# Program Structures and Algorithms

## Spring 2023(SEC-01)

### Assignment 6

Name: Raja Shekar Reddy Siriganagari

NUID: 002762145

**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.

You will run the benchmarks for merge sort, (dual-pivot) quick sort, and heap sort. You will sort randomly generated arrays of between 10,000 and 256,000 elements (doubling the size each time). If you use the SortBenchmark, as I expect, the number of runs is chosen for you. So, you can ignore the instructions about setting the number of runs.

For each experiment (a sort method of a given size), you will run it twice: once for the instrumentation, once (without instrumentation) for the timing.

Of course, you will be using the Benchmark and/or Timer classes, as you did in a previous assignment.

You must support your (clearly stated) conclusions with evidence from the benchmarks (you should provide log/log charts and spreadsheets typically).

**Observations:**

Metrics used for benchmarking:

**Array sizes**: 500,000 to 8,000,000 using doubling method.

**Array elements order**: All elements are shuffled using *Collections.shuffle()* before sorting.

**Sort algorithms**: **Quick sort dual pivot** implementation, **Merge sort** and **Heap sort**

Following are the timings observed as part of the experiment:

Table 1: Raw times for different sorting algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| Number of elements(N) |  | Raw time (in ms) |  |
| **QuickSortDualPivot** | **MergeSort** | **HeapSort** |
| 10000 | 3.2 | 3.23 | 1.73 |
| 20000 | 2.5 | 3.6 | 3.71 |
| 40000 | 4.65 | 7.7 | 7.87 |
| 80000 | 10.19 | 15.95 | 17.36 |
| 160000 | 31.24 | 44.99 | 50.4 |
| 320000 | 71.46 | 112.5 | 154.74 |
| 640000 | 376.04 | 400.91 | 575.74 |
| 1280000 | 593.45 | 841.82 | 1816.99 |
| 2560000 | 1159.15 | 1512.74 | 3218.24 |

We can clearly see that the Dual Pivot Quick Sort implementation outperforms the other two linearithmic sort algorithms.

**Analysis:**

* QuickSortDualPivot accesses the array least number of times(hits) when compared to the other two sorting algorithms. Although QuickSort does a greater number of comparisons compared to MergeSort it avoids unnecessary element swaps/copies leading to a smaller number of array accesses. QuickSort fixes a huge number of inversions per swap/copy as it is not a stable sorting algorithm. This is evident from the table below which shows the metrics for sorting an array of size 320,000 over a 100 runs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm |  | Metrics |  |  |
| **Hits** | **Compares** | **Swaps** | **Copies** |
| QuickSortDualPivot | 19,157,579 | 7,227,303 | 2,960,512 | 0 |
| MergeSort | 11,833,205 | 5,488,161 | 312,364 | 5,120,000 |
| HeapSort | 43,761,174 | 10,731,892 | 5,574,347 | 0 |

* MergeSort is the next best algorithm amongst the three. This version of the algorithms has no optimizations to it. At lower sizes (i.e., 16) we are switching to insertion sort to avoid performance overheads. Even though the number of compares is less merge sort is slower as it uses an extra auxiliary array to merge elements. The overhead to copy the elements to the auxiliary array and merge them back to the main array is very high. And as MergeSort and InsertionSort(used for smaller sized arrays) are stable sorting algorithms they require more number of swaps and copies to achieve a sorted array result.
* HeapSort offers a decent time as well but is the worst of the three algorithms. The time taken to construct a heap is a lot and it requires a huge number of comparisons and swaps. Irrespective of the ordering of the array HeapSort does O(nlogn) comparisons whereas this is much better for QuickSort and MergeSort depending on the ordering of the array.
* For random arrays, QuickSortDualPivot outperforms the other two sorting algorithms as it moves the elements less across the array than the other algorithms requiring less array accesses. The number of **hits (array accesses) which is affected by the number of compares and swaps is a defining factor for the time of the sort algorithms**. Swaps account for most of the hits and is an expensive operation.
* As the array size increases the number of hits also increase and as most of the array cannot be accommodated in small caches on a device it results in page faults. The array element should then be fetched from memory which takes a longer time. So, limiting the number of times we access an array is important for a fast algorithm.

**Evidence:**

Following is a graph for the array size vs the raw time to run these algorithms:

Chart, line chart

Description automatically generated

A log-log chart for the same values confirms the trend:Chart, line chart

Description automatically generated

**Conclusion:**

So, the number of time you access the memory to fetch values determines the amount of time taken by the sort algorithm to complete. QuickSortDP does this the least. So, it’s execution time is faster compared to others. Depending upon the hardware compares may be faster than swaps or slower. Irrespective of that both involve accessing memory which is always a time-consuming operation than computations.

Sort algorithms involving more read/write operations than computations consume the highest amount of time as the computations are almost trivial for all the latest sort algorithms and advanced processors.

**Test cases:**

**Text

Description automatically generated**

**Text

Description automatically generated**

**Text

Description automatically generated**

**Text

Description automatically generated**

**Text

Description automatically generated**