Hubert Wrona 15008

Lab3/2/PROGN

Experiment Description:

The goal of the experiment was to empirically examine the efficiency of four sorting algorithms: **InsertionSort**, **MergeSort**, **QuickSortClassical**, and **QuickSort** (a heuristic variant from the C# library).

Implementation:

1. **Algorithm implementation based on the provided GeeksforGeeks links:**
    - InsertionSort (https://www.geeksforgeeks.org/insertion-sort/)
    - MergeSort (https://www.geeksforgeeks.org/merge-sort/)
    - QuickSortClassical (https://www.geeksforgeeks.org/quick-sort/)
    - QuickSort (heuristic variant implemented in C# – Array.Sort())

2. **Benchmark environment setup** – a console application with the BenchmarkDotNet library added. Parameters were used to gather as much performance data as possible.

3. **Implementation of static methods** for generating test data:
    - GenerateRandom
    - GenerateSorted
    - GenerateReversed
    - GenerateAlmostSorted
    - GenerateFewUnique

Results:

Method	ArraySize	ArrayType	Mean	Error	StdDev	Median	Gen0	Gen1	Gen2	Allocated
InsertionSort	10	Almost sorted	10.445 ns	0.2591 ns	0.3716 ns	10.348 ns	-	-	-	-
MergeSort	10	Almost sorted	271.584 ns	5.4513 ns	11.6172 ns	269.210 ns	0.1183	-	-	992 B
QuickSortClassical	10	Almost sorted	208.547 ns	4.2056 ns	11.4416 ns	204.426 ns	0.0744	-	-	624 B
QuickSort	10	Almost sorted	13.364 ns	0.3184 ns	0.5406 ns	13.244 ns	-	-	-	-
InsertionSort	10	Few unique	10.526 ns	0.2576 ns	0.3777 ns	10.564 ns	-	-	-	-
MergeSort	10	Few unique	251.749 ns	5.0510 ns	5.8167 ns	251.356 ns	0.1183	-	-	992 B
QuickSortClassical	10	Few unique	208.485 ns	3.9768 ns	3.9058 ns	207.925 ns	0.0763	-	-	640 B
QuickSort	10	Few unique	12.958 ns	0.3145 ns	0.5829 ns	12.891 ns	-	-	-	-
InsertionSort	10	Random	9.272 ns	0.2419 ns	0.3837 ns	9.277 ns	-	-	-	-
MergeSort	10	Random	255.049 ns	4.4910 ns	4.2009 ns	255.764 ns	0.1183	-	-	992 B
QuickSortClassical	10	Random	231.655 ns	4.6391 ns	4.9638 ns	231.492 ns	0.0842	-	-	704 B
QuickSort	10	Random	12.693 ns	0.2187 ns	0.2046 ns	12.744 ns	-	-	-	-
InsertionSort	10	Reversed	8.569 ns	0.1678 ns	0.1310 ns	8.596 ns	-	-	-	-
MergeSort	10	Reversed	255.662 ns	5.1618 ns	11.9633 ns	254.962 ns	0.1183	-	-	992 B
QuickSortClassical	10	Reversed	218.497 ns	4.1028 ns	3.4260 ns	218.655 ns	0.0792	-	-	664 B
QuickSort	10	Reversed	13.480 ns	0.3247 ns	0.4445 ns	13.415 ns	-	-	-	-
InsertionSort	10	Sorted	9.353 ns	0.2443 ns	0.5807 ns	9.303 ns	-	-	-	-
MergeSort	10	Sorted	253.847 ns	4.8699 ns	4.7829 ns	254.754 ns	0.1185	-	-	992 B
QuickSortClassical	10	Sorted	222.323 ns	4.3726 ns	4.0902 ns	222.137 ns	0.0801	-	-	672 B
QuickSort	10	Sorted	12.958 ns	0.3100 ns	0.5006 ns	12.932 ns	-	-	-	-
InsertionSort	1000	Almost sorted	823.875 ns	13.3416 ns	13.1033 ns	820.726 ns	-	-	-	-
MergeSort	1000	Almost sorted	50,376.162 ns	835.1747 ns	740.3606 ns	50,399.469 ns	18.6768	0.2441	-	156632 B
QuickSortClassical	1000	Almost sorted	68,819.847 ns	1,366.0948 ns	1,626.2388 ns	68,351.111 ns	26.6113	5.6152	-	223192 B
QuickSort	1000	Almost sorted	3,415.738 ns	67.4543 ns	80.2996 ns	3,433.344 ns	-	-	-	-
InsertionSort	1000	Few unique	824.972 ns	16.4588 ns	16.9020 ns	824.348 ns	-	-	-	-
MergeSort	1000	Few unique	51,910.747 ns	769.9199 ns	682.5140 ns	52,000.699 ns	18.6768	0.2441	-	156632 B
QuickSortClassical	1000	Few unique	55,430.507 ns	1,078.4925 ns	956.0555 ns	55,559.998 ns	22.0337	3.4790	-	184392 B
QuickSort	1000	Few unique	3,361.875 ns	64.7159 ns	88.5838 ns	3,349.190 ns	-	-	-	-
InsertionSort	1000	Random	831.318 ns	16.4685 ns	20.2248 ns	824.825 ns	-	-	-	-
MergeSort	1000	Random	49,782.556 ns	990.4165 ns	1,451.7397 ns	49,857.648 ns	18.6768	0.2441	-	156632 B
QuickSortClassical	1000	Random	59,190.917 ns	1,155.6952 ns	1,375.7730 ns	59,158.289 ns	22.1558	3.9673	-	185624 B
QuickSort	1000	Random	3,335.101 ns	66.5698 ns	103.6412 ns	3,314.800 ns	-	-	-	-
InsertionSort	1000	Reversed	846.665 ns	16.9974 ns	22.1014 ns	844.176 ns	-	-	-	-
MergeSort	1000	Reversed	50,244.926 ns	1,001.4976 ns	1,754.0424 ns	50,504.773 ns	18.6768	0.2441	-	156632 B
QuickSortClassical	1000	Reversed	75,090.292 ns	1,435.5471 ns	1,708.9168 ns	75,593.884 ns	28.5645	6.3477	-	239744 B
QuickSort	1000	Reversed	3,396.583 ns	67.1070 ns	65.9080 ns	3,400.411 ns	-	-	-	-
InsertionSort	1000	Sorted	834.600 ns	12.1622 ns	11.3766 ns	831.808 ns	-	-	-	-
MergeSort	1000	Sorted	49,421.374 ns	974.6320 ns	1,854.3389 ns	49,361.365 ns	18.6768	0.2441	-	156632 B
QuickSortClassical	1000	Sorted	52,079.994 ns	1,010.0612 ns	1,202.4061 ns	52,346.207 ns	19.2261	2.6245	-	161144 B
QuickSort	1000	Sorted	3,477.421 ns	66.2291 ns	76.2696 ns	3,471.544 ns	-	-	-	-
InsertionSort	100000	Almost sorted	83,181.705 ns	1,281.9031 ns	1,070.4472 ns	83,269.531 ns	-	-	-	-
MergeSort	100000	Almost sorted	10,649,690.527 ns	205,859.1701 ns	202,181.2310 ns	10,664,165.625 ns	2484.3750	734.3750	609.3750	21262603 B
QuickSortClassical	100000	Almost sorted	29,119,515.296 ns	573,215.3016 ns	637,127.2890 ns	29,214,087.500 ns	3031.2500	2093.7500	1468.7500	50097779 B
QuickSort	100000	Almost sorted	762,524.567 ns	14,863.8035 ns	18,797.9538 ns	763,717.383 ns	-	-	-	-
InsertionSort	100000	Few unique	83,658.415 ns	1,662.6510 ns	1,632.9456 ns	83,691.302 ns	-	-	-	-
MergeSort	100000	Few unique	10,645,812.012 ns	208,673.5997 ns	204,945.3772 ns	10,655,001.562 ns	2484.3750	734.3750	609.3750	21262603 B
QuickSortClassical	100000	Few unique	22,998,748.864 ns	412,563.6866 ns	506,665.3020 ns	22,923,395.312 ns	3000.0000	2031.2500	1343.7500	38350944 B
QuickSort	100000	Few unique	752,985.916 ns	14,565.4922 ns	20,418.7875 ns	754,912.012 ns	-	-	-	-
InsertionSort	100000	Random	83,707.879 ns	1,668.0961 ns	2,048.5720 ns	83,559.906 ns	-	-	-	-
MergeSort	100000	Random	10,609,169.043 ns	207,065.8400 ns	203,366.3422 ns	10,636,076.562 ns	2484.3750	734.3750	609.3750	21262603 B
QuickSortClassical	100000	Random	23,371,461.433 ns	465,011.6655 ns	1,233,145.0600 ns	23,467,678.125 ns	3093.7500	2218.7500	1468.7500	40445878 B
QuickSort	100000	Random	750,232.978 ns	12,960.6038 ns	10,822.6912 ns	753,260.547 ns	-	-	-	-
InsertionSort	100000	Reversed	83,362.007 ns	1,630.3593 ns	1,525.0391 ns	83,255.640 ns	-	-	-	-
MergeSort	100000	Reversed	10,745,091.866 ns	210,758.1640 ns	340,335.1768 ns	10,674,703.906 ns	2484.3750	734.3750	609.3750	21262603 B
QuickSortClassical	100000	Reversed	21,611,810.859 ns	427,039.5797 ns	759,062.4790 ns	21,526,068.750 ns	2968.7500	2031.2500	1312.5000	36420060 B
QuickSort	100000	Reversed	744,080.156 ns	12,191.8868 ns	11,404.2985 ns	744,318.457 ns	-	-	-	-
InsertionSort	100000	Sorted	82,569.033 ns	1,335.0523 ns	1,183.4891 ns	82,499.365 ns	-	-	-	-
MergeSort	100000	Sorted	10,353,074.888 ns	195,772.4494 ns	173,547.1771 ns	10,331,257.031 ns	2484.3750	734.3750	609.3750	21262603 B
QuickSortClassical	100000	Sorted	17,730,408.628 ns	349,054.6370 ns	441,442.3914 ns	17,662,635.938 ns	2781.2500	1843.7500	1218.7500	31392898 B
QuickSort	100000	Sorted	721,690.632 ns	14,313.0522 ns	13,388.4379 ns	720,573.145 ns	-	-	-	-

Conclusions

1. **Performance Scaling with Input Size**
    - **InsertionSort** performs exceptionally well on small arrays (10 elements), with execution times in **single-digit nanoseconds**. However, its efficiency drops significantly for larger datasets (100,000 elements), reaching **~83 µs**, making it unsuitable for big data.
    - **MergeSort** and **QuickSortClassical** show **O(n log n)** behavior but with high memory overhead. MergeSort maintains stable performance across different input types, while QuickSortClassical varies more, especially in worst-case scenarios (e.g., reversed arrays).
    - **QuickSort (C# Array.Sort())** is the **fastest** for all array sizes, even for 100,000 elements (~750 µs), demonstrating superior optimization in the C# library.
2. **Impact of Input Type**
    - **InsertionSort** works best on **already sorted or nearly sorted** data (minimal swaps needed).
    - **QuickSortClassical** struggles with **reversed arrays** (slowest case), while **MergeSort** remains consistent.
    - **Few unique values** slightly improve QuickSortClassical's performance compared to random data but do not significantly affect MergeSort.
3. **Memory Efficiency**
    - **InsertionSort** and **QuickSort (C#)** allocate **no additional memory** (in-place).
    - **MergeSort** and **QuickSortClassical** consume **substantial memory** (e.g., ~21 MB for 100k elements), with QuickSortClassical being less efficient due to recursive stack usage.

Summary:

- **Best for speed:** QuickSort (C#)
- **Best for stability:** MergeSort
- **Avoid for large data:** InsertionSort and QuickSortClassical (due to time/memory inefficiency).