

## Southern New Hampshire University

Joshua Wozny

[joshua.wozny@snhu.edu](mailto:joshua.wozny@snhu.edu)

January 4, 2023

### CS-300 Week 2: Selection and Quick Sort Algorithms

---

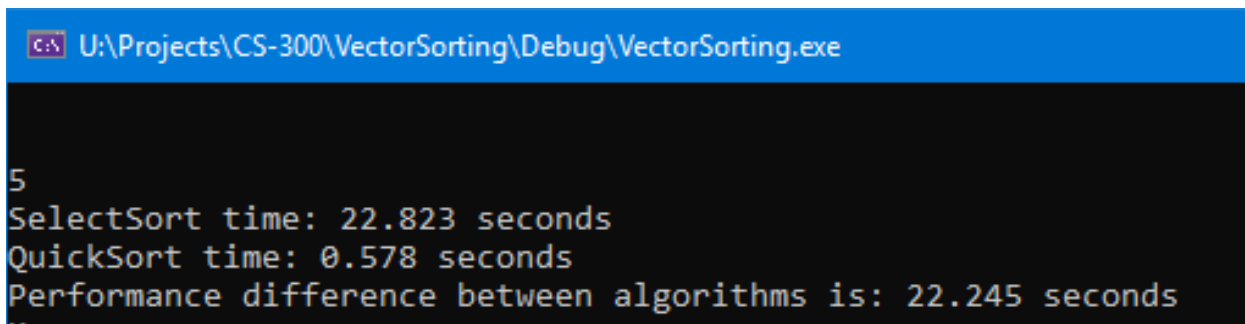
#### REFLECTION

This week we implemented two different sorting algorithms in C++ to develop a better understanding of the benefits and drawbacks of each. Both sorting algorithms, selection sort and quick sort, respectively, perform the same function. They each take a list (can be an array, or vector or other indexed data structure) and sort its elements in ascending order. Each algorithm uses a different approach which impacts its overall performance.

The selection sort algorithm is straightforward to implement. It iterates over each element in the list in order and finds the smallest value and puts it at the beginning of the list - and repeats until all elements have been sorted (see the pseudocode below). For small datasets, or when performance is not a relevant concern, selection sort performs adequately. However, with an average (and worst case) performance of  $O(n^2)$ , it is not an optimal choice for larger data sets.

The quick sort algorithm is more difficult to implement, requiring partitioning of the list into low and high sets, and calling the sort recursively until all elements have been sorted. The algorithm places all values below a pivot value (which can be any value, but our implementation used the middle value in the list) to the left of the pivot - leaving all values greater than the pivot to the right. It continues to partition until the partition size is 1 or 0, when all elements are sorted. Although more difficult to implement than the selection sort, its performance is much better with an average performance of  $O(n \log(n))$  - despite its worst case performance of  $O(n^2)$ .

The performance difference between the two sorting algorithms is significant for large datasets, but for small datasets the difference is not discernable. In my implementation with the large dataset in eBid\_Monthly\_Sales.csv the difference was 22.2 seconds.



```
U:\Projects\CS-300\VectorSorting\Debug\VectorSorting.exe
5
SelectSort time: 22.823 seconds
QuickSort time: 0.578 seconds
Performance difference between algorithms is: 22.245 seconds
```

## PSEUDOCODE

### Selection Sort

Average performance:  $O(n^2)$

Worst case performance  $O(n^2)$

SelectionSort (vector list, int start, int end):

- Definitions:
  - min as int (index of the current minimum bid), initialize as 0
  - size\_t as int equal to list.size()
  - pos as int is the position within bids that divides sorted/unordered, initialized to 0
- iterate over list from index=pos to index = size\_t -1
  - set min = pos
  - iterate over list from index = pos + 1 to index = t\_size -1 remaining elements to the right of position
    - if this element's title is less than minimum title
    - min = index;
  - if (min != pos) {
    - swap the current minimum with bid at index pos

### QuickSort

Average performance:  $O(n \log(n))$

Worst case performance  $O(n^2)$

QuickSort (vector list, int start, int end):

- set mid as int (index of the middle pivot point), initialize as 0
- Base case for recursion: if (start>=end) then list is already sorted
  - END OF SORT
- Partition list into low and high such that mid is location of last element in low
  - mid = partition(list, start, end)
    - set low and high equal to start and end
    - middle element as pivot point mid as int = (start + end)/2
    - set pivot as value of list at mid
    - keep incrementing low index while list at low < list at pivot

## CS-300 Week 2: Selection and Quick Sort Algorithms

- keep decrementing high index while list at pivot < list at high
- If there are zero or one elements remaining, all bids are partitioned.
- value of high is returned to QuickSort
- Recursively sort low partition: QuickSort(list, start, mid) (*Repeats until Base Case*)
- Recursively sort high partition: QuickSort(list, mid+1, end) (*Repeats until Base Case*)
- END OF SORT