**Brandon Hobbs**

**CS-300**

**Jan. 12, 2022**

**Module 2 VectorSorting Reflections**

The code is broken down into the following functions/structs:

* strToDouble
  + Used to convert the CSV file data into useable value
* Bid
  + Struct containing the data
  + Used with the vector that will be sorted
* displayBid
  + Used to send the values contained in the vector to the console
* getBid
  + Unused
* loadBids
  + Function used to read in the csv data
  + Can read the csv path in from arguments or use a default path
  + Add parts of the data into the Bid structure and then adds that Bid to the unsorted vector
* partition
  + Function that calculates the partition for the quick sort function
  + *partition* tries to find the elements in half of the of the vector that are out of order and then swap them
* quickSort
  + *quicksort* is a recursive function that drives the sorting via the partition function
* selectionSort
  + *selectionSort* is a function that uses the selection sort algorithm
  + there are 2 loops that scan for vector elements out of order and then swaps the elements
* main
  + *main* is the primary driver for the application
  + *main* has a menu to load the data, view the data, sort via *quickSort* or *selectionSort* algorithms and then exit the application
  + *main* also reports the timing each algorithm takes to perform the sort using the *time.h* library

The code was straight forward, especially since the parser was delivered to me. The only issue I faced was with Visual Studio throwing errors on *cin* and *cout* saying they were ambiguous. I have no idea why as I had the std namespace designated but I solved them by defining it as *std::cout* and the errors went away.

Pseudocode:

**Main** Function()

**Read** cmd arguments

**Store** argument as CSV file path

**If** no cmd arguments load default CSV file path

**Loop** while choice is not equal to ‘9’

**Output** menu

**Get** user input; Store in choice

**Validate** user input

**If** choice is not 1-4 or 9 throw an error

**If** choice equals ‘1’

**Start** the clock and **store** in ticks

**Call** loadBids and store CSV data in struct bids

**Output** number of records in the CSV file

**Stop** the clock

**Output** the elapsed time needed to read in the CSV file

**If** choice equals ‘2’

**Loop** through all the records in vector bids

**Call** displayBids()

**If** choice equals ‘3’

**Start** the clock and store in ticks

**Call** selectionSort passing bids

**Stop** the clock

**Output** the elapsed time needed to sort the vector

**If** choice equals ‘4’

**Start** the clock and store in ticks

**Call** quicksort() passing bids

**Stop** the clock

**Output** the elapsed time needed to sort the vector

**If** Choice equals ‘9’

**Exit** the application

**Output** ‘Good bye’

**End**

**selectionSort()**

**Get** vector to sort

**Loop** from smallest index to penultimate index of vector

**Set** *indexSmallest* to the current index position

**Loop** from the current index of outer loop through vector

**Compare** vector element at loop iterator to vector element at *indexSmallest*

**If** vector element at loop iterator is less than vector element at *indexSmallest* make *indexSmallest* equal to loop iterator

**Swap** the vector element at *indexSmallest* with the vector element at the outer loop’s position

**End**

**quicksort()**

**Get** vector to sort, lowest index of vector and highest index of vector

**If** lowest index if greater than or equal to highest index return nothing

**Call** partition() function

**Set** *lowEndIndex* equal to the value returned by the partition function

Recursively **call** quicksort passing the vector, lowest index, and *lowEndIndex* (from above)

Recursively **call** quicksort passing the vector, *lowEndIndex* (from above) plus one, and highest index

**End**

partition()

**Get** the vector to partition, the lowest index and the highest index

Determine the vector element at the midpoint between the lowest and highest index

**Set** pivot equal to this vector element

**Loop** until the lowest index is greater than or equal to the highest index

**Loop** through the vector from lowest index until a vector element larger than the pivot is found

**Overwrite** lowest index with this element’s position

**Loop** through the vector from lowest index until a vector element smaller than the pivot is found

**Overwrite** highest index with this element’s position

**Swap** the vector elements at the new highest and lowest index

**Overwrite** the lowest index by incrementing it one

**Overwrite** the highest index by decrementing it one

**Return** the highest index

**End**