Course: CSCI 335
Name: Jianhua Deng
Emplid: 24259587

	Size: 1k	Size: 31k	Size: 1 Million	Worse Case Quick Select
Standard Sort (Avg of the 3 same-size Samples)	0 ms	6 ms	266 ms	4 ms
Half Selection Sort (Avg of the 3 same-size Samples)	4 ms	4139 ms	N/A	1554 ms
Merge Sort (Avg of the 3 same-size Samples)	0 ms	19 ms	728 ms	9 ms
In Place Merge Sort (Avg of the 3 same-size Samples)	0 ms	11 ms	438 ms	4 ms
Half Heap Sort (Avg of the 3 same-size Samples)	0 ms	5 ms	222 ms	2 ms
QuickSelect (Avg of the 3 same-size Samples)	0 ms	1 ms	43 ms	582 ms
MedianofMedian (Avg of the 3 same-size samples)	0 ms	3 ms	88ms	1 ms

Standard Sort:
The Standard Library of Sort function in C++ is Introsort, it has O(nlogn) time complexity and it is a hybrid algorithm, meaning that it utilizes Insertionsort, Heapsort, and Quicksort. This algorithm is designed to mainly avoid the worst case of heapsort and quicksort’s O(n^2) time complexity, basically switching from quicksort to heapsort when the recursion depth goes beyond a certain threshold.

HalfSelection Sort

Halfselection sort took the longest out of all the algorithms because it is an O(n^2) time complexity, where for each item in the list, it goes through the whole list again to find the smallest, compared to that item, and then swaps with that item, it is basically a nested loop. Therefore, although it stops halfway to find the median, it's still the slowest out of all the algorithms, and it is not a good algorithm for sorting a large data set. In our scenario, because we are only doing half selection sort for half of the vector, hence the loop will run n/2 times. So the outer loop will be ½ times the original, and the inner will also be ½ times the original, therefore the time complexity is O(¼ n^2)

Merge Sort and In-Place Merge Sort:

These two sorting algorithms are quite surprising to me as I thought that since they are both utilizing the idea of merge sort, which has the time complexity of O(nlogn), whether they are in place or not wouldn't affect the run time all that much. I was expecting in-place merge sort to be a little faster than standard mergesort, but I wasn't expecting a huge difference. However, as the table has shown, as the size grew larger, the differences between the in-place mergesort and the standard mergesort have grown quite significantly. I think the cause of this might be the time of Standard mergesort required to allocate new memory. For instance, in the standard mergesort, sorting a vector of size 1 million requires the program to allocate memory for two 500,000-sized vectors, which will increase the size dramatically as the size increases.

So, in essence, for standard merge sort, the time complexity increases in N for allocating the vector, hence O(n logn + n). The same goes for in-place merge sort, it is just that since we do not need to allocate memory, we don't need to add the N, hence it is still O(nlogn)

Halfheap Sort

The time complexity for a normal heap sort is O(nlogn). In this sort, we first build heap by percolating down on the first non-leaf node, and we do this operator for all nodes going up until we reach the root. The build heap operation takes the time complexity of O(n). Then, in a normal heapsort, we would continuously delete the root, percdown, and place the root into a new list. When the heap is empty, it means that we have a sorted list. Such operation takes the time complexity of O(nlogn). However, since we are only doing half of the heapsort, this means we only delete the root until half of the heap is gone. Therefore, the time complexity of that would be O(n/2 nlogn + n), the plus n is for the build heap, but its typically neglectable to plus n, hence, the time complexity is O(n/2 nlogn)

Quick Select

Quick Select takes the concept of quicksort, in which itself also similar to mergesort. In quick sort, we break the data into two and recursively sort them on both sides. In quickselect, however, we only recurse on one side. Quick select has the time complexity of O(n) which is the quickest algorithm out of all. But it does have the worst-case time complexity of O(n^2). In our quickselect, we choose our pivot by using the median of 3 and use the Hoares partitioning method. These methods, especially the median of 3, still consist of the worst case of O(n^2), especially with my worse case quickselect as input. But in most cases, it stays on average of O(n)

HalfHeap Sort and Qucickselect

The worst-case time complexity for Heapsort is O(nlogn) and Quickselect is O(n^2), however, Quickselect has the average time complexity of O(n). So, even though heap sort generally has a better or worse case, quickselect will on average still be faster than heapsort since Quickselect is O(n) linear on the average case. Such behavior remains largely the same even after turning heapsort to half heap sort and quicksort to quickselect. Even though the time required should be reduced by half for both, since quickselect gradually sorts the list into smaller and smaller groups on either the left or right side, based on where the middle element of the data set is in comparison to where the pivot eventually land, therefore, quickselect will no doubt be much faster than half heapsort.

Quickselect with MedianofMedian and Quickselect with a median of 3
Quickselect with a median of the median will generally be slower than Quickselect with a median of 3, Quickselect with a median of the median will have a better worst case than Quickselect with a median of 3 as the whole purpose of having a median of median instead of a median of 3 is to get rid of the worse case that is caused by the median of 3 algorithm which causes Quickselect to be O(n^2). Such behaviors are also shown in the table, with quickselect using the median of 3 as the pivot selection, finishing finding the median at around 43 ms with size 1 million, while quickselect uses the median of median as the pivot selection algorithm finishes at around 88 ms. The specific reason for this slowdown is that the algorithm finds the median of the median, it first finds the median of 5 and puts them to the list, and recursively finds the median of 5 again. Doing so, will indeed slow down the algorithm but prevent the worst case of O(n^2)

Worse case Quick Select
Worse case quick select seems to only affect Quickselect, which is foreseeable, as it is a pattern that is created to purposely slow down the median of 3 quick select, and in using worse case quick select as input, it is shown that quick select using the median of 3, does indeed produce the worse time complexity of O(n^2). However, using the median of median as pivot selection for quickselect completely prevents that time complexity of O(n^2) and remains at 1 ms. Proving that it has gotten rid of the worse time complexity of O(n^2) and remained as O(n).