# Experimental Questions

1. Sorting Algorithms with various input sizes

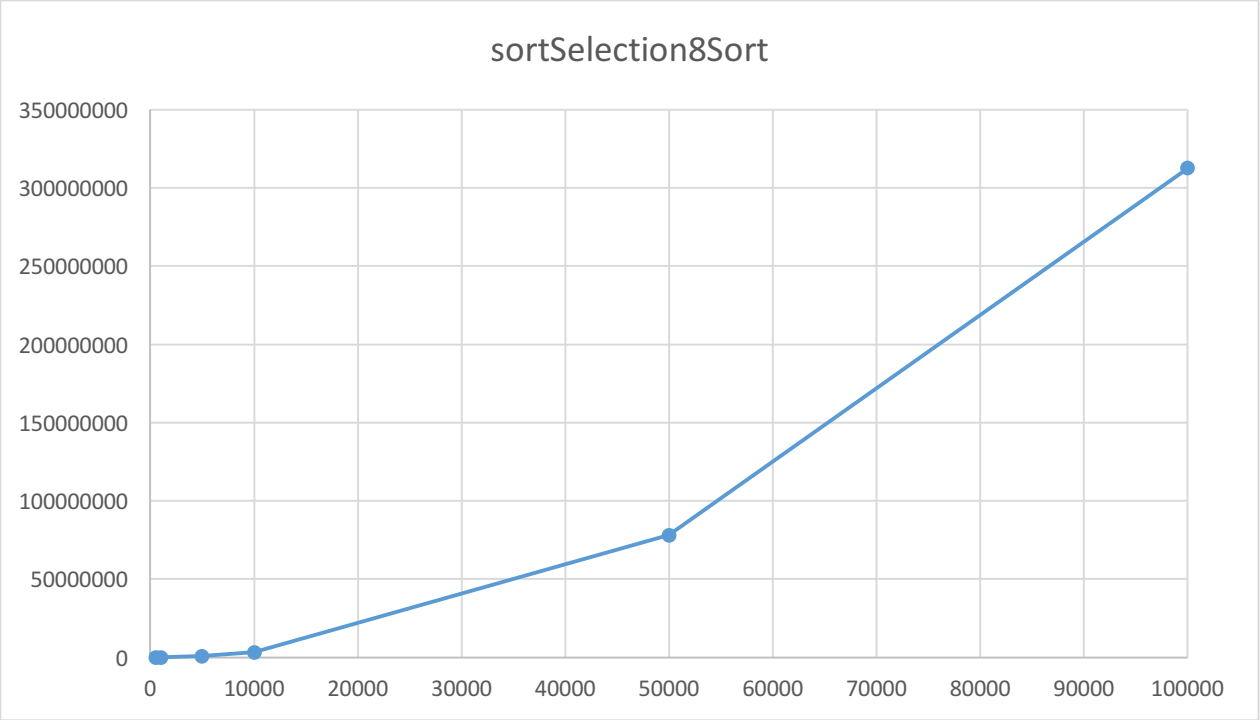
Average value of Compares

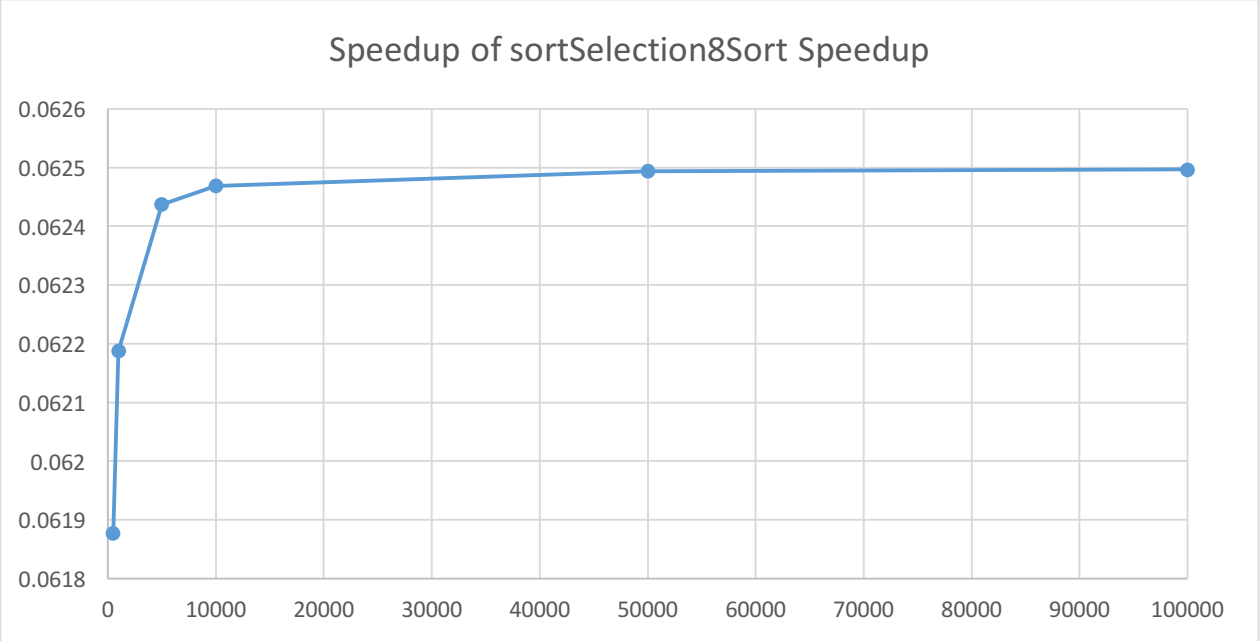
| Input Size | Standard Selection Sort | sortSelection8Sort | sortSelection8Min |
|---|---|---|---|
| 500 | 125249 | 7750 | 18036 |
| 1000 | 500499 | 31125 | 71786 |
| 5000 | 12502499 | 780625 | 1787500 |
| 10000 | 50004999 | 3123750 | 7146429 |
| 50000 | 1250024999 | 78118750 | 178589286 |
| 100000 | 5000049999 | 312487500 | 714321429 |

| Input Size | Standard Merge Sort | sortMerge8Sort |
|---|---|---|
| 500 | 3857.1 | 580.4 |
| 1000 | 8711.4 | 1336.8 |
| 5000 | 55214.5 | 9765.7 |
| 10000 | 120438.4 | 21246.6 |
| 50000 | 718171.8 | 120292.3 |
| 100000 | 1536272.5 | 257779.2 |

## Plot of the Speedup for each pair

### Standard Selection Sort vs sortSelection8Sort

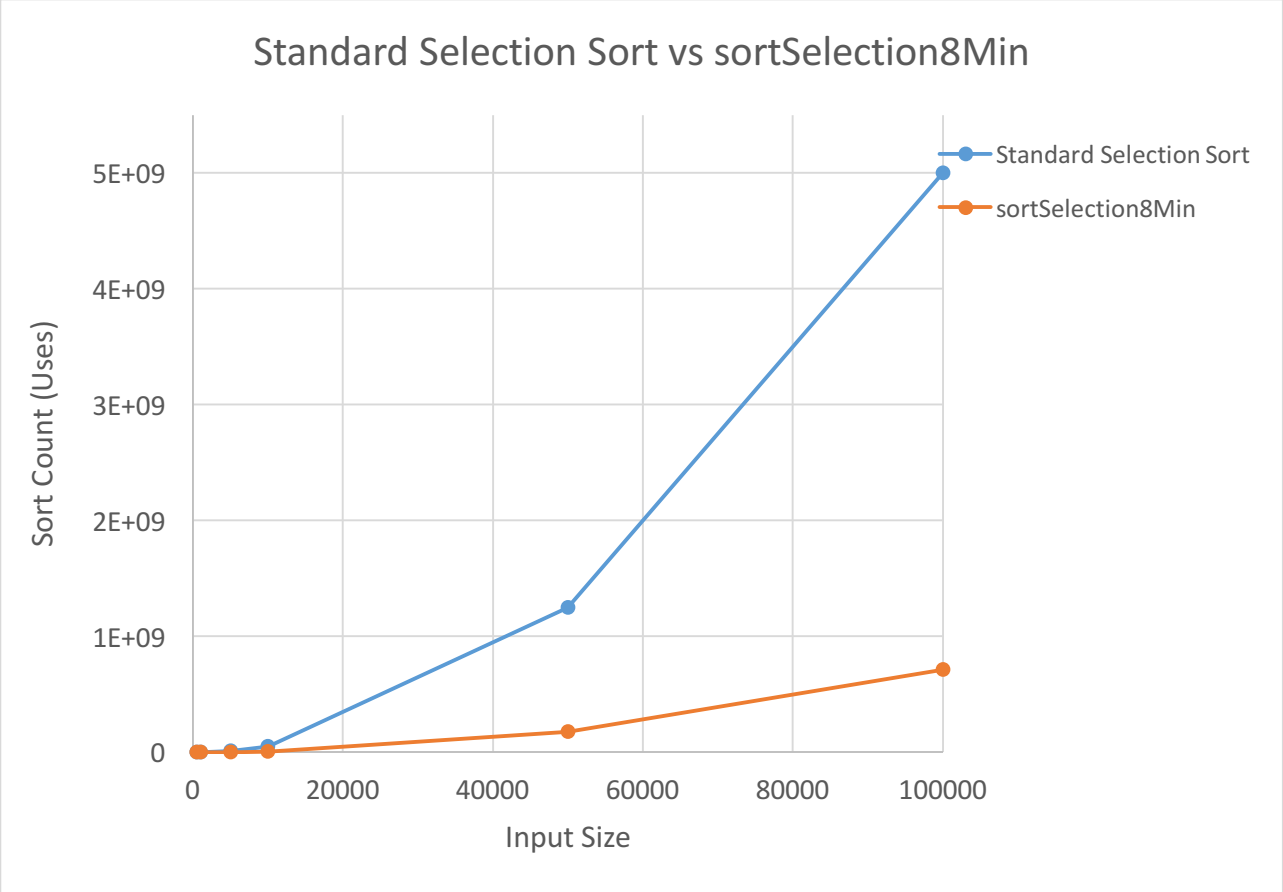


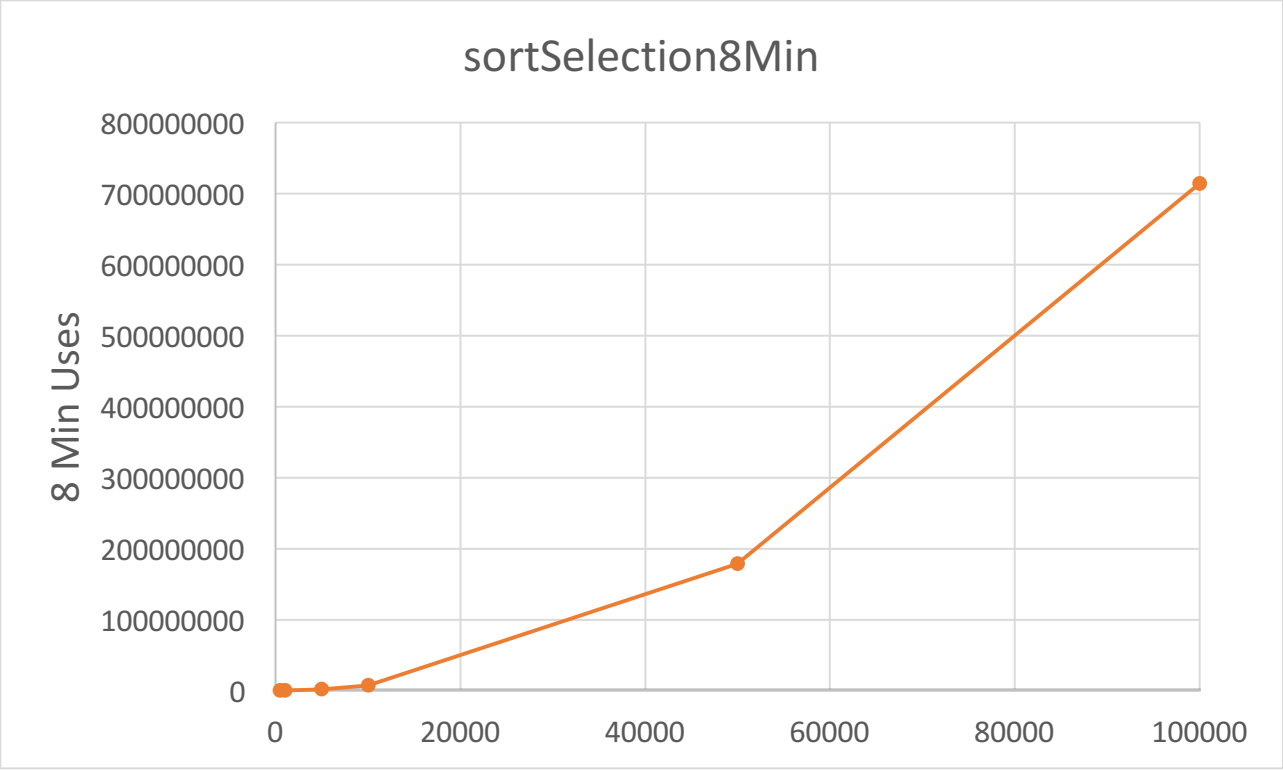



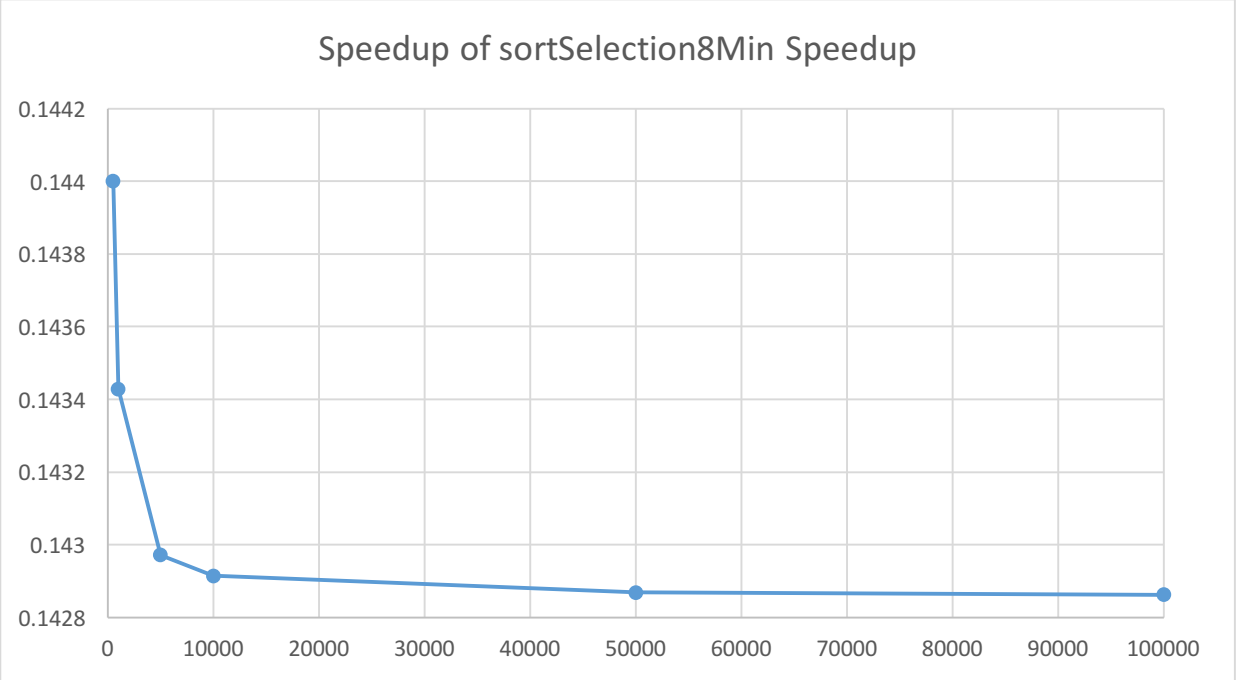

The rate of speedup for sortSelection8Sort is increasing with respect to n. As n grows larger, speedup stabilizes to 0.0625. sortSelection8Sort is using only about 6.19-6.25% of number of comparisons used for standard selection sort.

Number of comparisons is growing almost exponentially as n increases. Selection sort implementations have $O(n^2)$ time complexity and this is reflective in the plots above. Both standard selection sort and sortSelection8Sort are using exponential number of compares as n increases.

**Standard Selection Sort vs sortSelection8Min**

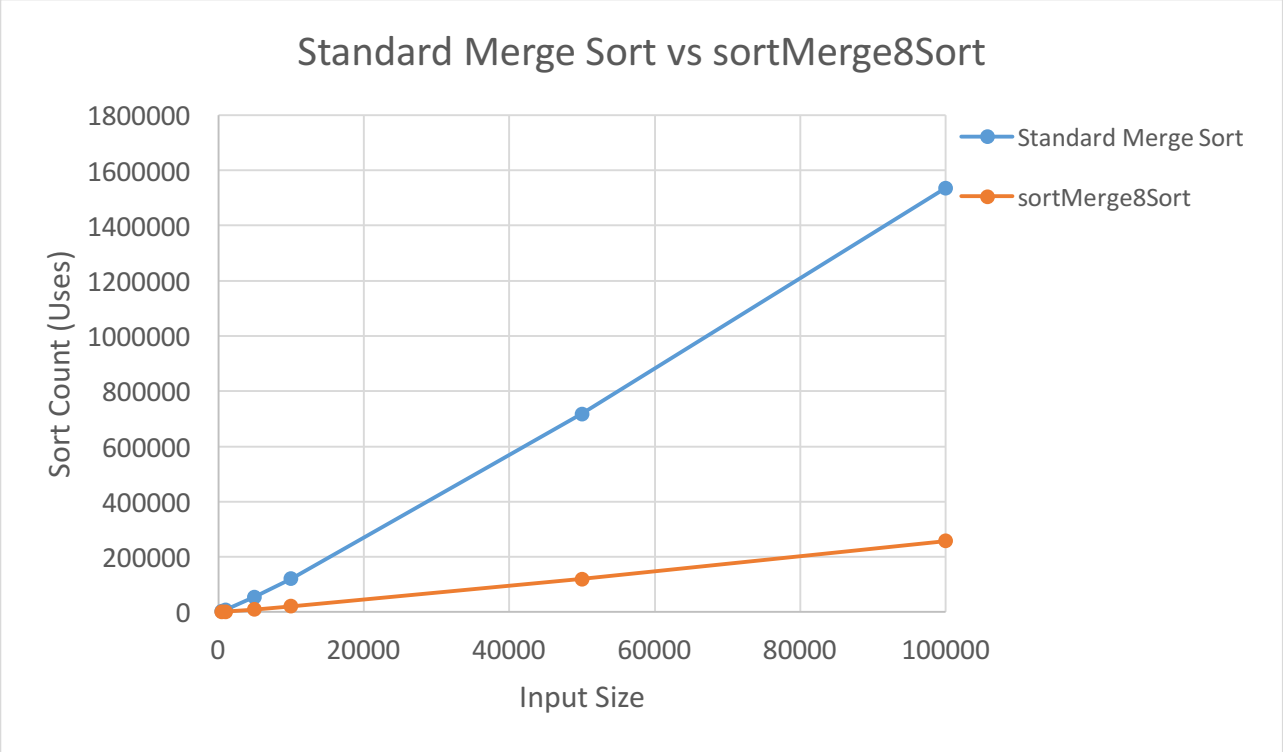


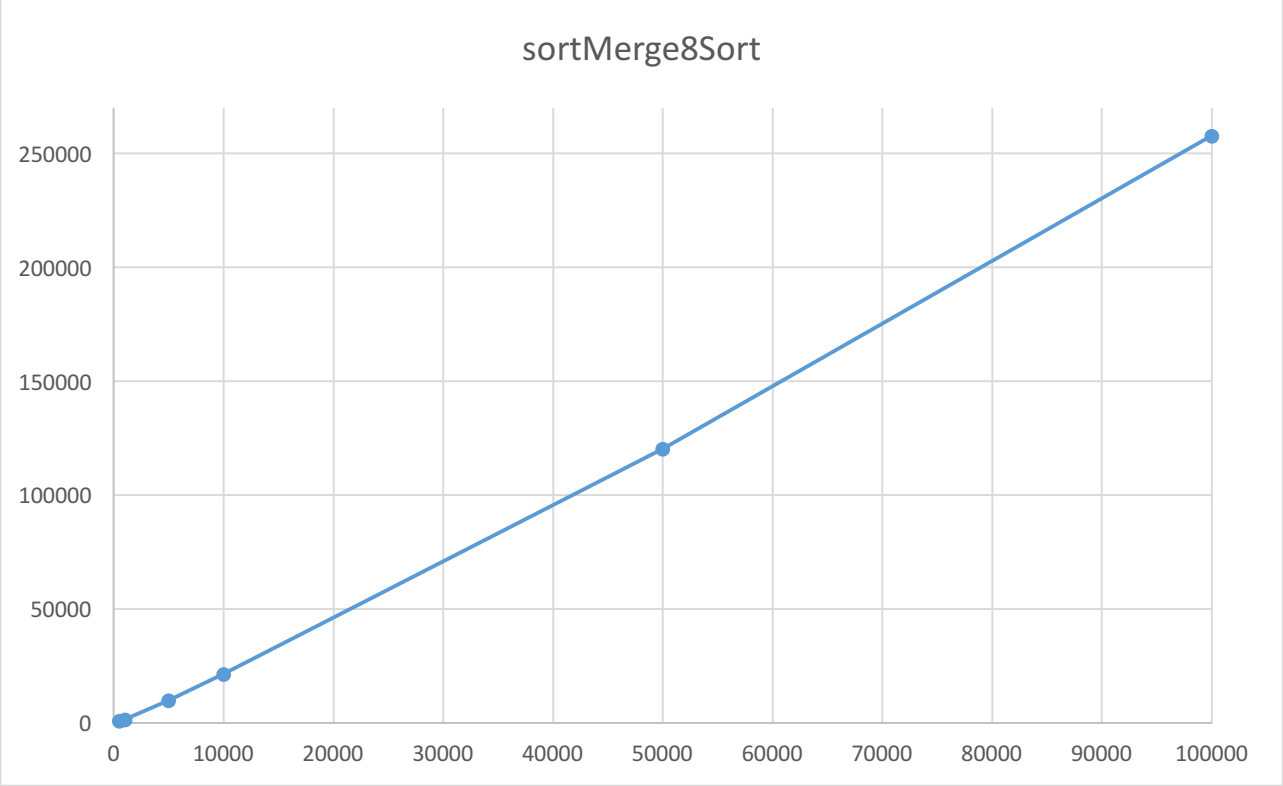



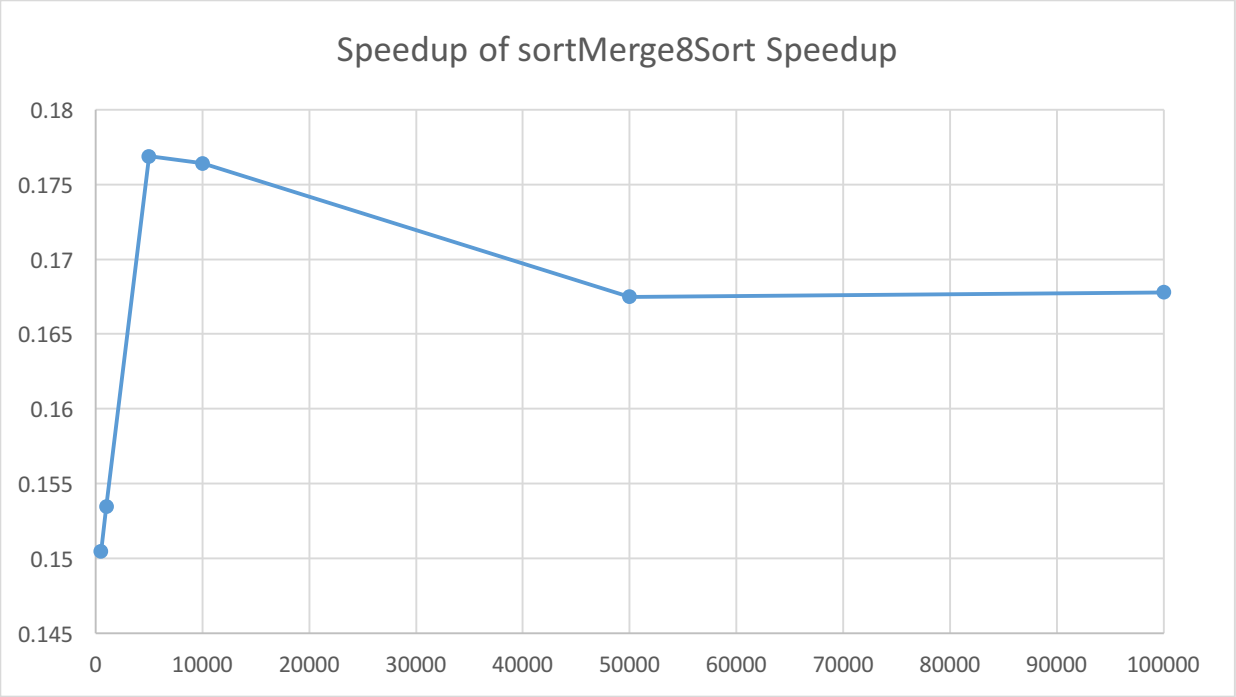



The rate of speedup for sortSelection8Min is decreasing with respect to n. As n grows, speedup stabilizes to 0.1429. sortSelection8Min is using about 14.4-14.29% of number of comparisons used for standard selection sort.
Number of comparisons is growing almost exponentially as n increases. Selection sort implementations have $O(n^2)$ time complexity and this is reflective in the plots above. Both standard selection sort and sortSelection8Min are using exponential number of compares as n increases.

**Standard Merge Sort vs sortMerge8Sort**






The rate of speedup for sortMerge8Sort increased when input size increased from 500 to 5000. When size of input was greater than 5000, speedup decreased with respect to n. As n grows, speedup stabilized to 0.167. sortMerge8Sort is using about 15.1-17.6% of number of comparisons used by standard merge sort.
Number of comparisons is growing almost linearly as n increases. Merge sort implementations have O(nlogn) time complexity and this is shown in the plots above.

2. Performance of all three Magic Box algorithms differs on the base of implementation. Selection sort implementations take more magicbox methods as the size of n increases. Meanwhile, merge sort implementation takes consistently much less time and fewer magicbox sort uses.

**Analytical Questions**

1. My implementation first gets the minimum of the first 8 elements of the array. It then copies the next 7 elements to an array (of size 8) and add the found minimum to the last index. If eightMin returns an index less than 7, the variable that contains the index of the minimum is updated. Since it compares first 8 elements and then 7 elements at a time, exact bound is $\left\lceil \frac{n-8}{7} \right\rceil + 1$ or Big-Oh of O(n). Here is a table of reported number of counts when n >= 8.

| Input size (n) | Number of Uses of eightMin |
|---|---|
| 8 | 1 |
| 64 | 9 |
| 96 | 14 |
| 100 | 15 |
| 500 | 72 |
| 1000 | 143 |
| 5000 | 715 |
| 10000 | 1429 |
| 100000 | 14286 |

2. isSorted8Min uses eightMin() n-1 times. My implementation iterates through array elements and run eightMin() at every index. If it returns 0, we know that the current index in the array is the minimum and that the array is sorted at that point. If eightMin() returns a value not equal to 0, we know that the array is not sorted and return false. If the array is sorted, my implementation will check until the second last element. At the last two elements, we just need to check if the current index is the smallest and we can deduce that the last element is the largest. Its exact bound is n-1 or Big-Oh of O(n).

My implementation of isSorted8Sort is almost identical with my min8Min() implementation described above. It first sends in first 8 elements to eightSort() and sees if the returning array returns indices from 0-7 order. If indices do not return 0-7 in numerical order, method returns false. If it's in order, method points to the $8^{th}$ element and sends it to eightSort() with 7 next elements. I don't move to $9^{th}$ element right away it could overlook the numerical order between the two. This implementation uses eightSort() $\left\lceil \frac{n-8}{7} \right\rceil + 1$times as described in the table below. It will have a Big-Oh of O(n).

| Input size (n) | Number of Uses of eightMin | Number of Uses of eightSort |
|---|---|---|
| 8 | 7 | 1 |
| 64 | 63 | 9 |
| 96 | 95 | 14 |
| 100 | 99 | 15 |
| 500 | 499 | 72 |
| 1000 | 999 | 143 |
| 5000 | 4999 | 715 |
| 10000 | 9999 | 1429 |
| 100000 | 99999 | 14286 |

3. In worst case, selection sorts take $O(n^2)$ time complexity. sortSelect8Sort and sortSelection8Min both traverse the whole array as they sort the elements in order. For sortSelection8Min, it finds the minimum from the array and starts to swap it with the beginning index. If the array is in reverse order, it will have to traverse the array to find the corresponding minimum to the index. As it's traversing the array twice in a nested for loop, worst case performance will be $O(n^2)$. sortSelection8Sort will be the same as it's a selection sort based implementation. Even if it makes about 1/16 (6.25%) number of comparisons compared to Standard (check speedup above), it still needs to travers the array and sort elements for each of the indices it traverses.

4. On average, sortMerge8Sort uses only about a quarter of the number of comparisons made by the Standard merge sort according to the ratio shown above and speedup plot mentioned in experimental section. This is about $\frac{nlogn}{4.244}$ in average with my implementation. Worst Case Asymptotic Performance of algorithms sortMerge8Sort is O(nlogn).

| Average for Merge Sort | | Average Ratio: | 4.244144092 |
|---|---|---|---|
| Input Size | Standard Merge Sort | sortMerge8Sort | Ratio |
| 500 | 3857.1 | 827 | 4.663966143 |
| 1000 | 8711.4 | 1904 | 4.575315126 |
| 5000 | 55214.5 | 13920 | 3.966558908 |
| 10000 | 120438.4 | 30340 | 3.969624258 |
| 50000 | 718171.8 | 173060 | 4.149842829 |
| 100000 | 1536272.5 | 371120 | 4.139557286 |

5. Assuming 8-sort box is stable, my algorithm for sortMerge8Sort will be stable. Base case for my recursive implementation is when array length is less than or equal to 8. In that case, it runs 8-sort. This already gives out a stable array of indices. When array length is greater than 8 (ex: 16), it first gathers first 4 elements from each group of 8 and uses 8-sort box to sort it. It stores sorted 4 elements to a result array and selects next 8 elements to compare from each group. Suppose two groups of 4 (one from first group and second from other) had a same element that should be part of first 4 sorted elements after merge sort. My algorithm will add first group to an array of 8 and the next 4 to the remaining spaces. It will then go through mergesort implementation. Since it's a group of 8, it will be processed by base case and be sorted by 8-sort box. This will return a stable output. When it sorts the array based on the result from 8-sort box, array will show stable results.

| 1st group of 4 | | | | 2nd group of 4 | | | |
|---|---|---|---|---|---|---|---|
| 0 (Index) | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |

result after merge sort algorithm

| Original Index | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 4 | 1 | 5 | 2 | 6 | 3 | 7 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |

My algorithm for sortSelection8Sort is stable based on the implementation. It grabs a group of 4 items and the next group of 4 that needs to be sorted. Assuming that the 8-sort box is stable, it will arrange the items in the array in stable order if there are items with same values. With the array returned from 8-sort box, my algorithm arranges the values of the array corresponding to the indices which organizes the items in stable order. So even if these items are revisited again for further sorts, they will be kept in stable order.

Original Array

| 0 (Index) | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 19 | -31 | 19 | 2 | 1 | -31 | 1 | 2 |

Resulting indices from 8-sort box
How the array elements are changed after sort

| 1 | 5 | 4 | 6 | 3 | 7 | 0 | 2 |
|---|---|---|---|---|---|---|---|
| -31 | -31 | 1 | 1 | 2 | 2 | 19 | 19 |

6. I would implement a similar approach to 8-sort magic box for both selection sort and merge sort. For selection sort, it will be identical with sortSelection8Sort. It would first sort first $\sqrt{n}$ elements in the array and grab first $\frac{\sqrt{n}}{2}$ elements and compare it to the next $\frac{\sqrt{n}}{2}$ items until it reaches the end of the array. That $\frac{\sqrt{n}}{2}$ elements would be the sorted and will be placed from 0 to $\frac{\sqrt{n}}{2} - 1$ index of the array (after swapping). Start from $\frac{\sqrt{n}}{2}$ and grab up to $\sqrt{n}$ elements and repeat the process until the whole array is sorted. In worst case, this box will be used $\sum_{n=1}^{2\sqrt{n}} n$ times. For example, when 8-sort box was used for sortSelection8Sort for an array in reverse order, resulting count equaled 136 which is $\sum_{n=1}^{2\sqrt{n}} n = \sum_{n=1}^{16} n$ .
   For merge sort, the base case will be $\sqrt{n}$. When it's $\sqrt{n}$, use $\sqrt{n}$-sort box to find the sorted indices and sort the elements accordingly. Continue this process until all the elements are sorted in the array. In worst case, $\sqrt{n}$-sort box will be used $\frac{n \log_2 n}{\sqrt{n}}$ times.