DOI: http://dx.doi.org/10.26483/ijarcs.v10i3.6433

ISSN No. 0976-5697

Volume 10, No. 3, May-June 2019

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

ANALYSIS OF SHELLSORT ALGORITHMS

Avik Mitra
BCA Department
The Heritage Academy
Kolkata, India

Jash Kothari
Final Year Student, BCA
The Heritage Academy
Kolkata, India

Annesa Ganguly
Final year student, BCA
The Heritage Academy
Kolkata, India

Abstract: Shellsort is a comparison sort that uses insertion sort at each iteration to make a list of interleaved elements nearly sorted so that at the last iteration the list is almost sorted. The time complexity of Shellsort is dependent upon the method of interleaving (called increment sequence) giving variants of Shellsort. However, the problem of finding proper of interleaving to achieve the minimum time complexity of O(n log n) is still open. In this paper, we have analyzed the performance of variants of Shellsort based on their time complexity. Our measure of time complexity is independent of the machine configuration and considers all the operations of a sorting process. We found that the interleaving method or increment sequence proposed by Sedgewick performs best among the analyzed variants.

Keywords: Shellsort; increment sequence; variants; survey; time complexity; algorithm; data structure

I. INTRODUCTION

Shellsort [1] is an in-situ comparison sort algorithm where at each iteration, each list of interleaved elements from the list *A[0, ..., n]*, are sorted by insertion sort; each list of interleaved elements forms disjoint sets of elements. The interleaving is reduced in subsequent iteration, until it becomes 1, in which case insertion sort gets applied to the whole, now nearly sorted, list *A*. The sorting algorithm is oblivion to the data [2] and one of its implementation is (sorting in non-decreasing order):

```
Void Shellsort (A[0, ..., n-1], SkipLength[0, ...., k-1])
//SkipLength[x] > SkipLength[x+1]

//SkipLength[k-1] = 1, SkipLength[i] = h_{i+1}

{

  For i = 0 to (k-1), step by (+1), do: //For each skip length

  {

    For j = 0, j < SkipLength[i], step by (+1), do:

    {

      Temp = A[j]. //Assignment operation – A[j] is assigned to Temp.

      For z = j – SkipLength[i], z ≥ 0 and A[z] > Temp, step by (-SkipLength[i]), do:

      {

        A[z + Skiplength[i] ] = A[z]

      } //End of For-loop

      A[ z + SkipLength[i] ] = Temp

    }//End of For-loop.

  }//End of For-loop

}//End of Shellsort
```

The sequence of interleaving $(h_1, h_2, ..., h_k)$ with $h_1 > h_2 > ... > h_k = 1$, is called *increment or offset sequence* and each h_i is called *skip length*. For [1], the increment sequence is $(\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, 1)$. There are many variants of the original Shellsort algorithm depending upon the increment sequence [2]-[15]; the time complexity of the algorithm is also dependent upon the increment sequence. Finding optimal increment sequence that will minimize the time complexity of the Shellsort is still an open problem [12]. In addition to this, data-oblivion property of the sorting algorithm makes it an attractive solution for deployment in those systems where a dataset, distributed over multiple nodes in a network, needs to be arranged in certain order. In this paper we will compare the Shellsort variants in terms of the time complexities. Unlike [16]-[18] that compared the shellsort variants based on a parameter that includes number of swaps or number of comparisons, we have defined a parameter that have included these factors and also the time consumed in checking conditions for loop to run. We believe that our parameter for comparison is closer to the general definition of theoretical time complexity of an algorithm. The rest of this paper is organized as follows: section II will make a brief description of the variants of Shellsort. The framework used to make the comparison among the Shellsort variants and findings are discussed in section III. We conclude our paper in section IV, followed by references.

II. SURVEY OF SHELLSORT VARIANTS

At i-th iteration of Shellsort, list *A* gets subdivided into h_i sublists each of size $\lfloor n/h_i \rfloor$ [12], and insertion sort is applied to each of these lists. So, the time complexity of the sorting algorithm depends upon the time complexity of sorting each of the sublists. Moreover, since each of these sublists gets sorted resulting partially sorted list, therefore, at subsequent iterations it is expected that there will be less swaps than the number of comparisons. The sequence proposed by Shell [1] uses $\lfloor \log n \rfloor$ length sequence. The time complexity in worst case is proved to be $O(n^2)$ when n is a power of 2. To reduce the time complexity, [3] proposed that even skip length should be replaced by next odd number, resulting time complexity of $O(n^{3/2})$. [4] also achieved the same worst-case complexity using the increment sequence $\left(\frac{n}{2}+1,\ldots \frac{n}{2^{\lfloor \log n \rfloor}}+1,1\right)$. [5] obtained a tighter bound of $\Theta\left(n^{\frac{3}{2}}\right)$ using reverse of the following generated sequence (note that mod represents modular operation):

For i =1 to $2^{\lfloor \log(n-1) \rfloor}$; i = i*2 //Double value of i at each iteration

{

J = i

Do{ Store J // Note J

J = (3*J)/2 //Integer division

} while (J mod 3 == 0 and J < n) //End of Do-while loop

}//End of For-loop

The problem of moving an element to its rightful place is reduced to Frobenius problem [19] by [6] where it is derived that at each iteration, the skip length should not be a linear combination of the skip length of the next iterations, thus avoiding unnecessary relocation of same element in the list; this resulted better time complexity of $O(n^{5/4})$. Time complexity of 3-tuple increment sequence (h, k, 1) is studied in [7] where the exact time complexities for each of the three iterations are derived. [8] proposed the increment sequences – $h_k = 1$ *and* $h_i = 3h_{i+1} + 1$ *where* h_1 *is such that* $3h_0 \geq n$, and *(2, 1),* whereas, [9] proposed the reverse of following increment sequence $(3h_i \geq n)$ using the Frobenius problem:

$$h_i = \begin{cases} 9.2^i - 9.2^{\frac{i}{2}} + 1 \; if \; s \; is \; even \\ 8.2^i - 6.2^{\frac{i+1}{2}} + 1 \; if \; s \; is \; odd \end{cases}$$

Using this increment sequence, $O(n^{4/3})$ time complexity is achieved. 3-tuple increment sequence for Shellsort are again explored in [10] where the increment sequence $(n^{7/15}, n^{1/5}, 1)$ is proposed, getting a time complexity of $O(n^{23/15})$. [11] proposed increment sequence where $h_i = 2^i - 1$ $(i \geq 1)$ until $h_i \geq n$; the achieved time complexity is almost same as in [3]. [12] analyzed three increment sequences: (1) $(n^{1/3}, 1)$ with time complexity $\Omega(n^{5/3})$ (2) $(n^{1/2}, n^{1/4}, 1)$ with time complexity $O(n^{3/2})$ (3) $(n^{11/16}, n^{7/16}, n^{3/16}, 1)$ with time complexity $\Omega(n^{21/16})$. [13] with increment sequence $\left(\lceil n^{\frac{1}{3}}+1 \rceil, \lceil n^{\frac{1}{3}} \rceil, 1\right)$ obtained time complexity of $O(n^{5/3})$. Genetic algorithm used in [14] to generate two increment sequences based on the size of the list A: 7-tuple increment sequence is used when size of A is between 1000 and 1 million; 15-tuple increment sequence is used when size of A exceeds 1 million. [15] uses binary search algorithm for each skip length so that the time complexity of the algorithm becomes O(n log n); the algorithm implicitly uses reverse of the increment sequence $h_i = \lceil \left(9(9/4)^i - 4\right)/5 \rceil \; such \; that \; \frac{9h_i}{4} < n \wedge i \geq 0.$ The discussed variants did not achieve the lower bound $\Theta(n \log n)$ [20] of a comparison sort algorithm. During writing of this paper this bound is probabilistically achieved by [2] and [17] proved that to achieve the lower bound the length of the increment sequence will be Θ¿, which is yet to be found.

Most of the time complexities of the Shellsort variants have considered either number of swaps or number of comparisons, except [14] and [21] which have additionally used a specialized machine for actual time taken. However, during run of algorithm associated variables like use of counter variables and temporary variables, using instruction for increment or decrement etc., adds up the time complexity and their contribution to the time complexity is proportional to the number of times a loop, using these, runs. In the next section we define a parameter measuring of time complexity where we include these factors and using this parameter we make compare the performances of the Shellsort variants.

III. COMPARATIVE ANALYSIS OF SHELLSORT VARIANTS

We first define the parameter for the comparison followed by the methodology for comparison and results obtained.

A. *Defining Complexity*

Time complexity of an algorithm B, *C(B)*, consisting (I_1, ... I_t) instructions running (n_1, ... n_t) times respectively can be defined as:

$$C(B) = \sum_{j=1}^{t} (I¿¿j \times n_j)¿ \quad \text{.. (1)}$$

We can, therefore, measure *C(B)* by using a global variable which is initially set to 0; the variable gets incremented by one for each execution of the instructions.

The definition (1) includes the number of comparisons, number of exchanges and the number of times associated variables are used. The definition is also independent of underlying platform used for implementation of the algorithm B.

B. *Methodology for Comparison*

We compare average case complexities of the Shellsort variants. We have selected the variants [1][3]-[5][7]-[13] and [15]. We have not taken [2] as it is a probabilistic algorithm where sorting is not guaranteed (though the probability of getting a sorted list is very high). The increment sequence

generation of [6] is complex as it requires checking co-prime of two numbers, thus it can take more time to generate the increment sequence than actual sorting if the size of list to be sorted is high. Therefore we have not considered the variant in our analysis. Based on similar reason (time complexity of genetic algorithms are generally high), [14] is also not considered.

We have implemented the selected Shellsort variants in Java programming language as generating random list of elements is easier in it. Each of the variants is implemented in a separate class. Objects are created for each of these variants in the main method and executed. During implementation we have not considered the time complexity of increment sequence generation as we are more inclined towards the time complexity of sorting (section I). To find time complexity of each Shellsort variant and size *n* of the list to be sorted, 1000 random lists are generated, the average of these 1000 runs are taken. *n* is varied from 500 to 10000 with interval of 500, that is, 20 values of *n* is taken. To ensure fairness of analysis among the selected variants, same 1000 lists for given *n* is fed into all the variants. Therefore, our analysis has used same 20 × 1000 = 20000 lists, for all the variants.

C. *Results*

We have plotted the obtained average of average-case time complexity in vertical axis and size of dataset in horizontal axis. The mapping of the labels used in plots is given in table I. The variation of time complexity with lists' size is shown in figure 1. To avoid overflow during measuring the time complexity of a variant, the variable used for the measure is incremented by 0.010 for a run of an observed instruction. So, the time complexities in figure are scaled down by 100.

From figure 1, it can be observed that the second increment sequence in [8] performed worst as it has used only two skip-lengths and hence it nearly reduced to insertion sort which has time complexity $O(n^2)$. [9] performed best among the selected variants since: (1) the possibility that two consecutive skip length in increment sequences are co-primes is high, thereby avoiding unnecessary movement of an element of list; (2) the number of skip lengths for given *n* is close to *log n* as suggested by [17]. On similar reason, [1] and [15] (though [15] performed better than [1]) performed nearly as good as [9].

Table I. Labelling of Increment Sequences for Plot

Increment Sequence	***Label***
[1]	DLShell
[4]	PapernovStasevich
[3]	LazarusFrank
[5]	Pratt
[7]	ACCYao
[8]	Knuth1, Knuth2
[9]	Sedgewick
[10]	JansonKnuth
[11]	Hibbard
[12]	Vitnayi1, Vitnayi2, Vitnayi3
[13]	Weiss
[15]	Tokuda

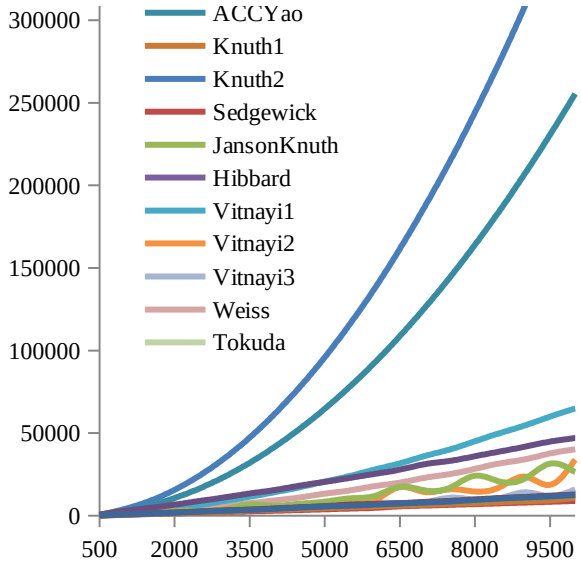

Figure 1. Comparison of Shellsort variants

IV. CONCLUSION

We have compared average case time complexity of variants of Shellsort algorithm and observed that the variants that uses increment sequences of length close to 1 tends to perform worse than those variants whose length of increment sequences is proportional or close to *log n*. Based on this fact increment sequence proposed by Sedgewick performed best among the other variants that we have considered. Therefore, we suggest use of Sedgewick's increment sequence for use in practical field.

V. REFERENCES

[1] D.L.Shell, "A High Speed Sorting Procedure", Communications of the ACM, volume 2, issue 7, pp 30-32, July 1959. DOI: 10.1145/368370.368387.

[2] Michael T. Goodrich, "Randomized Shellsort: A Simple Data-Oblivious Sorting Algorithm", Journal of the ACM, volume 58, issue 6, 2001.

[3] R.M.Frank, R.B. Lazarus, "A High Speed Sorting Procedure", Communications of the ACM, volume 3, issue 1, pp 20-22, January 1960.

[4] A.A.Papernov and G.Stasevich, "A Method of Information Sorting in Computer Memories", Problems in Information Transmission, volume 1, issue 3, pp 63-75, 1965.

[5] V.R.Pratt, "Shellsort and Sorting Networks", No. Stan-CS-72-260, Standford University CA Department of Computer Science, February 1972. Available from: https://apps.dtic.mil/dtic/tr/fulltext/u2/740110.pdf

[6] Janet Incerpi and Robert Sedgewick, " Improved Upper Bounds on Shellsort", Journal of Computer and System Sciences, volume 31, issue 2, pp 210-224, October 1985.

[7] Andrew Chi-Chih Yao, "An Analysis of (h, k, 1) – Shellsort", Journal of Algorithms, volume 1, issue 1, pp 14-50, March 1980.

[8] Donald. E. Knuth, "Art of Computer Programming: volume 3 Sorting and Searching", Second Edition, Addison-Wesley, 1998.

[9] Robert Sedgewick, "A New Upper Bound for Shellsort", Journal of Algorithms, volume 7, issue 2, pp 159-173, June 1986.

[10] Svante Janson and Donald E. Knuth, "Shellsort with Three Increments", Random Structures and Algorithms, volume 10, issue 1, pp 125-142, January 1997.

[11] Thomas N. Hibbard, "An Empirical Study of Minimal Storage Sorting", Communications of the ACM, volume 6, issue 5, pp 206-213, 1963.

[12] Paul Vitnayi, "On the Average-case Complexity of Shellsort", Random Structures and Algorithms, volume 52, issue 2, pp 354-363, 2018.

[13] M.A. Weiss, "Shellsort with Constant Number of Increments", Algorithmica, volume 16, issue 6, pp 649-654, December 1996.

[14] Richard Simpson, Shashidhar Yachavaram, "Faster Shellsort Sequences: A Genetic Algorithm Application", Computers and Their Applications, 1999.

[15] Naoyuki. Tokuda, "An Improved Shellsort", IFIP 12th World Computer Congress on Algorithms, Software, Architecture – Information Processing, 1992.

[22]

[16] Bronislava Brejova, "Analyzing Variants of Shellsort", Information Processing Letters, volume 79, issue 5, pp 223-227, September 2001.

[17] Tao Jing, Ming Li, Paul Vitanyi, "Average-case Complexity of Shellsort", International Colloquium on Automata, Languages and Programming, 1999.

[18] Janet Incerpi and Robert Sedgewick, "Practical Variations of Shellsort", Doctoral Dissertation, INRIA, 1986.

[19] J.L.Ramirez-Alfonsin, "Complexity of the Frobenius Problem", Combinatorica, volume 16, issue 1, pp 143-147, March 1996.

[20] Thomas H. Cormen, Charles E Leiserson, Ronald R. Rivest, Clifford Stein, "Introduction to Algorithms", 3rd Edition, MIT Press and Prentice Hall of India, February 2010.

[21] D. Ghoshdastidar and Mohit Kumar Roy, "A Study on the Evaluation of Shell's sorting Technique", The Computer Journal, volume 18, issue 3, pp 234-235, 1975.