Journal of Computer Science & Its Application, December 2007, Vol. 14, No. 2

Improved Shellsort for the Worst-Case, the Best-Case and a Subset of the Average-Case Scenarios

Oyelami, M.O *., Azeta, A.A. ** and Ayo, C.K.***
Department of Computer and Information Sciences, Covenant University, Ota, Ogun State, Nigeria
*olufemioyelami@yahoo.com, **azeta_ambrose@yahoo.com, ***ckayome@yahoo.com

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 comparisons made on the items to be sorted.

This paper presents an Improved Shellsort algorithm that further decreases the number of comparisons made on the items to be sorted through a modified diminishing increment sort.

The results obtained from the implementation of both Shellsort and the proposed algorithm shows that the proposed algorithm has a fewer number of comparisons made for all input sizes of the best and worst cases and for input size of twenty or less for the average case.

By implication, this means that the proposed algorithm is faster in these situations. The strength of the algorithm however diminishes for only the average case of input size greater than twenty.

Keywords: Algorithm, Sorting, Insertion Sort, Shellsort, Improved Shellsort, Worst-case, Best-case and Average-case.

1. Introduction

For computer to serve as a problem solving machine, it must be directed 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]:

(i) teness: An algorithm must always terminate after a finite number of steps;

(ii) Definiteness: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously specified for each case;

(iii)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;

(iv)Output: An algorithm has one or more outputs- quantities that have a specified relation to inputs;

(v) 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. This paper examines only sorting algorithms and specifically Insertion Sort and Shellsort and improves on the latter for the worst-case, best-case and a subset of the average-case scenario.

Arrangement of the Paper

2. Objective of the Research

Shellsort improves on Insertion sort by decreasing the number of comparisons made and hence the time taken to complete the sorting, the main of objective of this work is the development an algorithm that also improves on Shellsort by further decreasing the number of comparisons made on the items to be sorted in order to know the position each item will occupy. By implication, the time taken to run an algorithm to completion also decreases with a decreased number of comparisons.

3. Methodologies

Improved Shellsort 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 and Improved Shellsort algorithms were implemented on the same platform with different sets of numbers of varying input sizes for the best case, average case and the worst case situations and the results of the number of comparisons made in each situation which also affects the running time were compared and tabulated.

4. 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.

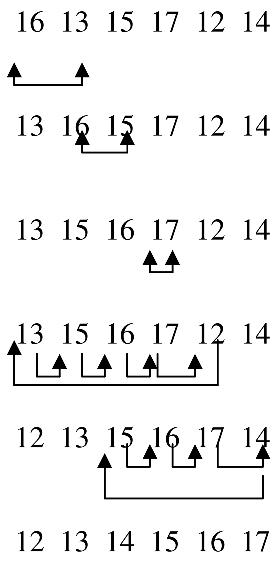
4.1 Insertion Sort

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]
    [insert A[i] into the sorted sequence a[1…i-1]
3)  j = i -1 [j is 1 position less than the current position of i]
4)  while (j > 0 and a[j] > temp)
    begin
5)  A[j + 1] = A[j] [Store A[j] in position (j + 1) ]
6)  j = j - 1
      end
7)  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



4.2 Shellsort

Shellsort proposed by Donald L. Shell improves on Insertion Sort by reducing the number of comparisons made. It sorts an array **A** with n elements by dividing it into subsequences and sorts the subsequences. Any sequence s_1, s_2, $s_{3,\ldots,}$ 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:

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:

51 35 17 9 6 4 2 1

6 35 17 9 51 4 2 1

6 4 17 9 51 35 2 1

6 4 2 9 51 35 17 1

6 4 2 1 51 35 17 9

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:

6 4 2 1 51 35 17 9

2 4 6 1 51 35 17 9

2 1 6 4 51 35 17 9

2 1 6 4 51 35 17 9

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

For the average-case, consider the problem of sorting the same set of numbers with the following arrangement:

51 17 35 9 4 1 2 6

First Pass

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

51 17 35 9 4 1 2 6

4 17 35 9 51 1 2 6

4 1 35 9 51 17 2 6

4 1 2 9 51 17 35 6

4 1 2 6 51 17 35 9

Second Pass

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

4 1 2 6 51 17 35 9

2 1 4 6 51 17 35 9

2 1 4 6 51 17 35 9

2 1 4 6 51 17 35 9

2 1 4 6 51 17 35 9

2 1 4 6 35 17 51 9

2 1 4 6 35 9 51 17

Third Pass

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

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, ..., 1
     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
6.   i=i+1
       end
7.   increment = increment / 2
     end
     [call insertion sort function to
     sort the array with increment =1 ]
8.   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. The constant 2 used can be changed.

4.2.1 Different Sequences Proposed for Shellsort

The sequence originally proposed by Shell is [N/2], [N/4], [N/8],….But, it has been found out that this sequence is not good enough and as such, different researchers have proposed different sequences: Hibbard proposed the sequence is 1,3,7,…,2^K-1 [7,8]. The sequence 2^K+1 was proposed by Papernov and Statsevich. Other sequences proposed are: (2^k- $(-1)^k$/3 and (3^k-1)/2, Pratt-like sequences {5^p11^q} and {7^p13^q}, Fibonacci numbers, the Incerpi Sedgewick's sequences for ρ =2.5 and ρ=2 as well as his sequence {1,5,19,41,109,…} in which the terms are either of the form $9.4^i - 9.2^i + 1$ or $4^i - 3.2^i + 1$ and N. Tokuda's sequence $h_0 = 1$, $h_{s+1} = 2.25h_s + 1$ [9,10].

4.3 Improved Shellsort

Improved Shellsort is the proposed sorting algorithm which is an improvement over the Shellsort algorithm. 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 made for the whole sorting process compared with when Shellsort is used for the worst-case, the best-case and small input size for average-case.

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

The Improved Shellsort for the worst-case scenario as can be seen performs better than Shellsort when the number of comparisons made in the two cases are compared.

For the average-case of sorting the same set of numbers used for Shellsort above, consider the following:

51 17 35 9 4 1 2 6

51 17 35 9 4 1 2 6

6 17 35 9 4 1 2 51

6 2 35 9 4 1 17 51

6 2 1 9 4 35 17 51

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

A call is now made to straight Insertion sort to sort these last numbers.

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

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

1 2 4 6 9 17 35 51

It is worthy of note that in the average-case scenario of both algorithms before straight Insertion sort is called, three boldened numbers are already in their correct positions in the case of Improved Shellsort and only two in the case of Shellsort when **(*)** and **(**)** above are compared. It is obvious that when the total number of comparisons made are compared in the two cases after performing straight Insertion sort on both **(*)** and **(**)**, Improved Shellort performs better. The algorithm is presented below:

```
improvedShellSort( 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
```

5. 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 [11]. 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:

i. Analytical method
ii. 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 [11]:

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

5.1 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 [12]. 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]:

i. 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.

ii. 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.

iii. The "average-case" is often roughly as bad as the worst case.

		Number of Comparisons Carried Out	
Case	**Size of Input**	**Shellsort**	**Improved Shellsort**
Worst-case	10	19	5
Best-case	10	13	5
Average-case	10	19	13
Worst-case	20	55	10
Best-case	20	43	10
Average-case	20	59	50
Worst-case	50	180	25
Best-case	50	154	25
Average-case	50	254	296
Worst-case	100	456	50
Best-case	100	404	50
Average-case	100	672	1183

5.2 Analysis of Shellsort and Improved Shellsort for the Worst-case and Best- case Scenarios

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 [7]. Since Shellsort improves on Insertion Sort by decreasing the number of comparisons made, the approach employed here in comparing Shellsort with this proposed algorithm is to compare the number of comparisons made in each case.

6.0 Results Obtained

The two algorithms were implemented and compiled using Turbo C++ 4.5 compiler on an Intel Celeron M microcomputer running Windows Vista^{TM} Basic. The results obtained showing the number of comparisons made in each case are summarized in the table below:

Table I: Number of Comparisons

The number of comparisons has a direct effect on the time; the lower the number of comparisons, the shorter the time taken to complete the sorting. For any input size **n** for the worst and the best cases, the number of comparisons carried out by the Improved Shellsort is half the size of the input**,** that is, **Number of comparisons = n/2.** For input size **n** greater than 1 for the worst case for

Shellsort, the minimum number of comparisons made is **n** and for the best case the minimum number of comparisons is $^{n}/_{2}$. The growth rate of the number of comparisons made in the worst case is higher than that of the best case as the size of **n** increases for Shellsort.

7. Conclusion

The Improved Shellsort algorithm obviously from the results obtained performs better than Shellsort in the worst-case, the best-case and a small size input of the average-case. The strength of this algorithm becomes more appreciated as the size of the input to it increases for the worst-case and best-case scenarios but when input size begins to be higher than twenty its strength diminishes for the average-case. Implementing the two algorithms on a different platform may produce different running time results but the same pattern will of course show. We therefore, conclude that this proposed Improved Shellsort will run faster than Shellsort for the worst-case, best-case and a subset of the average-case.

8. References

[1] Alfred V. Aho, John Horroroft and Jeffrey D. 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, Charles E. Leiserson, Ronald L. 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] Mark Allen Weiss (2006). Data Structures and Algorithm Analysis in C++. Pearson Education. Inc.

[8] Hibbard T. H., "An Empirical Study of Minimal Storage Sorting", Communications of the ACM, Vol. 6, Number 5, 1963, pp. 206 – 213.

[9] Donald E. Knuth (1998). The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition. Addison-Wesley.

[10] Papernov A. A. and Stasevich G. V., "A Method of Information Sorting in Computer Memories", Problems of Information Transmission, Vol. 1, Number 3, 1965, pp. 63 – 75.

[11] Sartaj Sahni (2000). Data Structures, Algorithms and Applications in Java. McGrawHill.

[12] William Ford and William Topp (2002). Data Structures With C++ Using STL. Prentice Hall.

About the Authors

Olufemi Moses Oyelami holds both BSc and MSc in Computer Science and currently teaches the same in Covenant University, Ota, Ogun State, Nigeria. He is a member of both Computer Professional Registration Council of Nigeria (CPN) and Nigerian Computer Society (NCS). He is a PhD student of Computer Science in the Department of Computer and Information Sciences, Covenant University, Ota. Algorithms, Programming Languages and Mobile Computing are his current research interests.

Azeta, A.A. is a Ph.D. student in the Department of Computer and Information Sciences, Covenant University, Ota, Nigeria. He holds B.Sc. and M.Sc. in Computer Science. His current research interests are in the following areas: Software Engineering, Algorithm Design and Mobile Computing. He is a member of the Nigerian Computer Society (NCS), and Computer Professional Registration Council of Nigeria (CPN).

Charles K. Ayo holds a B.Sc. M.Sc. and Ph.D in Computer Science. His research interests include: mobile computing, Internet programming, e-business and government, and object oriented design and development. He is a member of the Nigerian Computer Society (NCS), and Computer Professional Registration Council of Nigeria (CPN). He is currently the Head of Computer and Information Sciences Department of Covenant University, Ota, Ogun state, Nigeria, Africa. Dr. Ayo is a member of a number of international research bodies such as the Centre for Business Information,Organization and Process Management (BIOPoM), University of Westminister. http://www.wmin.ac.uk/wbs/page-744; the Review Committee of the European Conference on E-Government, http://www.academic-conferences.org/eceg/; and the Editorial Board, Journal of Information and communication Technology for Human Development.