Conducting Experiments on Search Algorithms

Darren Funes

A.

- Selection:
  - Selection sort will require the comparison to increment regardless of a swap leading to $\frac{1}{2}(n^2 - n)$ amount of comparisons.
  - This implies that the code for selection sort was applied the same for all 3 input types
  - 49995000 is the number produced by the program as well as found by O(n2)

Algorithm	Input Type	# of comparisons	Comments on time complexity and # of comparisons
Selection Sort	Ordered	49995000	The number of comparisons should remain constant. The time complexity is $O(N^2)$ as the lowest number is compared to the number at the front.
Selection Sort	Random	49995000	
Selection Sort	Reverse	49995000	

- Merge:
  - Splits array in half
  - Recursively split array further
  - Merge halves together again
  - $n \log n - 2\log n + 1$ is the equation that would derive the comparisons
  - The complexity is O(NlogN) in all cases
  - The comparisons are added when it keeps getting called recursively

Algorithm	Input Type	# of comparisons	Comments on time complexity and # of comparisons
Merge Sort	Ordered	69008	The time complexity is O(nlogn) in all cases however certain permutations of the given values can cause an increase in comparisons
Merge Sort	Random	61210	
Merge Sort	Reverse	64256	

- Quick Sort
    - Takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot
    - It is compared when the partition is called recursively and incremented then this is also O(NlogN) at best and average however its worst time is O(n^2)

Algorithm	Input Type	# of comparisons	Comments on time complexity and # of comparisons
Quick Sort (Last Pivot)	Ordered	49995000	Partitioning based on the last element would cause the worst time complexity O(n^2) with ordered input as it partitions based on the last value which is the greatest.
Quick Sort (Last Pivot)	Random	76333	Partition on the last pivot is less impactful on total time because the given is completely random so number of comparisons has a higher chance of being lower than worst complexity
Quick Sort (Last Pivot)	Reverse	24995000	Partition that is reversed has comparisons based on the first values which would

- Randomized Quick Sort
    - Takes random element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot
    - It is compared when the partition is called recursively and incremented then this is also O(NlogN) at best and average however its worst time is O(n^2)

Algorithm	Input Type	# of comparisons	Comments on time complexity and # of comparisons
Randomized QS	Ordered	74283	Partitions that are random allows less of a chance of reaching the worst time complexity

Randomized QS	Random	68614	
Randomized QS	Reverse	70161	

- Heap Sort
  - Similar to selection sort where we place find the maximum element and place it at the end and will recursively call until all elements are in the correct positions
  - Will increment the comparisons if comparing right child or left child to the largest element with a complexity of O(NlogN)

Algorithm	Input Type	# of comparisons	Comments on time complexity and # of comparisons
Heap Sort	Ordered	121077	The number of comparisons here is dependent on the input order. The best and worst case are each Θ(n log n) - There's no difference between the two, though they can differ by a constant factor. Since big-O notation ignores constants, though, this isn't reflected in the best-case and worst-case analysis.
Heap Sort	Random	112914	
Heap Sort	Reverse	153619	

B.

In Selection sort I created a temporary array (line 20 - 22 of selectionSort.cpp) to hold values for a generic swap for the minimum value and value at the front.

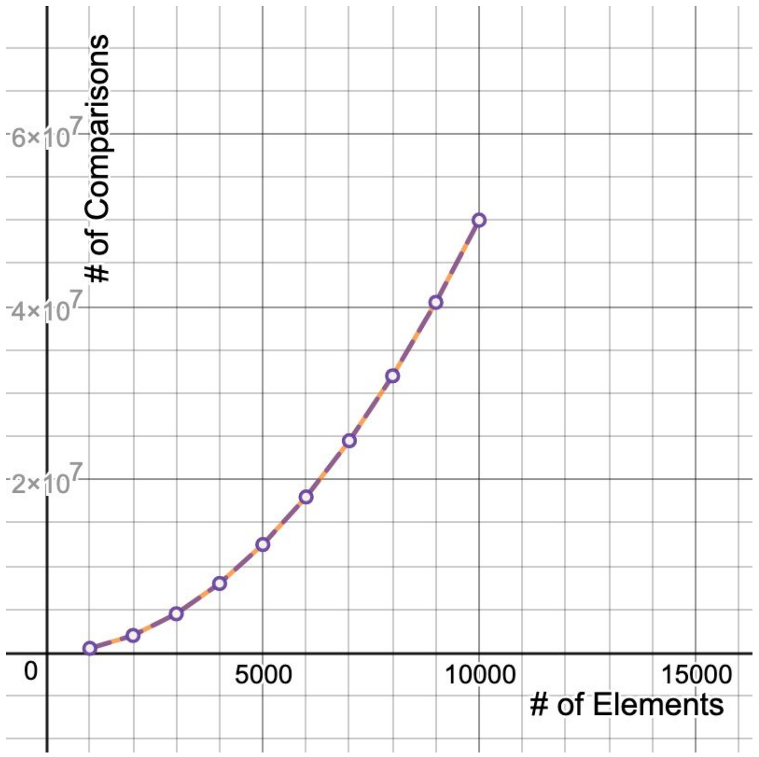
For merge sort it is necessary to use other data structures such as arrays to split recursively into left and right until not necessary. These are put to use in line 15 - 18 in mergeSort.cpp

Quick sort(line 16, 19 of quickSort.cpp), Heap sort (line 20, 36 of heapSort.cpp) and Randomized Quick Sort (line 17, 20, 29 of randomizedQuickSort.cpp) both requires use of the using namespace std and using swap. Swap contains a copy constructor and two assignment operations. As it is necessary to swap positions of the given input that pertain to the details of each algorithm.

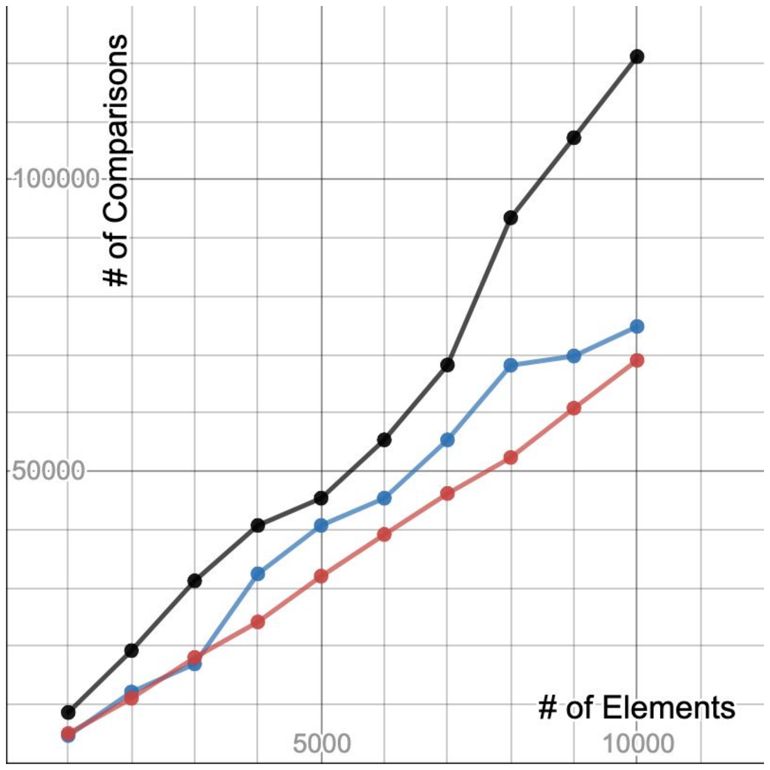
C.

Randomized Quick Sort is a good general purpose algorithm.

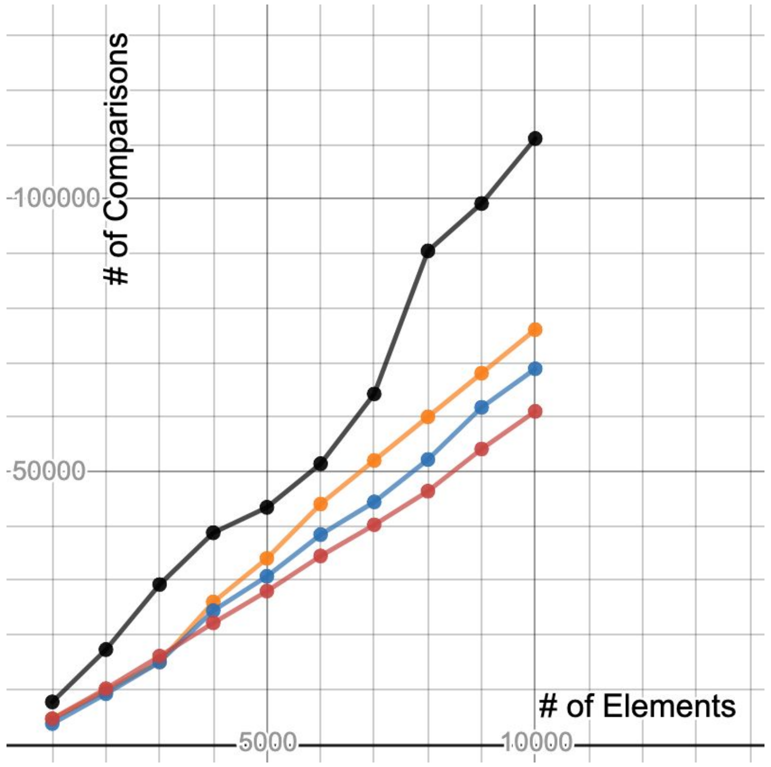
Ordered: Selection Sort(Purple) vs Quick Sort(Last Pivot) (Orange)



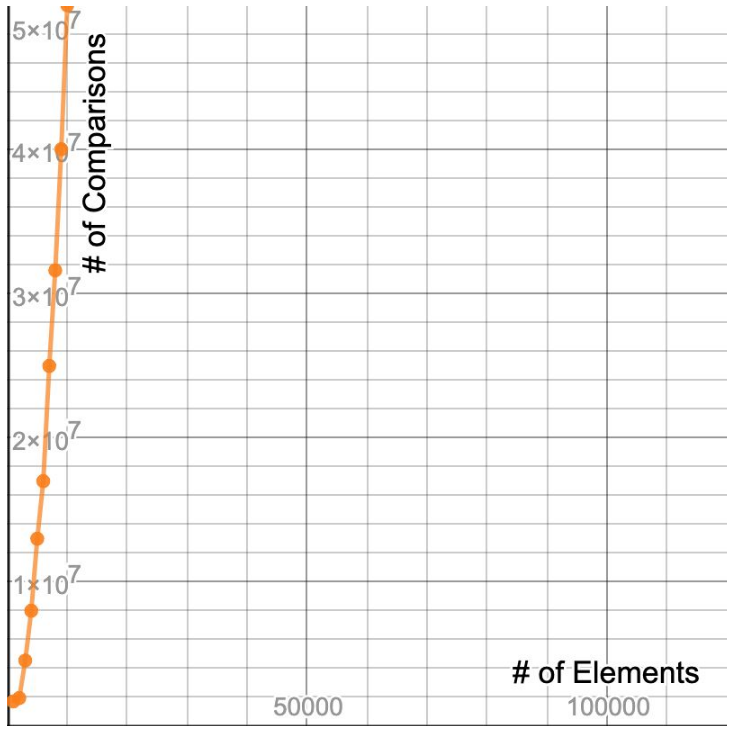
Ordered: Heap Sort(Black) vs Merge Sort(Red) vs Randomized QS (Blue)



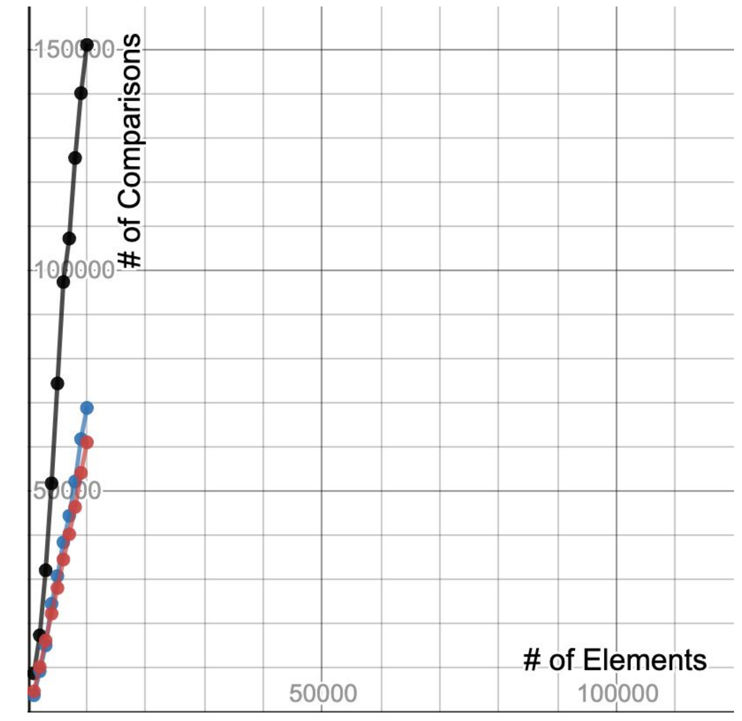
Random: Heap Sort(Black) vs Merge Sort(Red) vs Randomized QS (Blue) vs QS(Last Pivot)(Orange)



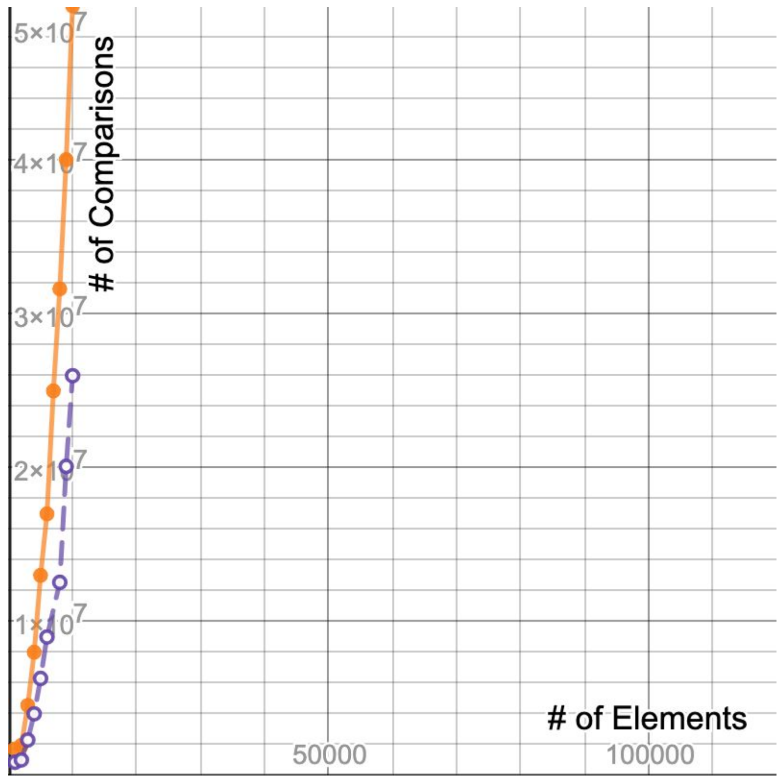
Random Selection Sort



Reverse: Heap Sort(Black) vs Merge Sort(Red) vs Randomized QS (Blue)



Reverse: Selection Sort(Purple) vs Quick Sort(Last Pivot) (Orange)


5×10^7
4×10^7
3×10^7
2×10^7
1×10^7
# of Comparisons
# of Elements