

Program Structures & Algorithms Spring 2023

Assignment No. 5

TASK:

In this assignment, your task is to determine--for sorting algorithms--what is the best predictor of total execution time: comparisons, swaps/copies, hits (array accesses), or something else.

ANSWER:

To identify the best predictor, following analysis has been done (note: all data is given in the respective csv files under assignment 6 folder in github):

Merge Sort:

MERGE SORT:							
No of elements	With instrumentation (millisec)	Without instrumentation (Millisec)	Hits	Copies	Swaps	Compares	
10000	16.6111254	11.8157998	259091.2	9772.8	0	121506.6	
20000	28.50805	9.4597916	558159.2	19539.8	0	263064.4	
40000	21.525225	18.363867	1196199.2	39049.8	0	565954.6	
80000	37.0881834	26.6034668	2552395.2	78098.8	0	1212045.6	
160000	68.7268502	71.8383	5424253.6	156063.4	0	2584144.4	

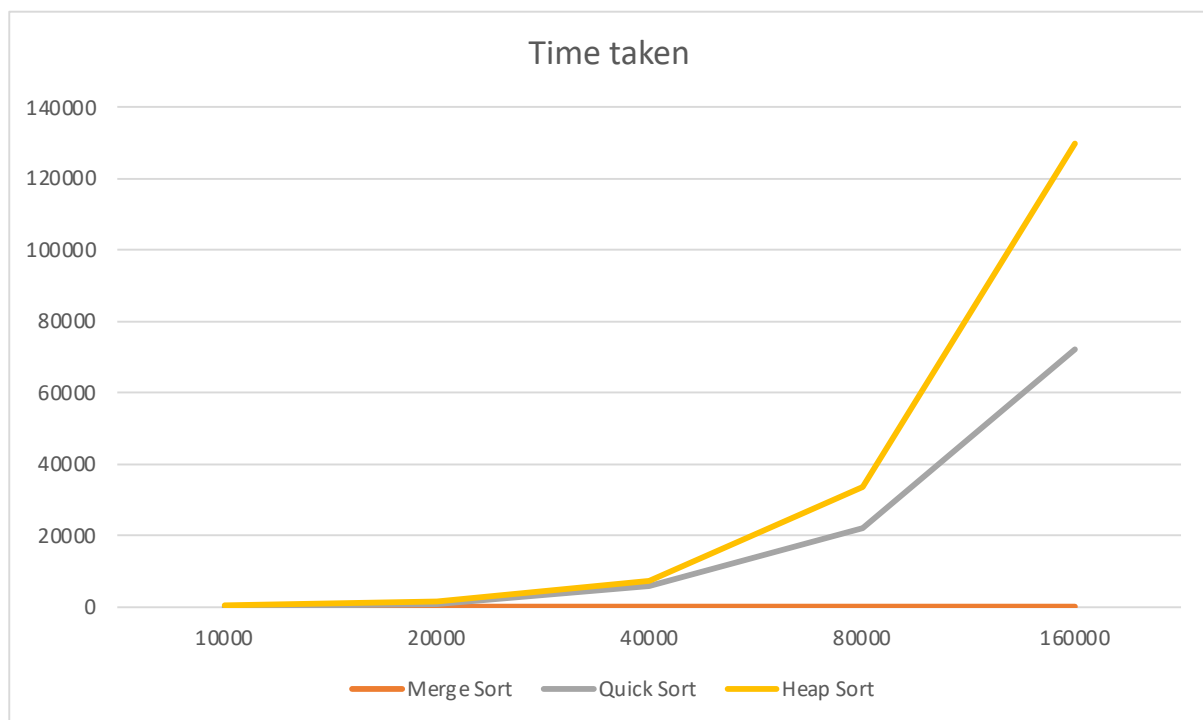
Heap Sort:

HEAP SORT:							
No of elements	With instrumentation (millisec)	Without instrumentation (Millisec)	Hits	Copies	Swaps	Compares	
10000	386.37055	11.1339666	967718.4	0	124223.6	235412	
20000	1738.65645	11.7096002	2095556.4	0	268464.6	510849	
40000	7240.264483	22.5612584	4509526	0	576670.6	1101421.8	
80000	33774.65812	43.9586748	9661624.4	0	1233814.6	2363183	
160000	129859.9997	128.5296998	20600000	0	2627287	5046041	

Quick Sort:

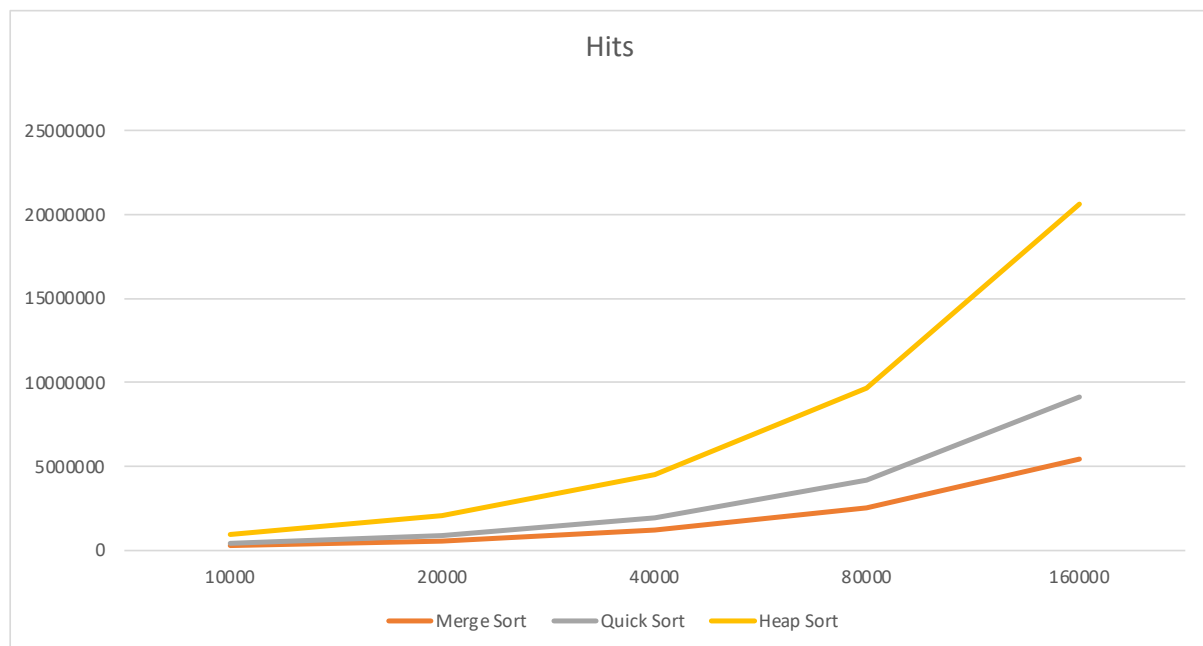
QUICK SORT:							
No of elements	With instrumentation (millisec)	Without instrumentation (Millisec)	Hits	Copies	Swaps	Compares	
10000	263.4287416	3.9055504	411597.8	0	64000.2	159316.8	
20000	930.1212252	13.5422256	858028.4	0	132866.4	333794	
40000	6021.12425	52.1592748	1936224.2	0	301206	746156.4	
80000	22058.78133	22.658675	4157034.2	0	648538.8	1592240	
160000	72146.56801	49.13245	9121877.8	0	1437622.8	3430107.2	

Comparing all the runtimes, we get the following graphs:



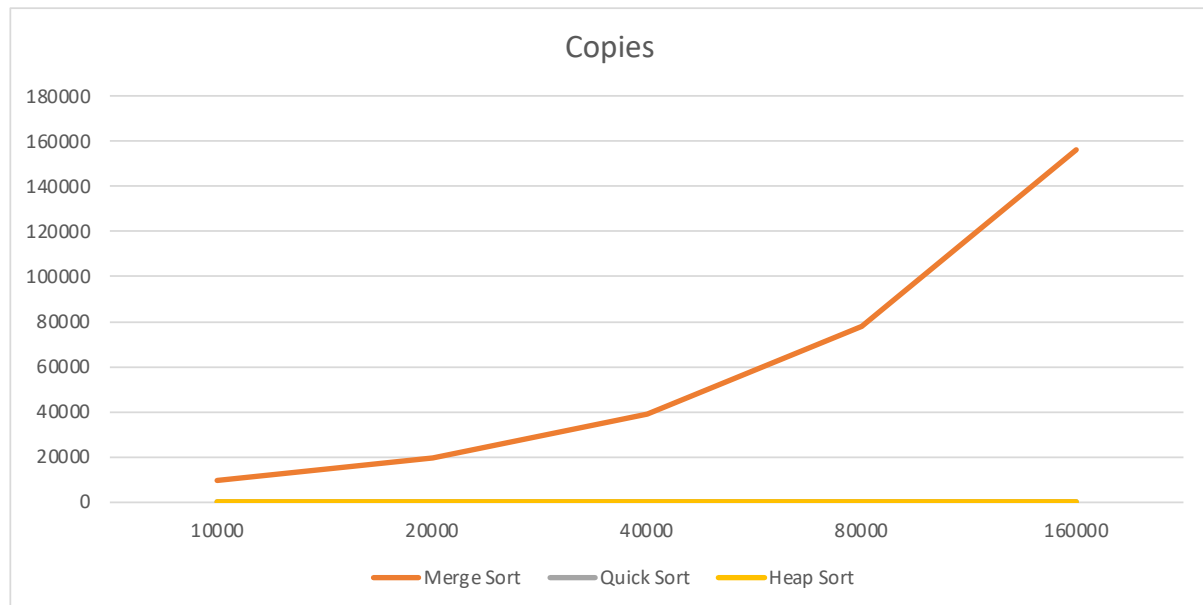
Looking at the above graph, we notice that time increases for every algorithm as the input size increases. For merge sort, it is a straight graph while for quick sort and heap sort, it is predominant as the input size increases.

Comparing all the hits (array accesses):



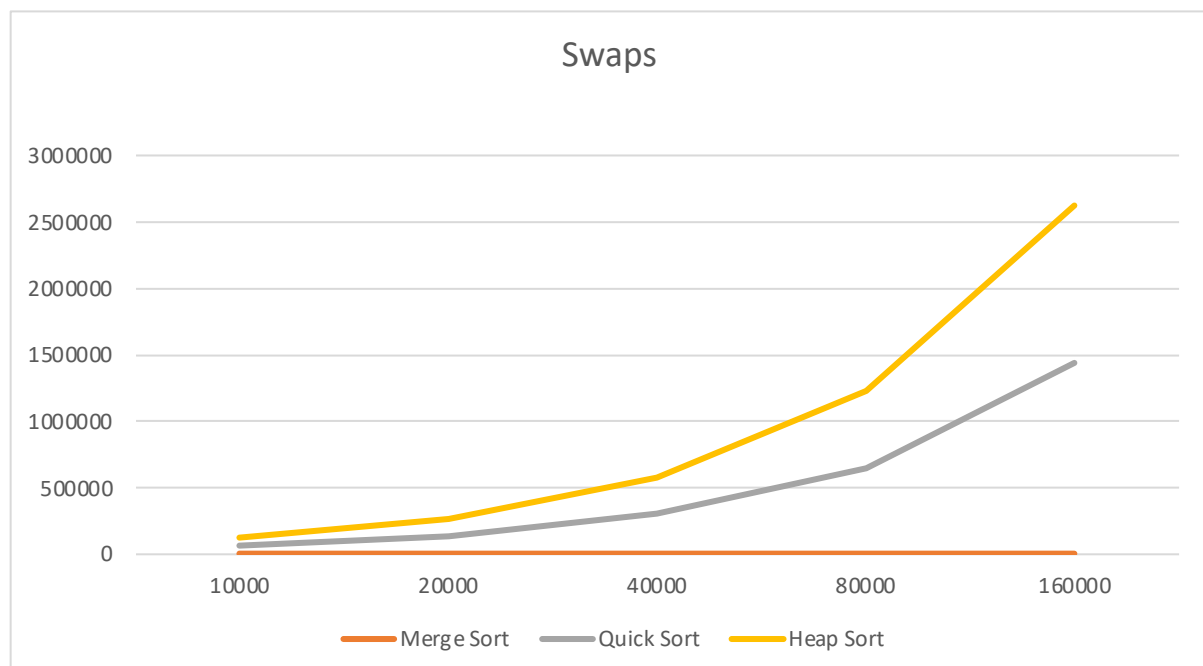
We can see that the effect of hits is high for heap sort as the input size increases, while for quick sort it seems to perform better compared to merge sort for smaller input size but merge sort comes out better for larger input sizes.

Comparing all the copies:



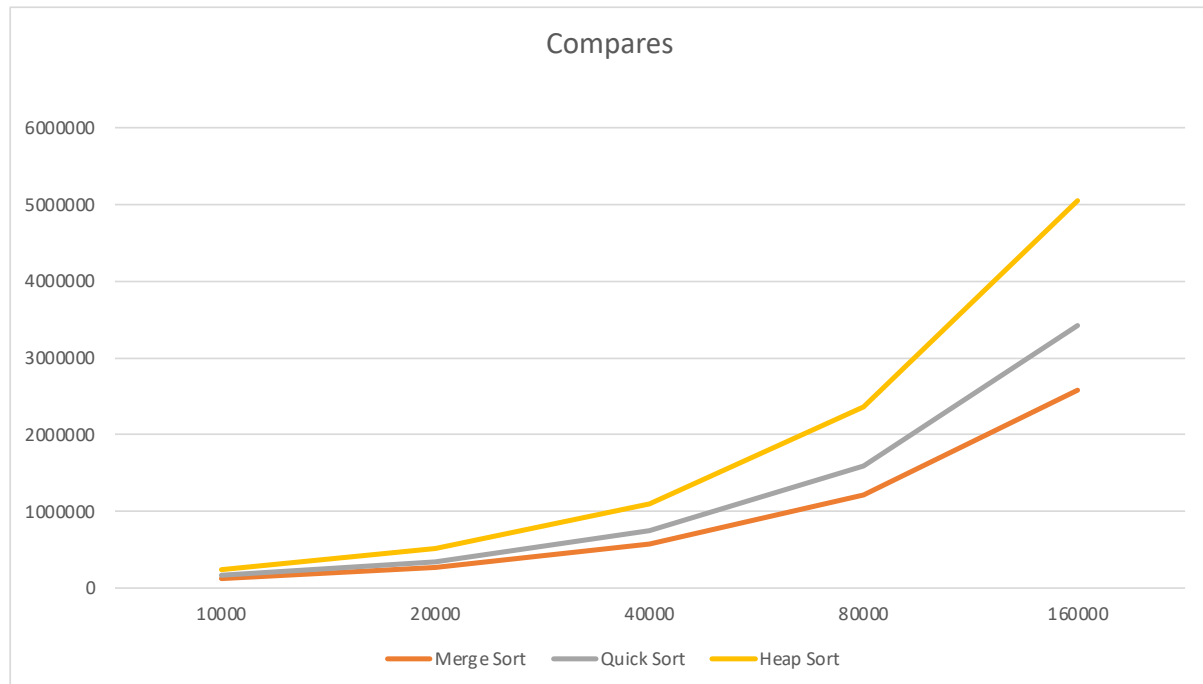
Copies are only used in merge sort, while quick sort and heap sort uses swaps. So only merge sort number of copies increase while input size increases.

Comparing all the swaps:



As the number of elements increases, there is a linear increase in the number of swaps required for sorting using comparison-based algorithms. Merge sort, being a non-comparison based algorithm, does not involve any swaps and therefore its swap count is 0. Among the comparison-based algorithms, heap sort requires the highest number of swaps, while quick sort requires a relatively lower number of swaps compared to heap sort.

Comparing all the compares:



We can see that the number of compares is the highest for heap sort followed by quick sort and least for merge sort.

CONCLUSION:

Based on the above analysis, the following conclusion can be arrived at:

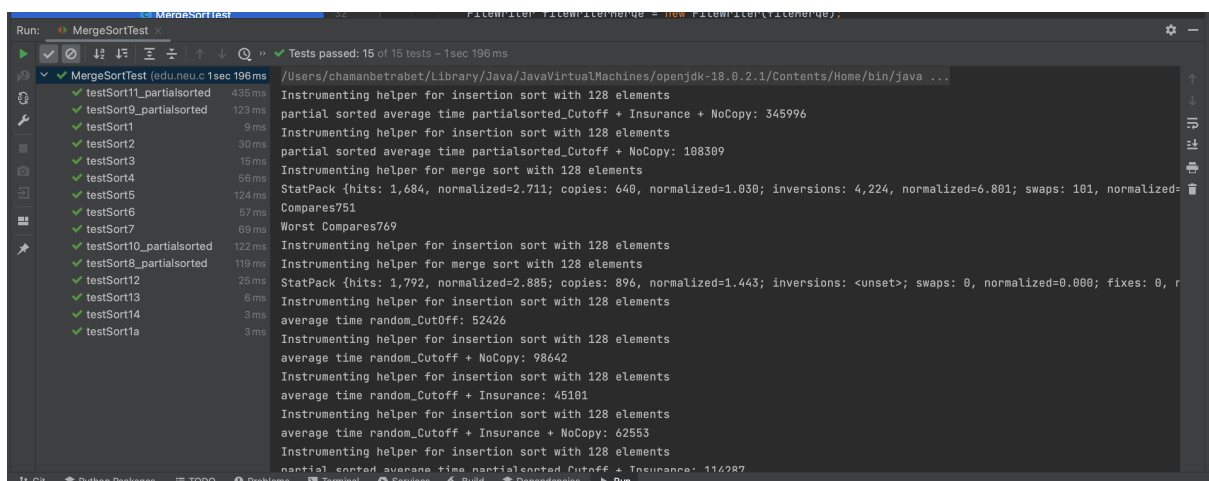
The quantity of comparisons and swaps/copies is typically a reliable indicator of the overall execution time for quick sort. Fast sort employs a divide-and-conquer strategy and heavily relies on comparisons to establish an element's proper place in the sorted sequence. A lot of swaps and duplicates are also carried out by rapid sort in order to shift items into their final placements.

The amount of swaps/copies is probably the best indicator of the overall execution time for heap sort. Building a heap data structure and continually exchanging the root element with the last element in the heap until the heap is sorted are both steps in the heap sort process. This method requires a large number of swaps and copies, which may slow down execution.

The number of comparisons is frequently the most accurate indicator of the overall execution time for merge sort. Recursively sorting the input data into smaller subarrays, merging them back together, and repeating this process is known as merge sort. The algorithm analyses elements from the two subarrays to ascertain their relative order during the merging phase. In contrast to quick sort and heap sort, merge sort doesn't include any swaps or copies, which can lead to quicker execution speeds.

TEST CASE RESULTS:

Merge Sort:

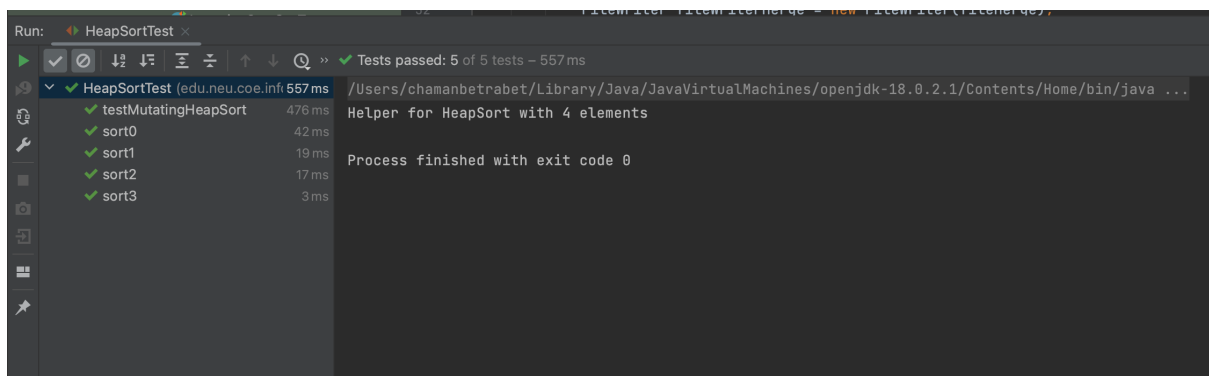


```
Run: MergeSortTest x
Tests passed: 15 of 15 tests - 1sec 196ms

MergeSortTest (edu.neu.coe.inf) 1sec 196ms
  testSort11_partialsorted 436ms
  testSort9_partialsorted 123ms
  testSort1 9ms
  testSort2 30ms
  testSort3 15ms
  testSort4 56ms
  testSort5 124ms
  testSort6 57ms
  testSort7 69ms
  testSort10_partialsorted 122ms
  testSort8_partialsorted 119ms
  testSort12 25ms
  testSort13 6ms
  testSort14 3ms
  testSort1a 3ms

Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + Insurance + NoCopy: 345996
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + NoCopy: 108309
Instrumenting helper for merge sort with 128 elements
StatPack {hits: 1,684, normalized=2.711; copies: 640, normalized=1.030; inversions: 4,224, normalized=6.801; swaps: 181, normalized=0.000; fixes: 0, normalized=0.000}
Worst Compares769
Instrumenting helper for insertion sort with 128 elements
Instrumenting helper for merge sort with 128 elements
StatPack {hits: 1,792, normalized=2.888; copies: 896, normalized=1.443; inversions: <unset>; swaps: 0, normalized=0.000; fixes: 0, normalized=0.000}
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff: 52426
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + NoCopy: 98642
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + Insurance: 45101
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + Insurance + NoCopy: 62553
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + Insurance: 114287
```

Heap Sort:

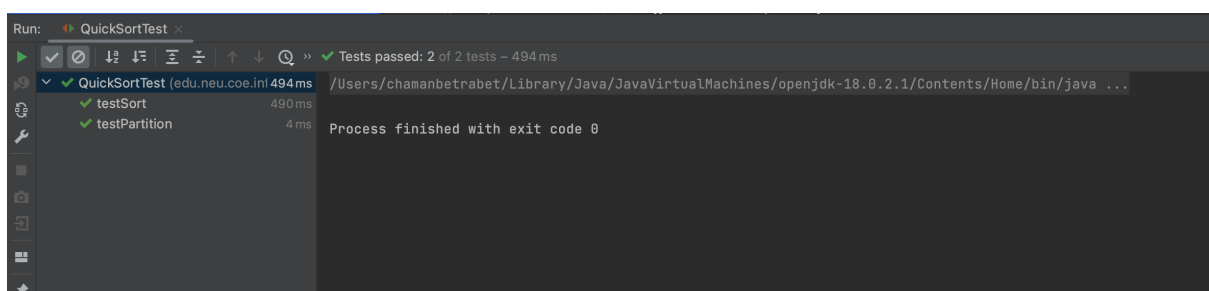


```
Run: HeapSortTest x
Tests passed: 5 of 5 tests - 557ms

HeapSortTest (edu.neu.coe.inf) 557ms
  testMutatingHeapSort 476ms
  sort0 42ms
  sort1 19ms
  sort2 17ms
  sort3 3ms

Helper for HeapSort with 4 elements
Process finished with exit code 0
```

Quick Sort:



```
Run: QuickSortTest x
Tests passed: 2 of 2 tests - 494ms

QuickSortTest (edu.neu.coe.inf) 494ms
  testSort 490ms
  testPartition 4ms

Process finished with exit code 0
```