**Vector Sorting: Code Reflection and Pseudocode**

Justin Starr

Department of STEM

CS 300 – DSA: Analysis and Design

Professor Ricardo Scarello

May 19, 2024

**Vector Sorting: Code Reflection and Pseudocode**

**Code Reflection**

The purpose of the code is to load information from a municipal government data feed that contains bids submitted for property auctions, which is unsorted. The code enables the user to load the unsorted data and then choose from either the selection sort or quick sort algorithms to sort the data based on the size of each bid’s title. The user can also choose to display the list of bids, which helps the user see whether or not the bids have been appropriately sorted. If the user wants to un-sort the list, the user can simply re-load the CSV file. Another purpose of the code is to show the user what the computational time is for each of the algorithms. After running the algorithms, the user can easily see that the quick sort algorithm is much faster than the selection sort algorithm. This is because while the selection sort algorithm is less complex than the quick sort algorithm, the number of comparisons the selection sort must make to sort a list, or in this case a vector, is much larger requiring longer execution/computational time.

To solve problems I followed the comments in the provided code as well as made use of the textbook and the code that was provided for the algorithms. I faced one major challenge while implementing the algorithms and that was I had placed a return inside the while loop in the partition function instead of outside loop. Therefore, my vector was not being sorted properly because it would return back to the quickSort function too quickly. Once I realized my mistake, everything worked as intended.

**Pseudocode**

The following is the pseudocode for the main function and the three functions necessary to perform the sorting algorithms (int partition, void quickSort, and void selectionSort):

**int main(int argc, char\* argv[])** function

DEFINE csvPath (type string)

SWITCH argc

case 2:

STORE csvPath = argv[1]

break

END case 2

default:

STORE csvPath = “eBid\_Monthly\_Sales.csv”

END default case

END SWITCH

DEFINE bids (type vector<Bid>)

DEFINE ticks (type clock\_t)

DEFINE choice (type int) and INITIALIZE choice = 0

LOOP WHILE choice not equal to 9

DISPLAY “Menu:” to output and end line

DISPLAY “1. Load Bids” to output and end line

DISPLAY “2. Display All Bids” to output and end line

DISPLAY “3. Selection Sort All Bids” to output and end line

DISPLAY “4. Quick Sort All Bids” to output and end line

DISPLAY “9. Exit” to output and end line

DISPLAY “Enter choice: “ to output and end line

INPUT choice

SWITCH choice

case 1:

STORE ticks = CALL clock() // Current clock ticks

STORE bids = CALL loadBids(csvPath)

DISPLAY bids.size() “ bids read”

STORE ticks = CALL clock() – ticks // current clock ticks minus

starting clock ticks

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “ seconds”

break

END case 1

case 2:

LOOP FOR int i=0; i < bids.size(); ++i

CALL displayBid(bids[i]);

END FOR LOOP

break

END case 2

case 3:

STORE ticks = CALL clock() // Current clock ticks

CALL selectionSort(bids);

STORE ticks = CALL clock() – ticks // current clock ticks minus

starting clock ticks

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “ seconds”

break

END case 3

case 4:

STORE ticks = CALL clock() // Current clock ticks

CALL quickSort(bids, 0, bids.size() - 1);

STORE ticks = CALL clock() – ticks // current clock ticks minus

starting clock ticks

DISPLAY newline for readability

DISPLAY bids.size(), “ bids sorted”

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “ seconds”

break

END case 4

END SWITCH

END WHILE LOOP

DISPLAY “Good bye.”

RETURN 0

END main function

**int partition(vector<Bid>& bids, int begin, int end)** function

DEFINE low (type int) and INITIALIZE low = begin

DEFINE high (type int) and INITIALIZE high = end

DEFINE middlePoint (type int) and INITIALIZE middlePoint = low + (high – low) /2;

DEFINE pivot (type string) and INITIALIZE pivot = bids.at(middlePoint).title;

DEFINE done (type bool) and INITIALIZE done = false;

LOOP WHILE not done

LOOP WHILE bids.at(low).title < pivot

low += 1

END WHILE LOOP

LOOP WHILE pivot < bids.at(high).title

High -= 1

END WHILE LOOP

IF low >= high

done = true

END IF

ELSE

swap(bids.at(low), bids.at(high))

low += 1

high -= 1

END ELSE

END WHILE LOOP

RETURN high

END partition function

**void quickSort(vector<Bid>& bids, int begin, int end)** function

DEFINE mid (type int) and INITIALIZE mid = 0

IF (begin >= end)

Return

END IF

STORE mid = CALL partition(bids, begin, end)

CALL quickSort(bids, begin, mid)

CALL quickSort(bids, mid + 1, end)

END quickSort function

**void selectionSort(vector<Bid>& bids)** function

DEFINE swap\_count (type int) and INITIALIZE swap\_count = 0

DEFINE min (type int) and INITIALIZE min = 0

DEFINE vectorSize (type size\_t) and INITIALIZE vectorSize = bids.size()

LOOP FOR size\_t pos = 0; pos < vectorSize; ++pos

STORE min = pos

LOOP FOR size\_t nextPos = pos + 1; nextPos < vectorSize; ++nextPos

IF bids.at(nextPos).title < bids.at(min).title

Min = nextPos

END IF

END FOR LOOP

CALL swap(bids.at(pos), bids.at(min))

INCREMENT swap\_count

END FOR LOOP

DISPLAY newline // for readability

DISPLAY swap\_count, “ bids sorted”

END selectionSort function