**CS 300 Module Two Assignment**

**Purpose of the Code**

The purpose of this code is to read data about bids from a CSV file, sort this data based on the title of each bid, and measure how long it takes to sort the data. The program allows users to choose between two sorting methods: selection sort and quicksort. It also provides options to load the bids from a file and display all the bids.

**Techniques Implemented**

To solve the problem of sorting the bids, the code implements two different sorting algorithms:

1. **Selection Sort**: This algorithm works by repeatedly finding the smallest element from the unsorted part and putting it at the beginning. It’s simple but not very efficient for large datasets.
2. **Quick Sort**: This is a more efficient sorting algorithm that works by dividing the array into two smaller arrays and sorting them independently. It’s faster for large datasets but more complex to implement.

To measure how long it takes to load and sort the bids, the code uses the clock() function from the C++ standard library, which provides a way to measure elapsed time by counting clock ticks.

**Challenges Encountered**

One challenge was ensuring the CSV file was read correctly. CSV files can have different formats, and any error in reading the file could lead to incorrect data or program crashes. Another challenge was accurately measuring the time taken for sorting, as this requires precise timing tools.

**Approaches to Overcome Challenges**

To handle CSV file reading, the code uses a CSV parser library, which simplifies reading and processing the file. Detailed error handling was added to catch and display any issues during file reading. For accurate time measurement, the clock() function was used, which calculates the time taken by measuring the number of clock ticks during the operation. Additionally, the program was structured in a way that makes it easy to understand and modify, with clear functions for loading, displaying, and sorting bids. This modular approach helps in isolating and fixing problems when they occur.

Pseudocode for Selection Sort Logic over bid.title (FIXME 1a)

FUNCTION selectionSort(bids)

// check size of bids vector

SET size to bids.size()

// pos is the position within bids that divides sorted/unsorted

FOR pos FROM 0 TO size - 1

// define min as int (index of the current minimum bid)

SET min to pos

// loop over remaining elements to the right of position

FOR j FROM pos + 1 TO size

// if this element's title is less than minimum title

IF bids[j].title < bids[min].title

// this element becomes the minimum

SET min to j

END IF

END FOR

// swap the current minimum with smaller one found

SWAP bids[pos] WITH bids[min]

END FOR

END FUNCTION

Pseudocode for Quick Sort Logic over bid.title (FIXME 2a)

Partition Function

FUNCTION partition(bids, begin, end)

SET low to begin

SET high to end

// Calculate the middle element as middlePoint (int)

SET middlePoint to (begin + (end - begin) / 2)

// Set Pivot as middlePoint element title to compare (string)

SET pivot to bids[middlePoint].title

// while not done

SET done to false

WHILE NOT done

// keep incrementing low index while bids[low].title < Pivot

WHILE bids[low].title < pivot

INCREMENT low

END WHILE

// keep decrementing high index while Pivot < bids[high].title

WHILE pivot < bids[high].title

DECREMENT high

END WHILE

// If there are zero or one elements remaining, all bids are partitioned. Return high

IF low >= high

SET done to true

ELSE

// swap the low and high bids

SWAP bids[low] WITH bids[high]

// move low and high closer

INCREMENT low

DECREMENT high

END IF

END WHILE

RETURN high

END FUNCTION

Implement the quick sort logic over bid.title (FIXME: 2a)

FUNCTION quickSort(bids, begin, end)

// Base case: If there are 1 or zero bids to sort, partition is already sorted

IF begin >= end

RETURN

// Partition bids into low and high such that midpoint is location of last element in low

SET mid to partition(bids, begin, end)

// recursively sort low partition (begin to mid)

CALL quickSort(bids, begin, mid)

// recursively sort high partition (mid + 1 to end)

CALL quickSort(bids, mid + 1, end)

END FUNCTION