASM application.

iv. **Note:** Because some browsers asynchronously load data**, all uses of the library must be made AFTER the module completely loads. This can be achieved by wrapping calls to the library in the Module.onRuntimeInitialized function, as seen below.**

1. For the multi-threaded library, specifically, it is not able to be run in this way without blocking the calling (or main) thread, which is not advisable in a browser environment.