BALTIC JOURNAL OF LAW & POLITICS
A Journal of Vytautas Magnus University
VOLUME 16, NUMBER 3 (2023)
ISSN 2029-0454

Cite: Baltic Journal of Law & Politics 16:3 (2023): 2496-2507
DOI: 10.2478/bjlp-2023-00000189

Enhancement of Classical Shell-Sort Algorithm Performance by Tuning in Partition Number based on Number of Input Data

Tedjo Darmanto
Informatics Engineering, Islamic University of Nusantara, Bandung, West Java, Indonesia
Email: tedjodarmanto@uninus.ac.id

Yusuf Iskandar
Informatics Engineering, Islamic University of Nusantara, Bandung, West Java, Indonesia
Email: yusufiskandar@uninus.ac.id

Fahmi Fauzi Al-Paridi
Informatics Engineering, Islamic University of Nusantara, Bandung, West Java, Indonesia
Email: fahmifauzi@uninus.ac.id

Wahyu Zaidi Arswenri
Informatics Engineering, Islamic University of Nusantara, Bandung, West Java, Indonesia
Email: wahyuzaidi@uninus.ac.id

M.Syani Fadillah
Informatics Engineering, Islamic University of Nusantara, Bandung, West Java, Indonesia
Email: muhammadSyani@uninus.ac.id

Received: December 24, 2022; reviews: 2; accepted: January 21, 2023

Abstract

To enhance the performance of shell-sort algorithm that has the best performance experimentally compared to other four classical sorting algorithms can be accomplished by adjusting the partition number as a tune in parameter based on the number of input data to be sorted.

Keywords

shell-sort algorithm, algorithm performance, tune in parameter, partition number, sorting algorithm comparison

Introduction

Sorting algorithm is still open to be discussed. The comparison among the classical sorting algorithms is always interesting to be analyzed. Some of them still can be tuned in their algorithm performance to be enhanced. The specification and the CPU utilization of machine used in the experiment in this paper can be seen in Figure-1 and 2 respectively.

There are many researchers presenting their paper related to the sorting algorithm comparison and the combination of two or more algorithms to enhance their performance. Chadha, A.R. et.al. proposed ARC Sort which has better performance compared to selection, insertion and bubble aborting algorithms by reducing the number of unnecessary comparison and swaps through the use of 2 dimensional arrays [1]. Alyasseri, Z.A.A. et.al. proposed the use of message passing interface (MPI) to enhance the parallelize bubble and merge sort algorithms that depend on the data structure approach [2]. Mohammed, A.S. et.al. proposed bidirectional conditional insertion sort (BCIS) which has better performance compared to the classical insertion sort and even compared to quicksort [3]. Abhay, G. et.al. proposed a method to combine two string algorithms, the bucket and shell sort algorithms in a way taking the advantage of strength of each to improve overall performance [4]. Fitro, A. compared three data sorting algorithm, insertion sort, quick sort and merge sort with random data to be sorted ranging from 1000 to 20,000 and showing that the insertion sort is better than the others and the merge sort is the worst among them by using MathLab programming language [5]. Alnihoud, J. & Mansi, R. proposed two new enhancement sorting algorithms, the selection sort and bubble sort by reducing the number of swaps and consequently reducing time of execution [6]. Wang, Z. et.al. accomplished an experimental study of an optimization and analysis of a large scale data sorting algorithm based on Hadoop by using more than two rounds of MapReduce [7]. Markina, M. & Buzdalov, M combined merge-sort and insertion sort or combine the divide-and-conquer algorithm and the best order sort algorithm applied to non-dominated sorting and compared to the original algorithms for large numbers data showing the performance better at least 20% [8]. Chen, F. et.al. developed an efficient sorting algorithm namely as ultimate heapsort (UHS) which has two parts, building and adjusting heaps. Through the asymptotic analysis and experimental analysis, time complexity of the algorithm can reach n.log(n) and space complexity is only O(1) and showing superiority to all previous algorithms [9]. Shirazi, S.M & Bagheri, A. applied sorting algorithm on finding minimum feedback arc set problem in tournaments or FAST problem by introducing pseudo insertion sort and merge sort algorithms that showed the average number of all backward edges in output

decreased [10]. Olukanmi, P. et.al. presented a useful connection between sorting and clustering that led to a general approach for accelerating sorting algorithms as an effective way to implement the divide-and-conquer strategy. They proposed a simplified one-pass k-means-type algorithm comprising a single iteration of Lloyds standard k-means algorithm and proved the improvement experimentally against the insertion sort, bubble sort and selection sort [11]. Durrani, O.K. & Hayan, S.A. implemented popular sorting algorithm using C++, Python and Java languages showing merge sort doing well in Python instead of quick sort, but in Java quicksort still doing well and in C++ all performing as per the verified behaviors. There are the indicated some miss behaviors made by Java and Python implementation results [12].

The special discussion on the sorting algorithm topic, especially on the quick sort algorithm discussed by many researchers, also on another popular sorting algorithm such as selection sort and a new technique. Roychoudhury, J. & Yadav, J. proposed efficient algorithms for sorting in trees by improving the lower and upper bounds on partially ordered sets [13]. Hayfron-Acquah, J.B & Riverson, K. proposed an improved selection sort algorithm by taking the distinct values in the dataset into consideration instead of solely based on the size of the list, especially when there are redundancies in the list [14]. Juge, V. presented a new sorting algorithm, called adaptive ShiversSort, that exploits the existence of monotonic runs for sorting efficiently partially sorted data as the variant of well-known algorithm the TimSort or k-aware merge-sort algorithm [15]. Singh, N.K. & Chakraborty, S. proposed a smart sort as a robust version of quick sort algorithm by doing fusion of heap construction procedures of heap sort algorithm into the conventional partition function of quick sort algorithm [16]. Jha, S.K. revisited the calculation of moments of number of comparisons used by the randomized quick sort algorithm helping in calculating these quantities with less computation [17]. Fasha, M. presented a comparison for the performance of sequential sorting algorithms under four different modes of execution, the sequential processing mode, a conventional multi-threading implementation and multi-threading with OpenMP Library as parallel computation on a supercomputer. The quick sort algorithm was selected to run the experiments performed by this effort [18]. Wang, D. proposed an efficient implementation of the quick sort algorithm based on java multi-thread technology for multi-core computer systems. On dual-core CPU performance of the implementation increased by about 40% and better performance on the more core of computer [19]. Pathak, N. & Tiwari, S. presented the work on the selection sorting technique for double ended selection sort, that programmatically analysis show performance enhancement due to the selection of the minimum and maximum elements accomplished simultaneously [20].

Methods

The method of this paper can be divided into three steps. The first step is selection of the best sort algorithm among four classical sort algorithms. The

second step is determining the key factor of the algorithm that can be adjusted as a parameter. The last step is the selection of the optimum parameter implementation based on the key factor. The comparison of four known classical sorting algorithms (insertion, quick, shell and selection sort algorithms) as the starting point of discussion is presented in Figure-4 at the Appendix based on the series of their running time with different numbers of input data by java programming experiment. It seems the shell-sort algorithm is superior compared to other sorting algorithms. Let's look closely at the shell-sort algorithm represented by the coding in the Java programming language as can be seen in Figure-1 below. The partition number p can be chosen to be the key factor that can be adjusted based on the number of input data.

```
static void shell (int [] a, int p) {
int in, out, h=1;
while (h <= a.length /p) h=h*p+1;
while (h>0) {
for (out=h; out < a.length; out++) {
int t=a[out]; in=out;
while (in > h-1 && a[in-h]>=t) {
a[in]=a[in-h]; in-=h;
}//end of while
a[in]=t;
}//end of for
h=(h-1)/p;
}//end of while
}//end of shell sort procedure
//p=partition number
```

Figure-1. The code portion of shell-sort algorithm in java coding with partition number as one of two parameters as key factor

To have the optimal value of the partition number, the partition number as a variable can be adjusted and the running time of execution can be displayed along with the variation numbers of input data. Based on the result of experiments with a series of random data as the input, the best performance of the execution which has the lowest running time of execution then the partition number value can be selected as the optimum one as discussed more detail in the results and discussion below.

Results and discussion

As already mentioned above, a series of experiments is conducted with the result showing that the optimal partition number depends on the number of random data to be sorted accordingly. The performance of shell-sort based on the tune in the number of partitions with various numbers of data can be observed in three

clusters as represented by three tables at the Appendix. Table-1 as the representation of the low number of input data, table-2 as the representation of the medium number of input data and table-3 as the representation of the high number of input data.

The value of partition number 5 looks superior compared to the value of partition number 4 and 6 in the case of a low number of input data (see the first column and last row of table-1). The value of partition number 4 looks superior compared to the value of partition number 5 and 6 in the case of medium number of input data (see the first column and last row of table-2). The value of partition number 5 looks superior compared to the value of partition number 4 and 6 in the case of a high number of input data (see the first column and last row of table-3).

Conclusion

The performance of the shell-sort algorithm still can be enhanced by tuning in the partition number parameter in it based on the number of input data to be sorted. From the majority point of view in the case of number of input data, it is looked like the value of partition number 5 is the optimum one based on the experimental result.

References

1. Chadha, A.R., Misal, R., Mokashi, T., Chadha, A.ARC Sort: Enhanced and Time Efficient Sorting Algorithm. *International Journal of Applied Information Systems* (IJAIS), ISSN: 2249-0868, Volume-7, No.2, 2014
2. Alyassei, Z,A.A., Al-Attar, K., Nasser, M., Ismail, Parallelize Bubble and Merge Sort Algorithms using Message Passing Interface (MPI). 2014, https://arxiv.org/ pdf/1411.5283.pdf
3. Mohammed, A.S., Amrahov, S.E., Celebi, F,V. Bidirectional Insertion Sort Algorithm – An efficient progress on the classical insertion sort. 2016, https://arxiv.org/pdf /1608.02615.pdf
4. Abhay, G., Abhishek, S., Namita, G. A Variant of Bucket Sort: Shell Sort versus Insertion Sort. 10th ICCCNT, IIT, IEEE 45670, 2019
5. Fitro, A. Comparison of Efficiency Data Soring Algorithms based on Execution Time. *International Journal of Scientific Research in Computer Science, Engineer-ing and Information technology*, ISSN: 2456-3307, 15-21, 2023. Doi: 10.32628/CSEIT2390151
6. Alnihoud, J. & Mansi, R. An Enhancement of Major Sorting Algorithms. International Arab Journal of Information Technology, Vol.7, No.1, 2010
7. Wang, Z., Tian, L., Guo, D. & Jiang, X. Optimization and Analysis of Large Scale Data Sorting Algorithm based on Hadoop. 2015, https://arxiv.org/pdf/1506.00449.pdf
8. Markina, M. & Buzdalov, M. Hybridizing Non-dominated Sorting Algorithms: Divided-and-Conquer Meets Best Order Sort. 2018. https://arxiv.org/pdf/1704. 04205.pdf

9. Chen, F., Chen, N., mao, H. & Hu, H. An Efficient Sorting Algorithm – Ultimate Heapsort (UHS). 2019. https://arxiv. org/pdf/1902.00257.pdf
10. Shirazi, S.M. & Bagheri, A. Apply Sorting Algoritms to FAST Problem. 2019. https://arxiv.org/pdf/1910.06920.pdf
11. Olukanmi, P., Popoola, P. & Olusanya, M. Centroid Sort: A Clustering-based Technique for Accelerating Sorting Algorithms. 2nd International Multi-disciplinary Information Technology and Engineering Conference (IMITEC), 2020. Doi: 10.1109/IMITEC50163.2020.9334102
12. Durrani, O.K. & Hayan, S.A. Asymptotic Performances of Popular Programming Languages for Popular Sorting Algorithms. *Semiconductor Optoelectronics*, Vol-41, No.1, 149-169, 2023
13. Roychoudhury, J. & Yadav, J. Efficient Algorithms for Sorting in Trees. 2022. https://arxiv.org/pdf/2205.15912.pdf
14. Hayfron-Acquah, J.B & Riverson, K. Improved Selection Sort Algorithm. International Journal of Computer Applications, 0975-8887, Vol-110, No.5, 2015
15. Juge, V. Adaptive Shivers Sort: An Alternative Sorting Algorithm. 2020. ACM-SIAM Symposium on Discrete Algorithms (SODA). 2020. Doi: 10.1137/1.9781611975994.101
16. Singh, N.K. & Chakraborty, S. Smart Sort: Design and Analysis of a fast, Efficient and Robust Comparison based on Internal Sort Algorithm. 2012. https://arxiv.org/pdf/ 1204.5083.pdf
17. Jha, S.K. Revisiting Calculation of Moments of Number of Comparisons used by the Randomized Quick Sort Algorithm. 2017. *Journal ref: Discrete math, Algorithms and Applications*. 9(1), 1-6
18. Fasha, M. Comparative Analysis for Quick Sort Algorithm under Four Different Modes of Execution. 2021. https://arxiv.org/pdf/2109.01719.pdf
19. Wang, D., Zhang, X., Men, T., Wang, M. & Qin, H. An Implementation of Sorting Algorithm based on Java Multi-thread Technology. International Conference on Computer Science and Electronics Engineering, 2012.
20. Pathak, N. & Tiwari, S. Improved Double Selection Sort using Algoritm. *S-JPSET*, Vol-9, No.2, e-ISSN: 2454-5767. 2017. Doi: 10.18090/samriddhi.v9i02.10866

Appendix

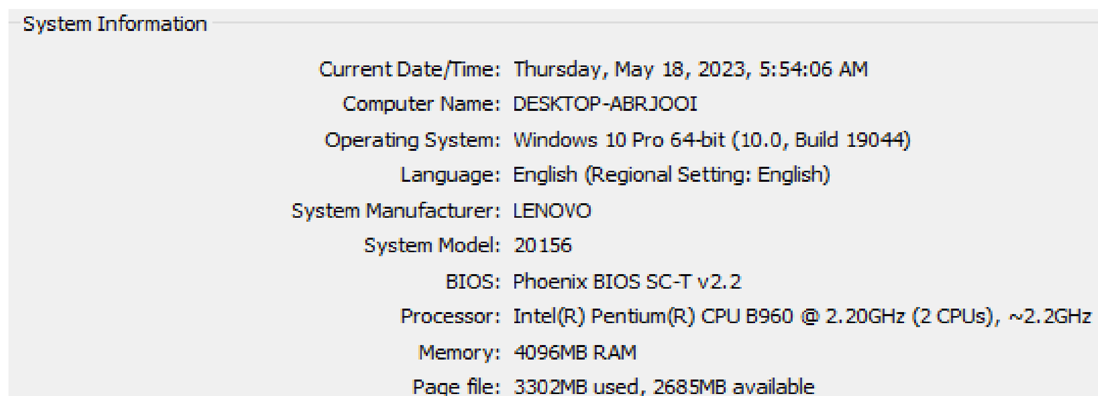

Figure-2. The specification of CPU used in experiment

ISSN 2029-0454
2023

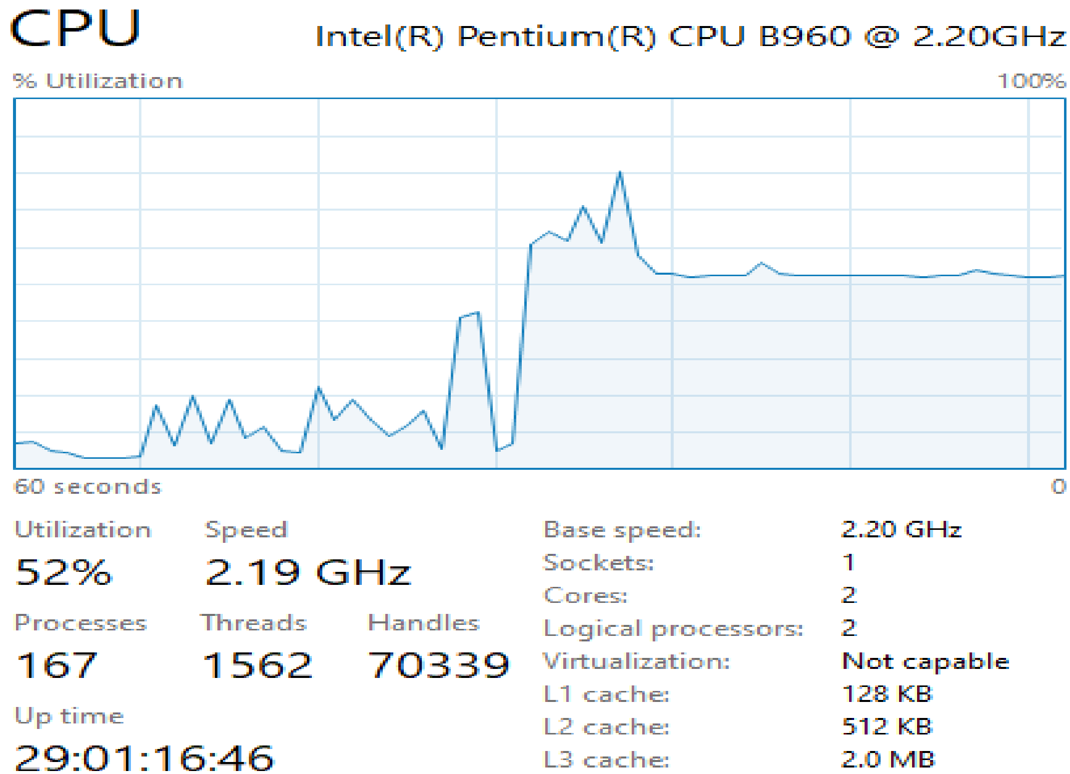

Figure-3. The CPU utilization transition before and during sorting execution at 9 million number of data

COMPARISON OF TIME SORTING: 1000 TO 12000 RANDOM DATA BY 4 CLASSICAL SORTING ALGORITHM (*)

InsSrt(1000):1463 microS === QckSrt(1000):815 microS === ShlSrt(1000):475 microS === SelSrt(1000):358 microS	InsSrt(2000):2893 microS === QckSrt(2000):2278 microS === ShlSrt(2000):752 microS === SelSrt(2000):1409 microS	InsSrt(3000):5435 microS === QckSrt(3000):4513 microS === ShlSrt(3000):973 microS === SelSrt(3000):3429 microS
Sorting 1000 random data	Sorting 2000 random data	Sorting 3000 random data
InsSrt(4000):8533 microS === QckSrt(4000):8289 microS === ShlSrt(4000):1301 microS === SelSrt(4000):5656 microS	InsSrt(5000):12597 microS === QckSrt(5000):12578 microS === ShlSrt(5000):1421 microS === SelSrt(5000):9102 microS	InsSrt(6000):17887 microS === QckSrt(6000):17680 microS === ShlSrt(6000):3566 microS === SelSrt(6000):12869 microS
Sorting 4000 random data	Sorting 5000 random data	Sorting 6000 random data
InsSrt(7000):24134 microS === QckSrt(7000):23762 microS === ShlSrt(7000):2029 microS === SelSrt(7000):18053 microS	InsSrt(8000):31296 microS === QckSrt(8000):34073 microS === ShlSrt(8000):2251 microS === SelSrt(8000):23871 microS	InsSrt(9000):38864 microS === QckSrt(9000):39624 microS === ShlSrt(9000):2612 microS === SelSrt(9000):28309 microS
Sorting 7000 random data	Sorting 8000 random data	Sorting 9000 random data
InsSrt(10000):48405 microS === QckSrt(10000):48696 microS === ShlSrt(10000):3312 microS === SelSrt(10000):35326 microS	InsSrt(11000):58171 microS === QckSrt(11000):56904 microS === ShlSrt(11000):3470 microS === SelSrt(11000):42428 microS	InsSrt(12000):69987 microS === QckSrt(12000):68659 microS === ShlSrt(12000):3886 microS === SelSrt(12000):50529 microS
Sorting 10000 random data	Sorting 11000 random data	Sorting 12000 random data

(*) Insertion, Quick, Shell and Selection sort algorithms (Processor: Intel® Pentium® CPU B960 Duo-core-2.20 GHz)

Figure-4. The performance comparison of four classical sorting algorithm based on running time for various number of input data

Table-1. Performance comparison of shell-sort algorithm in various partition number values based on the number of random data to be sorted with the range of 400 to 3000 (for the optimum partition number see the bold one and for the summary see the last row)

<table>
<tr><td>partition=2->T(400):315
microSpartition=3->T(400):307
microSpartition=4->T(400):35
microSpartition=5->T(400):34
microSpartition=6->T(400):35
microSpartition=7->T(400):32
microSpartition=8->T(400):35
microSpartition=9->T(400):33
microSpartition=10->T(400):33
microSpartition=15->T(400):37
microSpartition=20->T(400):39
microS</td><td>partition=2->T(400):312
microSpartition=3->T(400):314
microSpartition=4->T(400):36
microSpartition=5->T(400):35
microSpartition=6->T(400):34
microSpartition=7->T(400):33
microSpartition=8->T(400):32
microSpartition=9->T(400):34
microSpartition=10->T(400):34
microSpartition=15->T(400):36
microSpartition=20->T(400):39
microS</td><td>partition=2->T(400):563
microSpartition=3->T(400):371
microSpartition=4->T(400):37
microSpartition=5->T(400):120
microSpartition=6->T(400):35
microSpartition=7->T(400):33
microSpartition=8->T(400):34
microSpartition=9->T(400):32
microSpartition=10->T(400):34
microSpartition=15->T(400):39
microSpartition=20->T(400):42
microS</td></tr>
<tr><td>partition=2->T(800):806
microSpartition=3->T(800):787
microSpartition=4->T(800):82
microSpartition=5->T(800):76
microSpartition=6->T(800):75
microSpartition=7->T(800):81
microSpartition=8->T(800):76
microSpartition=9->T(800):84
microSpartition=10->T(800):79
microSpartition=15->T(800):93
microSpartition=20->T(800):94
microS</td><td>partition=2->T(800):809
microSpartition=3->T(800):1431
microSpartition=4->T(800):79
microSpartition=5->T(800):75
microSpartition=6->T(800):78
microSpartition=7->T(800):74
microSpartition=8->T(800):83
microSpartition=9->T(800):74
microSpartition=10->T(800):82
microSpartition=15->T(800):83
microSpartition=20->T(800):106
microS</td><td>partition=2->T(800):802
microSpartition=3->T(800):1133
microSpartition=4->T(800):79
microSpartition=5->T(800):77
microSpartition=6->T(800):79
microSpartition=7->T(800):82
microSpartition=8->T(800):82
microSpartition=9->T(800):73
microSpartition=10->T(800):77
microSpartition=15->T(800):89
microSpartition=20->T(800):101
microS</td></tr>
<tr><td>partition=2->T(1500):1844
microSpartition=3->T(1500):1290
microSpartition=4->T(1500):166
microSpartition=5->T(1500):160
microSpartition=6->T(1500):163
microSpartition=7->T(1500):165
microSpartition=8->T(1500):168
microSpartition=9->T(1500):182
microSpartition=10->T(1500):195
microSpartition=15->T(1500):212
microSpartition=20->T(1500):232 microS</td><td>partition=2->T(1500):1892
microSpartition=3->T(1500):3246
microSpartition=4->T(1500):169
microSpartition=5->T(1500):162
microSpartition=6->T(1500):159
microSpartition=7->T(1500):163
microSpartition=8->T(1500):162
microSpartition=9->T(1500):172
microSpartition=10->T(1500):176
microSpartition=15->T(1500):196
microSpartition=20->T(1500):219 microS</td><td>partition=2->T(1500):1719
microSpartition=3->T(1500):1652
microSpartition=4->T(1500):162
microSpartition=5->T(1500):161
microSpartition=6->T(1500):156
microSpartition=7->T(1500):162
microSpartition=8->T(1500):174
microSpartition=9->T(1500):216
microSpartition=10->T(1500):171
microSpartition=15->T(1500):231
microSpartition=20->T(1500):243
microS</td></tr>
<tr><td>partition=2->T(3000):1846
microSpartition=3->T(3000):1745
microSpartition=4->T(3000):372
microSpartition=5->T(3000):356
microSpartition=6->T(3000):362
microSpartition=7->T(3000):362
microSpartition=8->T(3000):547
microSpartition=9->T(3000):481
microSpartition=10->T(3000):428
microSpartition=15->T(3000):453
microSpartition=20->T(3000):756 microS</td><td>partition=2->T(3000):3323
microSpartition=3->T(3000):8106
microSpartition=4->T(3000):387
microSpartition=5->T(3000):396
microSpartition=6->T(3000):362
microSpartition=7->T(3000):361
microSpartition=8->T(3000):364
microSpartition=9->T(3000):391
microSpartition=10->T(3000):385
microSpartition=15->T(3000):451
microSpartition=20->T(3000):511 microS</td><td>partition=2->T(3000):2032
microSpartition=3->T(3000):1628
microSpartition=4->T(3000):377
microSpartition=5->T(3000):362
microSpartition=6->T(3000):404
microSpartition=7->T(3000):361
microSpartition=8->T(3000):373
microSpartition=9->T(3000):382
microSpartition=10->T(3000):410
microSpartition=15->T(3000):521
microSpartition=20->T(3000):546
microS</td></tr>
<tr><td>Partition=5-> 5Partition=6-> 2Partition=7-> 3Partition=8-> 1Partition=9-> 1</td><td>800,1500(2),3000(2)800,1500 400,800,3000400400</td><td>400 ->7,8,9 -> 32
(@0.08)800 ->5,6,7 -> 74
(@0.0925)1500->5,5,6 -> 159
(@0.106)3000->5,5,7 -> 356
(@0.1187)</td></tr>
</table>

Table-2. Performance comparison of shell-sort algorithm in various partition number values based on the number of random data to be sorted with the range of 6000 to 100.000 (for the optimum partition number see the bold one and for the summary see the last row)

partition=2->T(6000):2366 microS partition=3->T(6000):2433 microS partition=4->T(6000):826 microS partition=5->T(6000):841 microS partition=6->T(6000):817 microS partition=7->T(6000):842 microS partition=8->T(6000):1105 microS partition=9->T(6000):907 microS partition=10->T(6000):981 microS	partition=2->T(6000):2398 microS partition=3->T(6000):5028 microS partition=4->T(6000):925 microS partition=5->T(6000):800 microS partition=6->T(6000):810 microS partition=7->T(6000):881 microS partition=8->T(6000):825 microS partition=9->T(6000):850 microS partition=10->T(6000):895 microS	partition=2->T(6000):2467 microS partition=3->T(6000):2248 microS partition=4->T(6000):830 microS partition=5->T(6000):893 microS partition=6->T(6000):803 microS partition=7->T(6000):853 microS partition=8->T(6000):842 microS partition=9->T(6000):915 microS partition=10->T(6000):915 microS
partition=2->T(10000):3244 microS partition=3->T(10000):2737 microS partition=4->T(10000):1528 microS partition=5->T(10000):1508 microS partition=6->T(10000):1464 microS partition=7->T(10000):1588 microS partition=8->T(10000):1657 microS partition=9->T(10000):1664 microS partition=10->T(10000):1703 micro	partition=2->T(10000):3640 microS partition=3->T(10000):2633 microS partition=4->T(10000):1758 microS partition=5->T(10000):1495 microS partition=6->T(10000):2170 microS partition=7->T(10000):1918 microS partition=8->T(10000):1593 microS partition=9->T(10000):1699 microS partition=10->T(10000):1682 microS	partition=2->T(10000):3506 microS partition=3->T(10000):2873 microS partition=4->T(10000):1895 microS partition=5->T(10000):1579 microS partition=6->T(10000):1482 microS partition=7->T(10000):1649 microS partition=8->T(10000):1557 microS partition=9->T(10000):1541 microS partition=10->T(10000):1702 microS
partition=2->T(25000):7302 microS partition=3->T(25000):6452 microS partition=4->T(25000):4452 microS partition=5->T(25000):6906 microS partition=6->T(25000):4786 microS partition=7->T(25000):4510 microS partition=8->T(25000):11334 micro partition=9->T(25000):4820 microS partition=10->T(25000):6229 microS	partition=2->T(25000):6973 microS partition=3->T(25000):7949 microS partition=4->T(25000):4319 microS partition=5->T(25000):4706 microS partition=6->T(25000):4385 microS partition=7->T(25000):4682 microS partition=8->T(25000):4960 microS partition=9->T(25000):4781 microS partition=10->T(25000):5179 microS	partition=2->T(25000):7279 microS partition=3->T(25000):5796 microS partition=4->T(25000):4626 microS partition=5->T(25000):4650 microS partition=6->T(25000):4949 microS partition=7->T(25000):4699 microS partition=8->T(25000):4726 microS partition=9->T(25000):4834 microS partition=10->T(25000):5255 microS

partition=2->T(50000):33706 microS partition=3->T(50000):13120 microS partition=4->T(50000):12610 microS partition=5->T(50000):11697 microS partition=6->T(50000):10330 micro partition=7->T(50000):10522 microS partition=8->T(50000):11047 microS partition=9->T(50000):12691 microS partition=10->T(50000):12715 micro	partition=2->T(50000):19468 microS partition=3->T(50000):12375 microS partition=4->T(50000):10132 microS partition=5->T(50000):10279 microS partition=6->T(50000):10079 microS partition=7->T(50000):11269 microS partition=8->T(50000):12968 microS partition=9->T(50000):15706 microS partition=10->T(50000):11621 micro	partition=2->T(50000):17371 microS partition=3->T(50000):11541 microS partition=4->T(50000):10165 microS partition=5->T(50000):10761 microS partition=6->T(50000):10608 microS partition=7->T(50000):16465 microS partition=8->T(50000):10764 microS partition=9->T(50000):14668 microS partition=10->T(50000):11539 microS
partition=2->T(100000):31831 micro partition=3->T(100000):25617 micro partition=4->T(100000):23586 micro partition=5->T(100000):26652 micro partition=6->T(100000):25306 micro partition=7->T(100000):23668 micro partition=8->T(100000):24052 micro partition=9->T(100000):25995 micro partition=10->T(100000):26684 mico	partition=2->T(100000):39662 micro partition=3->T(100000):26882 micro partition=4->T(100000):38214 micro partition=5->T(100000):45000 micro partition=6->T(100000):38650 micro partition=7->T(100000):38837 micro partition=8->T(100000):44122 micro partition=9->T(100000):34648 micro partition=10->T(100000):28139 micr	partition=2->T(100000):36582 microS partition=3->T(100000):45338 microS partition=4->T(100000):22962 microS partition=5->T(100000):30798 microS partition=6->T(100000):25141 microS partition=7->T(100000):25559 microS partition=8->T(100000):29634 microS partition=9->T(100000):24684 microS partition=10->T(100000):26266 micro
Partition=3 ->1 Partition=4 ->6 Partition=5 ->3 Partition=6 ->5	100000 25000(3),50000(1),100000(2) 6000,10000 6000(2),10000(2),50000(2)	6000 ->5,6,6 -> 800 (@0.1333) 10000 ->5,6,6 -> 1464 (@0.1464) 25000 ->4,4,4 -> 4319 (@0.1728) 50000 ->4,5,6 -> 10079 (@0,2016) 100000->3,4,4-> 22962 (@0.2296)

Table-3. Performance comparison of shell-sort algorithm in various partition number values based on the number of random data to be sorted with the range of 1 to 15 million (for the optimum see the bold one and for the summary see the last row)

partition=2->T(1000000):371 milliS partition=3->T(1000000):343 milliS partition=4->T(1000000):303 milliS partition=5->T(1000000):295 milli partition=6->T(1000000):319 milliS partition=7->T(1000000):312 milliS partition=8->T(1000000):352 milliS partition=9->T(1000000):388 milliS partition=10->T(1000000):373 milliS	partition=2->T(2000000):896 milliS partition=3->T(2000000):763 milliS partition=4->T(2000000):706 milliS partition=5->T(2000000):695 milli partition=6->T(2000000):737 milliS partition=7->T(2000000):791 milliS partition=8->T(2000000):834 milliS partition=9->T(2000000):877 milliS partition=10->T(2000000):884 milliS	partition=2->T(3000000):1426 milliS partition=3->T(3000000):1220 milliS partition=4->T(3000000):1103 milli partition=5->T(3000000):1138 milliS partition=6->T(3000000):1175 milliS partition=7->T(3000000):1248 milliS partition=8->T(3000000):1369 milliS partition=9->T(3000000):1432 milliS partition=10->T(3000000):1644 milli

partition=2->T(4000000):2062 milliS partition=3->T(4000000):1717 milliS partition=4->T(4000000):1634 milliS partition=5->T(4000000):1593 milli partition=6->T(4000000):1737 milliS partition=7->T(4000000):1901 milliS partition=8->T(4000000):1860 milliS partition=9->T(4000000):2048 milliS partition=10->T(4000000):2164 mill	partition=2->T(5000000):2376 milliS partition=3->T(5000000):1909 milli partition=4->T(5000000):2052 milliS partition=5->T(5000000):2047 milliS partition=6->T(5000000):2313 milliS partition=7->T(5000000):2334 milliS partition=8->T(5000000):2666 milliS partition=9->T(5000000):2698 milliS partition=10->T(5000000):2813 milli	partition=2->T(6000000):2890 milliS partition=3->T(6000000):2425 milli partition=4->T(6000000):2597 milliS partition=5->T(6000000):2724 milliS partition=6->T(6000000):2784 milliS partition=7->T(6000000):2931 milliS partition=8->T(6000000):3206 milliS partition=9->T(6000000):3734 milliS partition=10->T(6000000):3525 milli
partition=2->T(7000000):3970 milliS partition=3->T(7000000):3223 milliS partition=4->T(7000000):3069 milli partition=5->T(7000000):3196 milliS partition=6->T(7000000):3361 milliS partition=7->T(7000000):3620 milliS partition=8->T(7000000):4352 milliS partition=9->T(7000000):4533 milliS partition=10->T(7000000):4567 mill	partition=2->T(8000000):4687 milliS partition=3->T(8000000):4014 milli partition=4->T(8000000):4116 milliS partition=5->T(8000000):3827 milliS partition=6->T(8000000):4004 milliS partition=7->T(8000000):4153 milliS partition=8->T(8000000):5136 milliS partition=9->T(8000000):4977 milliS partition=10->T(8000000):5673 milli	partition=2->T(9000000):5756 milliS partition=3->T(9000000):3940 milli partition=4->T(9000000):4182 milliS partition=5->T(9000000):4024 milliS partition=6->T(9000000):4577 milliS partition=7->T(9000000):4700 milliS partition=8->T(9000000):5702 milliS partition=9->T(9000000):5237 milliS partition=10->T(9000000):6279 milli
partition=2->T(10000000):6312 mill partition=3->T(10000000):4724 mil partition=4->T(10000000):5121 mill partition=5->T(10000000):4853 mill partition=6->T(10000000):5430 mill partition=7->T(10000000):5895 mill partition=8->T(10000000):6364 mill partition=9->T(10000000):6530 mill partition=10->T(10000000):7729 mi	partition=2->T(11000000):7242 milli partition=3->T(11000000):6055 milli partition=4->T(11000000):5306 mil partition=5->T(11000000):5769 milli partition=6->T(11000000):5841 milli partition=7->T(11000000):6287 milli partition=8->T(11000000):8120 milli partition=9->T(11000000):7439 milli partition=10->T(11000000):7978 mil	partition=2->T(12000000):8091 milli partition=3->T(12000000):6358 milli partition=4->T(12000000):6080 milli partition=5->T(12000000):6043 mill partition=6->T(12000000):6726 milli partition=7->T(12000000):7133 milli partition=8->T(12000000):8513 milli partition=9->T(12000000):9013 milli partition=10->T(12000000):8794 mil

partition=2->T(13000000):7575 mill partition=3->T(13000000):7211 mill partition=4->T(13000000):6533 mil partition=5->T(13000000):6600 mill partition=6->T(13000000):7000 mill partition=7->T(13000000):8109 mill partition=8->T(13000000):9478 mill partition=9->T(13000000):8717 mill partition=10->T(13000000):9791 m	partition=2->T(14000000):8212 milli partition=3->T(14000000):6726 mil partition=4->T(14000000):7238 milli partition=5->T(14000000):8355 milli partition=6->T(14000000):7926 milli partition=7->T(14000000):8709 milli partition=8->T(14000000):9108 milli partition=9->T(14000000):10552 mil partition=10>T(14000000):10964 mi	partition=2->T(15000000):9381 milli partition=3->T(15000000):8332 milli partition=4->T(15000000):8067 milli partition=5->T(15000000):7966 mill partition=6->T(15000000):8405 milli partition=7->T(15000000):8988 milli partition=8->T(15000000):11952 mil partition=9->T(15000000):11021 mil partition=10>T(15000000):11530 mi
Partition=3->5 Partition=4->4 Partition=5->6	5M,6M,9M,10M,14M 3M,7M,11M,13M 1M,2M,4M,8M,12M,15M	1M->0.295S(@0.29); 8M->3.83S(@0.478) 2M->0.60S(@0.298); 9M->3.94S(@0.438) 3M->1.10S(@0.37); 10M->4.72S(@0.472) 4M->1.593S(@0.4); 11M->5.30S(@0.487) 5M->1.91S(@0.382); 12M->6.04S(@0.50) 6M->2.43S(@0.404); 13M->6.53S(@0.50) 7M->3.07S(@0.438); 14M->6.73S(@0.48) 15M->7.97S(@0.53)