

A Modified Diminishing Increment Sort for Overcoming the Search for Best Sequence of Increment for Shellsort.

Oyelami, M.O.

Department of Computer and Information Sciences,
Covenant University, Ota, Ogun State, Nigeria.

Abstract: Sorting involves rearrangement of items into ascending or descending order. There are several sorting algorithms but some are more efficient than others in terms of speed and memory utilization. Shellsort improves on Insertion sort by decreasing the number of inversions made on the items to be sorted. In each of its passes, it divides the list into sublists and uses insertion sort technique to sort each sublist. The sequence of increments proposed by Shell has been found to be inefficient. Several researchers have proposed different sequences. Although nobody has been able to determine the best possible sequence of increments, those of Tokuda and Sedgewick have been found to be outstanding. This paper presents a modified diminishing increment sort that overcomes the pain of looking for the best sequence of increments. The results obtained from the experimentation of the proposed algorithm shows its better performance over Tokuda's and Sedgewick's sequences of increment.

Key words: Algorithm, Sorting, Insertion Sort, Shellsort, Modified Diminishing Increment Sort, Worst-case, Best-case and Average-case

INTRODUCTION

For computer to serve as a problem solving machine, it must be directed on what steps to follow in order to get the problem solved. An algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite amount of time^[1]. Algorithms are paramount in computer programming. An algorithm could be of no use even though it is correct and gives a desired output if the resources like time and storage it needs to run to completion are intolerable.

To say that a problem is solvable algorithmically means, informally, that a computer program can be written that will produce the correct answer for any input if we let it run long enough and allow it as much storage space as it needs^[2].

In an algorithm, instructions can be executed any number of times, provided the instructions themselves indicate repetition. However, no matter what the input values may be, an algorithm terminates after executing a finite number of instructions. Thus, a program is an algorithm as long as it never enters an infinite loop on any input^[2].

An algorithm can either be correct or incorrect. A correct algorithm is one that halts with a correct output

while an incorrect algorithm halts with an incorrect output or may not halt at all. An algorithm has five important features^[3]:

- Finiteness: An algorithm must always terminate after a finite number of steps;
- Definiteness: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously specified for each case;
- Input: An algorithm has zero or more inputs-quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs. These inputs are taken from specified sets of objects;
- Output: An algorithm has one or more outputs-quantities that have a specified relation to inputs;
- Effectiveness: An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper.

An algorithm can be described using a computer language. It can also be specified using pseudocode. Pseudocode provides an alternative step between an English language description of an algorithm and an implementation of this algorithm in a programming

language^[4]. Different kinds of problems can be solved by algorithms: sorting, searching, determining the subsequences of the 3 billion chemical base pairs that make up human DNA, etc. There are also a group of problems christened 'hard problems'. These are problems for which no efficient solution is known^[5]. NP-Complete problems are a subset of hard problems and are interesting because although no efficient algorithm has been found for them, no one has ever proved that an efficient algorithm for one cannot exist. They also have the property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them.

Shellsort, named after its inventor, Donald Shell, was one of the first algorithms to break the quadratic barrier, although it was not until several years after its initial discovery that a subquadratic time was proven^[11]. Shellsort is an improvement over insertion sort. The Shellsort technique in each of its passes divides the list into sublists and uses insertion sort for each sublist. The number of sublists decreases from one pass to another and sorting process terminates only when a sublist is left. This final list is again sorted with insertion sort and the sorting method terminates. Since insertion sort method is still used for the final list, it follows that the previous passes are only meant to reduce the number of inversions and consequently reduce the distance each element of the list has to be moved in the final pass to get to its correct position^[6]. The increment, while it is fixed in a pass decreases from one pass to the other until it becomes 1. A popular (but poor) choice for increment sequence is to use the sequence suggested by Shell: $h_1 = [N/2]$ and $h_k = [h_{k-1}/2]$. Several researchers have proposed different sequences. Although nobody has been able to determine the best possible sequence of increments, that of Sedgewick has been found to be the best known in practice^[11] and better results have been reported by N. Tokuda using his sequence of increments^[10]. This paper presents a modified diminishing increment sort that overcomes the barrier of looking for the best sequence of increments. The results obtained from the experimentation of the proposed algorithm shows its better performance over Tokuda's and Sedgewick's sequences of increment.

Arrangement of the Paper:

Objective of the Research: Shellsort improves on insertion sort by introducing a mechanism by which the elements to be sorted can take long leaps instead of short leaps. The sequence of increment proposed by Shell has been found to be poor. One of Sedgewick's sequences has been found to be the best known in practice and better results have been

reported by N. Tokuda using his sequence of increments. The objective of this paper is to develop an algorithm that modifies diminishing increment sort as used by Shell to remove the barrier of looking for the best sequence.

MATERIALS AND METHODS

The proposed algorithm was developed based on the concept of dividing items to be sorted into subsequences and the subsequences sorted just like Shellsort does but using a different approach. Shellsort, using Sedgewick's best sequence, that of Tokuda and the proposed algorithm were implemented on the same platform with the same sets of numbers of varying input sizes for the worst case situation and the results of the number of inversions made were compared and tabulated.

Sorting Algorithms: Given a list of input elements or objects, sorting arranges the elements either in ascending order or descending order and produces a sorted list as the output. The elements to be sorted need to be stored in a data structure for manipulation. Among the various data structures usually used for sorting are: arrays, linked list, heap, etc. Sorting can either be internal or external. Internal sorting is the type of sorting that requires all the elements to be sorted to be in the main memory throughout the sorting process while an external sorting allows part of the elements to be sorted to be outside the main memory during the sorting process^[6]. Examples of internal sorting algorithms are: Insertion Sort, Selection Sort, Bubble Sort, Shellsort, etc. There is no known "best" way to sort; there are many best methods, depending on what is to be sorted, on what machine and for what purpose^[3]. What needs to be done is to learn the characteristics of each sorting algorithm and make a good choice for a particular problem.

Insertion Sort: Insertion Sort assumes the first element in the array is sorted, so we start with the second element. The second element is compared with the first. If it is less than the first, the two swap positions. The third element is picked and compared with the second, if it is less, it is swapped with the second. Otherwise, it remains where it is. Suppose it has been swapped with the second element, it now occupies the second position. It is still further compared with the first element and necessary action taken. The fourth element is taken and the same operations performed until all the elements have been sorted. The algorithm is presented below:

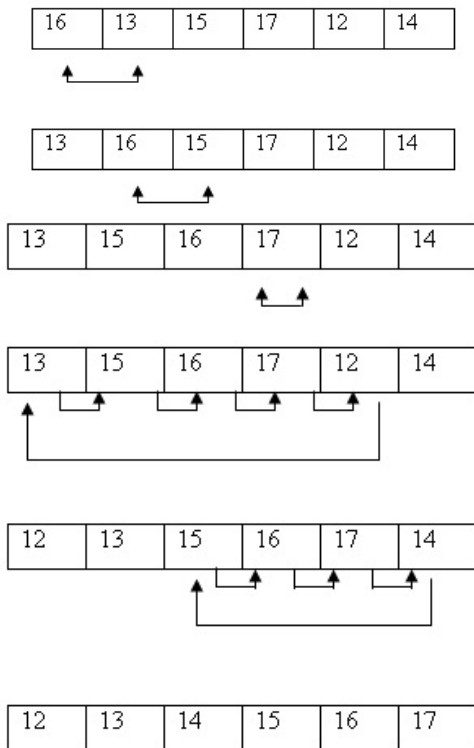
insertionsort (A, size: int)

Begin

- 1 for i =2 to size of A [A is the array, while size is the length of the array A] begin
 - 2 temp = A[i] [temp is a temporary storage]
 - 3 [insert A[i] into the sorted sequence a[1...i-1]
 - 4 j = i -1 [j is 1 position less than the current position of i]
 - 5 while (j > 0 and a[j] > temp) begin
 - 6 A[j + 1] = A[j] [Store A[j] in position (j + 1)]
 - 7 j = j - 1 end
 - 8 A [j + 1] = temp end
- End

The actions performed by the algorithm given the list of numbers below to be sorted in ascending order of magnitude are shown diagrammatically below:

Given list: 16 13 15 17 12 14



Shellsort: Shellsort proposed by Donald L. Shell improves on Insertion Sort by reducing the number of inversions and comparisons made on the elements to be sorted. It sorts an array A with n elements by dividing it into subsequences and sorts the subsequences. The sequence proposed by Shell himself is: $[N/2], [N/4], [N/8], \dots$ ^[10,11,13]. Any sequence $s_1, s_2, s_3, \dots, s_n$ can be used for the subsequences in as much as the last

subsequence is 1. In the first pass, elements that are s_1 distance apart are sorted using insertion sort starting from the first on the list. For the second pass, elements that are s_2 distance apart are sorted using Insertion Sort also by starting from the first. This continues until elements that are 1 distance apart are sorted using straight Insertion Sort. Integer division is carried out on s_1 to get s_2 , integer division also carried out on s_2 to get s_3 and so on. Shellsort is also called Diminishing Increment Sort. The elements to be sorted are assumed to be stored in an array.

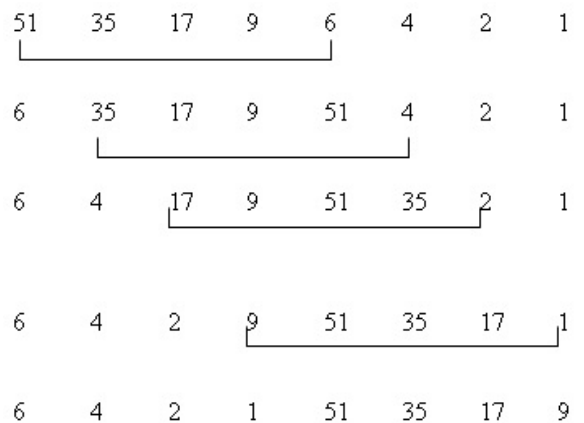
Consider the worst-case problem of sorting the following elements in ascending order using Shell's sequences:

51 35 17 9 6 4 2 1

Let us take $s_1 = 4$ to be the initial value.

First Pass

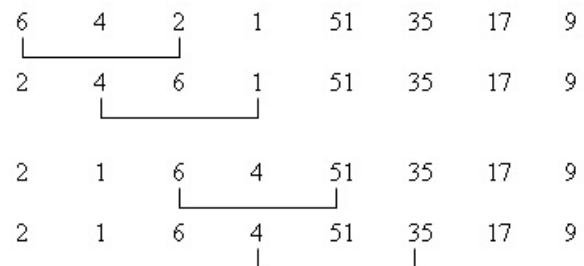
For the first pass, numbers that are 4 distance apart are sorted. They are sorted in ascending order as follow:



Second Pass

$$s_2 = s_1 \div 2 = 4 \div 2 = 2$$

For the second pass, numbers that are 2 distance apart are sorted. They are sorted in ascending order as follow:



```

2   1   6   4   51  35  17  9
           └───┬───┘
2   1   6   4   17  35  51  9
                   └───┬───┘
2   1   6   4   17  9   51  35
    
```

Third Pass

$$s_3 = s_2 \div 2 = 2 \div 2 = 1$$

Numbers that are 1 distance apart are sorted as shown below.

```

2   1   6   4   17  9   51  35
└───┬───┬───┬───┬───┬───┬───┬───┘
    
```

After sorting each one with straight Insertion Sort we will have the following sorted list:

```
1  2   4   6   9   17  35  51
```

The algorithm is presented below:

```

shellsort(A,size:int)
Begin
1  increment = size/2 [ increment here represents s1,
   s2, ..., sl described above]
2  while(increment ≥ 2) begin
3  i = 1
4  while(i+increment) ≤ size begin
5  if array[i] > array[i + increment] swap the two
   i=i+1 end
6  increment = increment / 2 end
   [call Insertion Sort function to sort the array with
   increment =1 ]
7  insertsort(A, size:int)
End
    
```

Insertsort function in line 8 of the algorithm above applies insertion sort on the whole array when increment is 1. In this algorithm, we have assumed that for each array to be sorted, elements that are (size/2) distance apart are first sorted.

Proposed Algorithm (Modified Diminishing Increment Sort): This proposed sorting algorithm also divides the elements to be sorted into subsequences just like Shellsort does but by first of all comparing the first element with the last. If the last is less than the first, the two swap positions, otherwise, they maintain their positions. Later, the second element is compared

with the second to the last, if the second to the last element is smaller than the second, they are swapped. Otherwise, they maintain their positions. This process continues until the last two consecutive middle elements are compared or until it remains only one element in the middle. After this, straight Insertion Sort is applied to sort the elements that are 1 distance apart just as Shellsort does. This approach reduces the number of comparisons and inversions made for the whole sorting process compared with when Shellsort is used (using Shell's increments).

Consider the worst-case scenario of sorting the following elements used for Shellsort in ascending order:

```
51  35  17  9   6   4   2   1
```

The algorithm works like this:

```

51  35  17  9   6   4   2   1
└───┬───┬───┬───┬───┬───┬───┬───┘
1   35  17  9   6   4   2   51
           └───┬───┬───┬───┬───┬───┘
1   2   17  9   6   4   35  51
                   └───┬───┬───┬───┬───┬───┘

1   2   4   9   6   17  35  51
           └───┬───┬───┬───┬───┬───┘

1   2   4   6   9   17  35  51   (*)
    
```

Insertion Sort is now applied to (*) to sort the elements 1 distance apart. But as can be seen, no inversion will be applied on (*) because each element is already its rightful position. As seen above, there are only four moves. The proposed algorithm as can be seen performs better than Shellsort when the number of comparisons and inversions made in the two cases are compared. The algorithm is presented below:

```

Modified Diminishing Increment Sort (array, size)
Begin
1  i = 1
2  j = size
3  while( i < j) do begin
4  if array[i] > array[j] swap( array, i, j)
5  i = i + 1
6  j = j - 1 end
   [call insertion sort function to sort the array with
   increment =1 ]
7  insertsort(A, size:int)
End
    
```

Choice of Increments: The problem with Shell's increments is that pairs of increments are not necessarily relatively prime and thus the smaller increment can have little effect. Hibbard suggested a slightly different increment sequence, which gives better results in practice (and theoretically). His increments are of the form $1, 3, 7, \dots, 2^k - 1$. Although these increments are almost identical, the key difference is that consecutive increments have no common factors^[11,14]. Papernov and Stasevich suggested the form 2^{k+1} ^[10,15]. Other sequences proposed are: $(2^k - (-1)^k)/3$ and $(3^k - 1)/2$, Fibonacci numbers and the Incerpi-Sedgewick sequences for $\rho = 2.5$ and $\rho = 2$, Pratt-like sequences $\{5^p 11^q\}$ and $\{7^p 13^q\}$ ^[10]. According to^[11], Sedgewick's sequence $\{1, 5, 19, 41, 109, \dots\}$ in which the terms are either of the form $9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$ ^[12] is the best known in practice although^[12] reports that better results have been reported by N. Tokuda using the quantity $2.25h_s$ in place of $3h_s$ in the sequence $h_0 = 1, h_{s+1} = 3h_s + 1$.

We can compare Sedgewick's sequence and the proposed algorithm performances by considering the problem of sorting the following already considered:

51 35 17 9 6 4 2 1

Using Sedgewick's sequence $\{1, 5\}$, $s_1 = 5$ is the initial value.

First Pass

For the first pass, numbers that are 5 distance apart are sorted. They are sorted in ascending order as follow:

51 35 17 9 6 4 2 1
└──────────────────┘

4 35 17 9 6 51 2 1
└──────────────────┘

4 2 17 9 6 51 35 1
└──────────────────┘

4 2 1 9 6 51 35 17

Second Pass

For the second pass, $s_2 = 1$ and straight insertion sort is used to sort the elements 1 distance apart.

4 2 1 9 6 51 35 17 (**)

Insertion Sort is therefore used to sort (**) using elements that are 1 distance apart:

4 2 1 9 6 51 35 17
└──────────────────┘

After sorting each one with straight Insertion Sort we will have the following sorted list:

1 2 4 6 9 17 35 51

In all, we have ten swappings (movements/inversions). When compared with what we obtained above with the proposed algorithms (only four swappings), the proposed algorithms performs better than Shellsort with Sedgewick's sequence. According to^[10] if the size of the input is greater than 1000, Sedgewick's sequence is recommended and if N is less than 1000, a simple rule such as $h_0 = 1, h_{s+1} = 3h_s + 1$ seems to be about as good as any other.

Performance Analysis of Algorithms: The most important attribute of a program/algorithm is correctness. An algorithm that does not give a correct output is useless. Correct algorithms may also be of little use. This often happens when the algorithm/program takes too much time than expected by the user to run or when it uses too much memory space than is available on the computer^[7]. Performance of a program or an algorithm is the amount of time and computer memory needed to run the program/algorithm. Two methods are normally employed in analyzing an algorithm:

- Analytical method
- Experimental method

In analytical method, the factors the time and space requirements of a program depend on are identified and their contributions are determined. But since some of these factors are not known at the time the program is written, an accurate analysis of the time and space requirements cannot be made. Experimental method deals with actually performing experiment and measuring the space and time used by the program. Two manageable approaches to estimating run time are^[7]:

- Identify one or more key operations and determine the number of times they are performed;
- Determine the total number of steps executed by the program.

Worst-case, Best-case and Average-case Analysis of Sorting Algorithms: The worst-case occurs in a sorting algorithm when the elements to be sorted are in reverse order. The best-case occurs when the elements

are already sorted. The average-case may occur when part of the elements are already sorted. The average-case has data randomly distributed in the list^[8]. The average case may not be easy to determine in that it may not be apparent what constitutes an ‘average’ input. Concentration is always on finding only the worst-case running time for any input of size n due to the following reasons^[5]:

- The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst-case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm’s worst-case will often occur when the information is not present in the database. In some searching applications, searches for absent information may be frequent.
- The “average-case” is often roughly as bad as the worst case.

Analysis of Shellsort and the Proposed Algorithm:

Analysis of Shellsort is very difficult and incomplete. A complete analysis is extremely difficult and requires answers to some mathematical problems that have not yet been solved^[2,3]. The running time of Shellsort depends on the choice of increment sequence and the proofs can be rather complicated. The average-case analysis is a long-standing open problem, except for the trivial increment sequences^[9]. According to^[10], five factors determine the execution time of Shellsort:

- the size of the file;
- the number of passes (i.e the number of increments);
- the sum of the increments;
- the number of comparisons and
- the number of moves.

The factor that governs the running time is the number of moves or inversions according to^[10]. The approach used in this paper is therefore, is to compare the number of moves in each case of the algorithms in the worst case scenario for various input sizes.

Results Obtained: The three algorithms were implemented and compiled using Turbo C++ 4.5 compiler on an Intel Celeron M microcomputer running Windows Vista™ Basic using the same set of data for each input size. The results obtained showing the number of inversions made in each case are summarized in the table below:

Size of Input	Number of Inversions Carried Out		
	Shellsort (Using Sedgewick’s Sequence)	Shellsort (Using Tokuda’s Sequence)	Proposed Algorithm
20	96	22	10
101	1040	272	50
500	9636	2546	250
700	6178	3078	350
900	6142	14830	450
1000	5024	18248	500
1100	5544	9058	550
2019	33751	61123	1009

From the results obtained, Tokuda’s increment performs better than Sedgewick’s for small input sizes but as the size increases, Sedgewick’s sequence performs better but in all situations, the proposed algorithm performs better than the two.

The number of inversions has a direct effect on the time; the lower the number of inversions, the shorter the time taken to complete the sorting. It can then be concluded that the proposed algorithm is the fastest out of the three considered.

Conclusion: The proposed algorithm obviously from the results obtained performs better than Shellsort using Sedgewick’s sequence and Tokuda’s. With this result, the pain of looking for a possible best sequence is overcome. We therefore, conclude that this proposed algorithm will run faster for all input sizes than Shellsort using both Tokuda’s and Sedgewick’s sequences which are acclaimed to be outstanding and as such, overcomes the headache of looking for the best possible sequence for Shellsort.

REFERENCES

1. Alfred V. Aho, John Horroroft and D. Jeffrey Ullman, 2002. Data Structures and Algorithms. Pearson Education Asia.
2. Sara Baase and Allen Gelder, 2000. Computer Algorithms (Introduction to Design & Analysis). Addison Wesley Longman.
3. Donald E. Knuth, 1997. The Art of Computer Programming, Volume I, Fundamental Algorithms; Third Edition. Addison-Wesley.
4. Kenneth H. Rosen, 2003. Discrete Mathematics and its Applications. McGrawHill.
5. Thomas H. Cormen, E. Charles Leiserson, L. Ronald Rivest and Clifford Stein, 2003. Introduction to Algorithms. The Massachusetts Institute of Technology.
6. Shola P.B., 2003. Data Structures With Implementation in C and Pascal. Reflect Publishers.
7. Sartaj Sahni, 2000. Data Structures, Algorithms and Applications in Java. McGrawHill.

8. William Ford and William Topp, 2002. Data Structures With C++ Using STL. Prentice Hall.
9. Mark Allen Weiss, 2006. Data Structures and Algorithm Analysis in C++. Pearson Addison Wesley.
10. Donald E. Knuth, 1998. The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition. Addison-Wesley.
11. Mark Allen Weiss, 2006. Data Structures and Algorithm Analysis in C++. Pearson Education. Inc.
12. Sedewick R., 1998. Algorithms in C++. Parts 1-4 (3rd edition). Addison-Wesley, Reading Mass.
13. Shell D.L., 1959. A High-Speed Sorting Procedure. Communications of the ACM, 2: 30-32.
14. Hibbard T.H., 1963. An Empirical Study of Minimal Storage Sorting". Communications of the ACM, 6: 206-213.
15. Papernov A.A. and G.V. Stasevich, 1965. A Method of Information Sorting in Computer Memories. Problems of Information Transmission, 1: 63-75.