Lietuvos matematikos rinkinys ISSN 0132-2818
Proc. of the Lithuanian Mathematical Society, Ser. A www.mii.lt/LMR/
Vol. **54**, 2013, 61–66

# An empirical study of the gap sequences for Shell sort

**Irmantas Radavičius, Mykolas Baranauskas**

*Department of Mathematics and Informatics, Vilnius University*
Naugarduko 24, LT-03225 Vilnius
E-mail: irmantas.radavicius@mif.vu.lt; mykolas.baranauskas@mif.vu.lt

**Abstract.** We present an improved version of the Shell sort algorithm. Using the algorithm, we study various geometrical sequences and the performance of Shell sort. We demonstrate that neither number of assignments nor the number of comparisons is sufficient to properly evaluate Shell sort and pick the optimal gap sequence. We argue that one should count both operations, as well as measure actual running times.

**Keywords:** sorting algorithm, Shell sort, gap sequence.

## Introduction

The sorting problem is one of the most fundamental problems in computer science: given a sequence $S = (s_1, s_2, \ldots, s_N)$ of comparable data elements, produce a permutation such that the initial data elements are arranged in a non-decreasing (or non-increasing) order. While this problem has been studied in depth over the last several decades, new sorting algorithms and various improvements are still constantly being developed. There are many types of sorting algorithms: comparison-based (use comparison operators to determine the order and do not rely on any particular specifics of the data), stable (preserve the initial relative order of equal elements), in-place (use only $O(1)$ additional storage space) algorithms, etc. There is no single best algorithm – the choice of the algorithm depends on many factors.

*Shell sort* [8] is a comparison-based in-place sorting algorithm. Instead of comparing adjacent elements (as *insertion sort* does), Shell sort makes the elements move to their final position quicker by comparing the more distant elements. The algorithm splits (multiple times) the sequence $S$ into $h$ subsequences $S_1, S_2, \ldots, S_h$, where the subsequence $S_i = (s_i, s_{i+h}, s_{i+2h}, \ldots)$ consists of the data elements that are $h$ positions apart. Every such subsequence is sorted separately (hence, sequence $S$ is made $h$-sorted) using insertion sort (which is known to work better with partially sorted data). Since $h$-sorting a $k$-sorted sequence still leaves it $k$-sorted [5], a *gap sequence* $H = (h_1, h_2, \ldots, h_{|H|})$ is used to gradually increase the sortedness of $S$. During the algorithm, $S$ is $h_1$-sorted, $h_2$-sorted, and so on. Setting $h_{|H|} = 1$ makes the last pass equivalent to simple insertion sort thus ensuring that $S$ will be properly sorted.

The running time of Shell sort depends on the gap sequence used (see, for example, the study [9] by Weiss). In the worst-case, optimal comparison-based sorting algorithms make $\Theta(N \log N)$ comparisons. The lower bound proven for the worst-case of Shell sort is $\Omega(N \log^2 N/(\log \log N)^2)$ making the Shell sort suboptimal [4]. Mean-

while, the study of the average case for Shell sort has been a long-standing problem. For any gap sequence $H$, Jiang et al. [2] showed a lower bound of $\Omega(|H|N^{1+1/|H|})$. It follows that to achieve the optimal time of $O(N \log N)$ for the average-case, some gap sequence of length $\Theta(\log N)$ should be used. However, to the best of our knowledge, such optimal sequence is yet to be provided, and the asymptotically best sequence known is still the one given by Pratt [5] consisting of all the numbers of the form $2^p 3^q, p, q \geqslant 0$ put in the descending order. This sequence has length $\Theta(\log^2 N)$ and gives the worst-case running time of $\Omega(N \log^2 N)$. Due to large number of increments, Pratt's sequence does not perform well in practice. Many other sequences have been suggested by various authors: Shell [8], Knuth [3], Sedgewick [6], Ciura [1] and others. The best sequences actually used are often found experimentally.

In this paper we address the problem of choosing the gap sequence for Shell sort. Our contributions are:

- we present an improved version of the Shell sort algorithm (which performs fewer assignment operations);
- we discuss the problem of the proper evaluation of the performance of Shell sort;
- we perform experiments to study the family of the geometrical sequences and report the results.

## 1 The Shell sort algorithm

Shell sort is considered as one of the fundamental sorting algorithms, and generally taught as a part of the basic computer science curriculum. For the *textbook version* (TSS) of the algorithm (f.e. [7]), see *Algorithm 1.* The algorithm uses a gap sequence $H$ to sort an array $S$ of size $N$. Considering $gap = 1$, lines 2–8 can be recognized as steps of the insertion sort algorithm. The inner *while* loop (lines 4–6) shifts the data elements to create space for the current element $S[i]$ which is inserted into the proper position at line 7. The outer loop (line 1) advances the algorithm along the gap sequence, and finishes with data fully sorted, in ascending order.

While the textbook version can be preferred for the sake of simplicity and other teaching purposes, it is inefficient. It can happen quite often that the current element $S[i]$ is already in the proper position. If the second condition at line 4 evaluates as *false*, the inner loop is not executed. In this case, two assignments (lines 3 and 7) are unnecessary. The proper implementation of the algorithm should make assignments only if at least one swap is required.

```
1 foreach gap in H do
2 |   for i ← gap + 1 to N do
3 |   |   j ← i; temp ← S[i];
4 |   |   while j > gap and S[j − gap] > S[j] do
5 |   |   |   S[j] ← S[j − gap]; j ← j − gap;
6 |   |   end
7 |   |   S[j] ← temp;
8 |   end
9 end
```

**Algorithm 1.** The textbook version of the Shell sort algorithm.

```
1  foreach gap in H do
2  |  for i ← gap + 1 to N do
3  |  |  if S[i − gap] > S[i] then
4  |  |  |  j ← i; temp ← S[i];
5  |  |  |  repeat
6  |  |  |  |  S[j] ← S[j − gap]; j ← j − gap;
7  |  |  |  until j ≤ gap or S[j − gap] ≤ S[j];
8  |  |  |  S[j] ← temp;
9  |  |  end
10 |  end
11 end
```

**Algorithm 2.** The improved version of the Shell sort algorithm.

We present the *improved version* of Shell sort (ISS) as *Algorithm 2*. At line 3, we first check whether $S[i]$ is already in place. If yes, we simply proceed to the next element. Otherwise, the condition at line 3 evaluates to *true*, and since $j \geqslant gap + 1$ after assignment at line 4, we can switch the loop type from *while* to *repeat*, to avoid checking the same condition twice. The improved version results in slightly larger code, but makes the same number of comparisons, and on average makes significantly fewer assignments which results in better running times of Shell sort.

We do not claim to be the first to have suggested this modification. However, we have not seen it published before, and the widespread use of the textbook version motivates us to do so.

## 2 Gap sequences

Picking a good gap sequence is essential for the Shell sort algorithm to perform well. There are many factors that contribute to this. The size of the gap sequence should be $\Theta(\log N)$. The sequence is usually constructed as decreasing, so that the data elements that are farther apart would be compared first. The particular values of the gaps sizes are also important to decrease the number of the redundant comparisons. To accomplish this, good gap sequences contain gap sizes that are relatively prime to each other. The ratio between the gap sizes matters, too. Too small ratio may result in many redundant comparisons, whereas too large ratio may cause the algorithm to be slow with smaller gaps due to lack of partial sorting. A lot of effort has been put in finding better sequences, to improve running times.

Evaluating the performance of the Shell sort is not an obvious task. Many authors tend to measure only element assignments and hence pick the suboptimal sequences that perform poorly in practice. As pointed out in [1], for the Shell sort the number of comparisons is more appropriate. However, both presented versions of the Shell sort algorithm (see Section 1) make the same number of comparisons, but have different running times, due to different number of assignments. Because of this, their optimal gap sequences are also different (see Section 3). Hence we argue that the number of comparisons by its own to properly evaluate the Shell sort is not sufficient either.

To study the Shell sort, most authors work with permutations alone and perform experiments by sorting integers, assuming that the costs of using comparison and assignment operators are similar. Yet in practice the picture is much more complex.

For example, when sorting *text strings*, the complexity of the comparison operation might be proportional to the length of the strings, whereas assignment could be implemented by simple assignment of indices or pointers, without the need to move whole string. Similarly, when sorting *records* by some simple key, the comparison operation might be cheap, while the assignment might as well involve moving all the contents of the possibly large record. Due to differences in costs of the operations, different gap sequences could be applicable, too. Therefore, to properly evaluate the Shell sort and the gap sequences, we suggest to measure *all* the involved operations as well as *actual* running times that are platform specific, but nevertheless, are the actual goal of the optimization.

## 3 Results

In this section we present the results of the empirical study of the geometrical gap sequences, i.e. sequences such that $h_{i-1} = qh_i$, where $q > 1$ is a *common ratio* for the sequence, $h_i < N$ and $h_{|H|} = 1$. We performed the experiments on a machine with 2 GHz Intel(R) Core(TM)2 Duo CPU and 3 GB RAM, running Windows Vista. All programs were implemented in C++ using GNU g++ compiler. To obtain the results, we take averages from testing 100-1000 randomly generated arrays of integers independently taking random values from from 0 to INT_MAX, resulting in few to none duplicate values.

Figure 1 shows operation counts while comparing the textbook version of Shell sort (TSS) with the improved version (ISS). Here the array size is fixed at 1000, and the common ratio $q$ ranges from 1.05 to 1.95. While TSS makes more, ISS makes fewer assignments than comparisons, and for ISS the ratio between assignments and comparisons quickly approaches 1 as $q$ increases. The improvement by ISS is significant, resulting in 80–40 percent fewer assignments as compared to TSS. This also results in around 20% better running times. It is clear that the actual running time is dependent on *both* comparison and assignment operations, and neither should be excluded from the analysis.

Figure 2 shows the results for various geometrical sequences with $q$ ranging from 1.35 to 10, for array sizes 2000 (red), 1500 (orange), 1000 (green) and 500 (blue). We count the comparisons and assignments for both TSS and ISS. Note the differences: in general, optimal $q$ based on comparisons is smaller than optimal $q$ based on assignments using TSS, and yet larger when compared to $q$ based on assignments with ISS, where small $q$ is optimal. In other words, for the choice of the gap sequence,

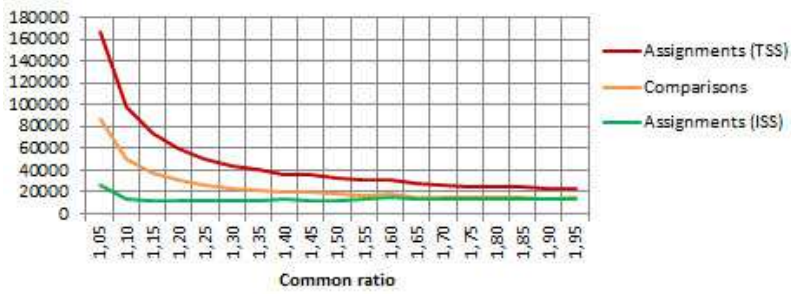
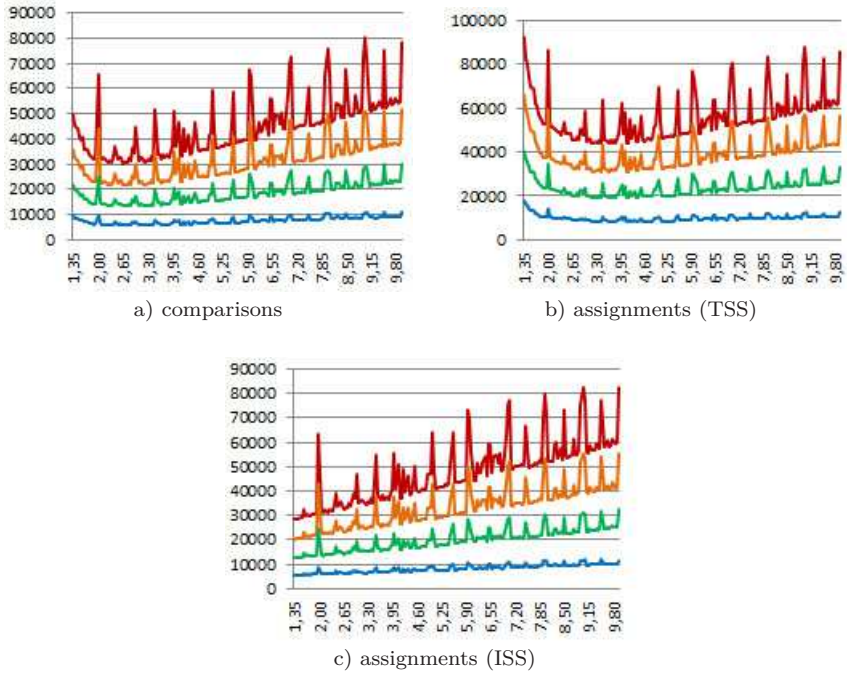



**Figure 1.** Comparing TSS and ISS.




a) comparisons b) assignments (TSS)

c) assignments (ISS)

**Figure 2.** Comparing the geometrical gap sequences.

*it matters what we measure*. The spikes represent the values of $q$ that make many redundant comparisons (this happens f.e. at points where fractional part of $q$ is zero). Also note that the positions of spikes are independent of the data size.

Table 1 shows 10 best common ratios (sorted by value) based on comparisons, assignments, and running times, respectively. The optimal ratios tend to decrease slightly as array size grows. It is not a surprise that using running times we get larger common ratios. They result in shorter sequences which might be better in practice, due to taking into account additional factors that influence running time such as maintenance of loops, memory access, etc.

**Table 1.** Best common ratios for the geometrical gap sequences.

| $N$ | Best common ratios based on comparisons | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 500 | 2.35 | 2.65 | 2.70 | 2.75 | 3.20 | 3.25 | 3.30 | 3.35 | 3.45 | 3.65 |
| 1000 | 2.25 | 2.35 | 2.65 | 2.70 | 2.75 | 2.85 | 3.20 | 3.25 | 3.35 | 3.45 |
| 1500 | 2.25 | 2.35 | 2.65 | 2.70 | 2.75 | 2.85 | 3.20 | 3.25 | 3.35 | 3.45 |
| 2000 | 2.25 | 2.35 | 2.65 | 2.70 | 2.75 | 2.85 | 3.20 | 3.25 | 3.35 | 3.45 |
| $N$ | Best common ratios based on assignments (ISS) | | | | | | | | | |
| 500 | 1.15 | 1.20 | 1.25 | 1.30 | 1.35 | 1.40 | 1.45 | 1.50 | 1.55 | 1.75 |
| 1000 | 1.15 | 1.20 | 1.25 | 1.30 | 1.35 | 1.40 | 1.45 | 1.50 | 1.55 | 1.75 |
| 1500 | 1.15 | 1.20 | 1.25 | 1.30 | 1.35 | 1.40 | 1.45 | 1.50 | 1.55 | 1.65 |
| 2000 | 1.15 | 1.20 | 1.25 | 1.30 | 1.35 | 1.40 | 1.45 | 1.50 | 1.55 | 1.75 |
| $N$ | Best common rations based on running times (ISS) | | | | | | | | | |
| 500 | 2.65 | 2.70 | 3.25 | 3.30 | 3.40 | 3.45 | 4.15 | 4.20 | 4.40 | 4.75 |
| 1000 | 2.65 | 2.70 | 3.20 | 3.25 | 3.30 | 3.45 | 3.65 | 3.75 | 3.85 | 4.15 |
| 1500 | 2.65 | 2.70 | 2.75 | 2.80 | 2.85 | 3.05 | 3.20 | 3.25 | 3.75 | 4.20 |
| 2000 | 2.65 | 2.70 | 2.75 | 2.85 | 3.05 | 3.20 | 3.30 | 3.45 | 3.65 | 3.75 |

## 4 Concluding remarks

We have presented an improved version of the Shell sort, which significantly reduces the number of assignments. As a consequence, the optimal gap sequences should be re-examined. To properly evaluate a gap sequence, a large number of factors should be taken into account. Geometrical sequences are a good starting point of study: they have length of $\Theta(\log N)$, are simple to generate and thus do not require any additional storage space.

Shell sort is suboptimal in the worst-case, but so is *quick sort*, which is one of the best algorithms for sorting. There are still open questions on Shell sort, including finding the best sequence for the average case. Meanwhile, Shell sort still seems like a viable alternative to other comparison-based algorithms.

## References

[1] M. Ciura. Best increments for the average case of shellsort. In *Fund. Comput. Theor.*, pp. 106–117. Springer, 2001.

[2] T. Jiang, M. Li and P. Vitányi. A lower bound on the average-case complexity of shellsort. *J. ACM*, **47**(5), 2000.

[3] D.E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[4] C.G. Plaxton and T. Suel. Lower bounds for shellsort. *J. Algorithms*, **23**(2):221–240, 1997.

[5] V.R. Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford, CA, USA, 1972. AAI7216773.

[6] R. Sedgewick. Analysis of shellsort and related algorithms. In *ESA'96: Fourth Annual European Symposium on Algorithms*, pp. 25–27. Springer, 1996.

[7] R. Sedgewick. *Algorithms in C: Parts 1–4, Fundamentals, Data Structures, Sorting, and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997. ISBN 0201314525.

[8] D.L. Shell. A high-speed sorting procedure. *Commun. ACM*, **2**(7):30–32, 1959.

[9] M.A. Weiss. Empirical study of the expected running time for shellsort. *Comput. J.*, **34**(1):88–91, 1991.

REZIUMĖ

**Empirinis tarpų sekų Shell rikiavimo algoritme tyrimas**

*I.Radavičius, M.Baranauskas*

Šiame tekste pristatoma patobulinta Shell algoritmo versija. Naudojant algoritmą, atliekamas įvairių geometrinių sekų ir Shell algoritmo efektyvumo tyrimas. Tyrimas leidžia teigti, jog tiek priskyrimų skaičius, tiek palyginimų skaičius savaime nėra pakankamas, siekiant tinkamai įvertinti Shell algoritmo veikimą ir pasirinkti optimalią tarpų seką. Pilnam įvertinimui turėtų būti naudojami abu skaičiai, o taip pat matuojamas ir realus algoritmo vykdymo laikas.

*Raktiniai žodžiai*: rikiavimo algoritmas, Shell metodas, tarpų seka.