Interpolation Sort and Its Implementation with Strings

Gourav Saha and S. Selvam Raju

***Abstract*—In this paper we propose a new array sorting algorithm with average and best time complexity of O(n). Its best, worst and average time complexity has been analysed. Also the difficulty of applying this algorithm with strings has been discussed and its solution too is found. The limitation of the solution is also analysed.**

***Index Terms*—Sorting algorithm, interpolation, sub-arrays, time complexity, strings.**

I. Introduction

There are basically two kind of sorting algorithms, $O(n^2)$ and $O(n \times Log(n))$. $O(n^2)$ algorithms takes more time but less while $O(n \times Log(n))$ algorithms takes less time but more space and as described by Knuth[1]. So $O(n^2)$ algorithms are preferred for small arrays and $O(n \times Log(n))$ for large arrays There are sorting algorithms with $O(n)$ time complexity too like Radix sort but with limitation on the range of the data.

A new sorting algorithm which derives its motivation from interpolation search is proposed [2]-[5]. It shows a high probability to show $O(n)$ time complexity for a well distributed data. The algorithm has a disadvantage of large code size and taking a lot of RAM memory for sorting. Also, it can't be used to sort large strings.

II. Explanation of the Algorithm

The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example and then discuss how to implement the same in reality.

Consider the following unsorted array of size 15:

56	32	12	65	37	80	55	60	40	77	50	9	68	35	20

The backbone of the sorting algorithm is the interpolation formula as described by GH Gonnet [4]:

$$IPOS[i] = SPOS + (N-1) \times \frac{(DATA[i] - DATA[MIN])}{(DATA[MAX] - DATA[MIN])}$$

where,

IPOS[i] → Interpolated position of the ith element of the unsorted array.

SPOS → Starting index of the array.

N → Number of elements in the array

DATA[i] → Data at the ith position of the unsorted array

DATA[MIN] → Smallest data of the array.

DATA[MAX] → Largest data of the array.

It is to be noted that the division performed in the formula is integer division, i.e. decimal part is ignored.

For the given array,

SPOS=1

N=15

DATA[MAX]=80

DATA[MIN]=9

Substituting these values in the interpolation formula we get the interpolated positions of the elements as,

56	32	12	65	37	80	55	60	40	77	50	9	68	35	20	DATA
10	5	1	12	6	15	10	11	7	14	9	1	12	6	3	IPOS

Rearranging the array from smaller to bigger IPOS we get:

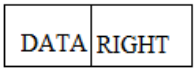
12	9	20	32	37	35	40	50	56	55	60	65	68	77	80	DATA
1	1	3	5	6	6	7	9	10	10	11	12	12	14	15	IPOS

So we see that most of the elements got sorted but there are few groups of elements whose IPOS values turned out to be the same. These groups of elements are treated as sub-arrays. The above technique is applied on each of these sub-arrays until we get no further sub-arrays.

To achieve the same algorithm in reality we have to find a way to "rearrange the array from smaller to bigger IPOS".

Two structure types have to be used to do the above effectively. They are:

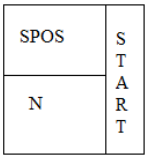
A. NODE1



The data type of the field DATA is that of the data being sorted.

RIGHT is a pointer of type NODE1.

B. NODE2



START is a pointer of type NODE1.

SPOS and N are integer variables.

An array "BEG" of type NODE2 is used to store the information of the sub-arrays yet to be sorted. The starting index of the sub-array in the main array is stored in SPOS while N contains the number of elements in the sub-array. The information of the sub-array to be sorted is always present in BEG [1], i.e. the first element of BEG. Using the

Manuscript received June 24, 2012; revised August 7, 2012.

Gourav Saha is with the Madras Institute of Technology, Anna University, India (e-mail: sahahotmail@hotmail.com).

information of the sub-array present in BEG[1] the sub-array is sorted.

To do this another array "SUBARRAY" of type NODE2 is used. The size of SUBARRAY is equal to the BEG [1]. *N*, i.e. number of elements in the sub-array to be sorted. Each and every element of SUBARRAY represents an interpolated position. For a sub-array with N elements there are N possible interpolated positions possible, so the size of SUBARRAY is equal to the number of elements in the sub-array. The sub-array to be sorted is traversed to interpolate the position of the element. The elements are then stored as a linked list in SUBARRAY according to their interpolated position. The first element of every linked list is pointed by the field START of NODE2. The value of N for every element of SUBARRAY is incremented to represent the number of nodes in the corresponding linked list.

The linked list of SUBARRAY represent the smaller sub-arrays created in the process of sorting the main sub-array. These smaller sub-arrays are transferred to the actual array sequentially, i.e. starting from the linked list pointed by the first element of SUBARRAY to the last. Now the sub-arrays whose number of elements is less than or equal to 3 is directly sorted using Bubble Sort. Else the information of the sub-array is stored in the BEG. To do this the information of these sub-arrays is stored in an array NEWBEG of type NODE2. Then the information of the arrays stored in BEG which are yet to be sorted is transferred to NEWBEG, i.e. all the elements of BEG except BEG [1]. After that the array BEG is deleted and NEWBEG is renamed as BEG.

So basically the BEG is acting like a stack containing information of those sub-arrays which are yet to be sorted.

III. The Algorithm

Input: Unsorted array ARRAY[] of size SIZE.
Output: Sorted array ARRAY[] of size SIZE.
INTERSORT (ARRAY[], SIZE)

1) Make an array BEG of type NODE2 and size 1 using dynamic memory allocation.
2) Set BEG [1].SPOS=1, BEG [1]. *N*=SIZE and BEG [1].START=NULL.
3) Set NUM = 1.
4) Repeat steps 5 to 14 while NUM ≠ 0
5) Traverse ARRAY from index BEG [1].SPOS to BEG[1].SPOS + BEG[1].*N* – 1 and find the maximum and the minimum data. Save them as MAX and MIN respectively.
6) Make an array SUBARRAY of type NODE2 and size BEG [1]. *N* using dynamic memory allocation . For each and every element of SUBARRAY initialize there fields *N*=-1 and START=NULL.
7) Set A = 0.
8) Traverse ARRAY from index BEG [1].SPOS to BEG[1].SPOS + BEG[1]. *N* – 1 and for each and every data in this range:
   a) Interpolate the position IPOS of the data in SUBARRAY using interpolation formula. For interpolation take SPOS = 1, *N* = BEG [1].*N*, DATA[MIN] = MIN and DATA[MAX]=MAX.
   b) Save the element in the memory location pointed by SUBARRAY [IPOS].START. To do this create a variable VAR of type NODE1. Set VAR.DATA = THE DATA, VAR.RIGHT = SUBARRAY[IPOS].START. Then Point SUBARRAY[IPOS].START to VAR.
   c) If SUBARRAY[IPOS]. *N* = -1 set it to 1. Else increase it by 1.
   d) If SUBARRAY[IPOS].N = 4 then increment A by 1.
9) Set NUM = NUM + *A* – 1.
10) Make an array NEWBEG of type NODE2 of size NUM using dynamic memory allocation.
11) Traverse SUBARRAY and for each and every element for which $N \neq -1$
   a) Set the field SPOS = BEG[1].SPOS for very first element for which $N \neq -1$. For other elements SPOS = TEMP.
   b) Set TEMP = SPOS + *N*.
   c) Copy all the data from the memory location pointed by START to the ARRAY in consecutive array indices starting from index SPOS in the actual array. Delete the memory locations pointed by START.
   d) If *N*<=3 then sort the data in ARRAY using bubble or insertion sort.
   e) If *N*>3 copy the element in NEWBEG.
12) Copy all the elements of BEG to NEWBEG except BEG [1].
13) Delete BEG and SUBARRAY.
14) Set NEWBEG as BEG.

IV. Implementation with Strings

The interpolation formula is:

$$IPOS[i] = SPOS + (N-1) \times \frac{(DATA[i] - DATA[MIN])}{(DATA[MAX] - DATA[MIN])}$$

The interpolation formula involves arithmetic operations which is not possible with strings. So before carrying out the sorting we need to represent each and every string with a numerical value such that:

1) Numerical value is unique for each string.
2) Lexically greater strings have greater numerical value.

We have,

STRING = "*C*1 *C*2 *C*3…….. *C*(*m*-1) *Cm*"

Is a string of 'm' characters, where *C*1, *C*2, *C*3…….. *C*(*m*-1), Cm belongs to a domain containing 'N' characters. Each character in this domain have a numerical value starting from '0' to '(*N*-1)' depending on there ASCII code. The character with higher ASCII code has greater numerical value. Let the numerical value of a character *Ci* be represented by *A*[*Ci*].

The formula to assign numerical value to the string is:

$$A[C1] \times N^{0} + A[C2] \times N^{-1} + A[C3] \times N^{-2} + \ldots\ldots\ldots + A[Cm] \times N^{-(m-1)}$$

or

$$\sum_{i=1}^{m} A[Ci] \times N^{-(i-1)}$$

The above formula is same as the one used to convert number of any base to base 10. Since each and every number of a base has an unique decimal representation and numerically greater number in a base will have numerically greater decimal representation so, the above formula will successfully assign numerical values to string meeting the desired requirement.

So the string "HELLO" whose character domain is all upper case alphabets will have a numerical value

= 7 × 26^(0) + 4 × 26^(-1) + 11 × 26^(-2) + 11 × 26^(-3) + 14 × 26^(-4)

= 7.163822

This method has a limitation.

Consider two strings with the difference only in the ith character. The i^{th} character differs lexically by unit 1 only.

The numerical difference of the two string = $N^{-(i-1)}$

However large be "*i*" an ideal method should be able to assign separate numerical values to each of these strings. But each and every compiler has a limit till which it can differentiate a decimal number. For *C* compiler it is till 5^{th} decimal place. So for *N*=26, i.e. the number of upper case English alphabets, the compiler won't be able to distinguish between two strings if *i*>4.

V. Correctness of the Algorithm

The necessary and sufficient condition to prove the correctness of the algorithm is to prove that the interpolation formula when applied to an array at least create two sub arrays. In that way the array will finally get sorted.

Proof:

The interpolation formula is:

$$IPOS[i] = SPOS + (N-1) \times \frac{(DATA[i] - DATA[MIN])}{(DATA[MAX] - DATA[MIN])}$$

So IPOS for the minimum element will be SPOS and the IPOS for the maximum element will be (SPOS+*N*-1).

So whatever be the IPOS of the other elements of the array we will be getting two or more sub-arrays.

Hence we prove the correctness of the interpolation sort algorithm.

VI. Analysis of Best and Worst case Time Complexity

The sorting algorithm shows its worst time complexity of $O(n^2)$ for those arrays which when sorted have data agreeing the following relationship:

$$DATA[i] > (i-1) \times (DATA[i-1] - DATA[1]) + DATA[1]$$

In such cases for each and every traversal of the whole array only two sub arrays will be generated, one of size (N-1) and the other of size 1, where *N* is the size of the array which is broken down into sub arrays. Hence $\frac{N*(N+1)}{2}$ iterations will be required to sort the whole array.

An example of such an array is:

1238	2	4	1	11	8661	42	69282	623532	207

This array when sorted yields:

1	2	4	11	42	207	1238	8661	69282	623532

All the data in the sorted array satisfies the above relationship.

The best time complexity of $O(n)$ is shown for those arrays which when sorted have data agreeing the following relationship:

$$\frac{(i-1) \times (DATA[N] - DATA[1])}{(N-1)} + DATA[1] <= DATA[i] < \frac{i \times (DATA[N] - DATA[1])}{(N-1)} + DATA[1]$$

In such cases all the elements of the array will be allotted unique IPOS values and hence the whole array will get sorted in a single traversal. Hence N iterations will be required to sort the array.

An example of such an array is:

102	50	256	14	215	156	130	68	176	237

This array when sorted yields:

14	50	68	102	130	156	176	215	237	256

All the data in the sorted array satisfies the above relationship.

It can be easily seen that an unsorted array which gives $O(n^2)$ time complexity is very unlikely while that which gives $O(n)$ time complexity is very common. Hence this algorithm has a high tendency to show $O(n)$ time complexity.

VII. Average Time Complexity Analysis

Consider an array with N elements. The number of iterations needed to sort the array is the sum of the number of iterations required to traverse the array once to find the interpolated position of the *N* elements and the time taken to sort the sub arrays generated.

The number of iterations required to traverse the array once to find the interpolated positions of the N elements is *N*.

If $n_1, n_2, n_3, \ldots\ldots\ldots, n_N$ are the number of elements in the sub arrays generated corresponding to interpolated positions 1,2,3,……..,*N* respectively then, the number of iterations T(N) required to sort the array is given by,

$$T(N) = N + [T(n_1) + T(n_2) + T(n_3) + \ldots\ldots + T(n_N)] \quad (1)$$

But,

$$n_1 + n_2 + n_{3+} \ldots\ldots\ldots + n_N = N \quad (2)$$

There are many possible combinations of

$n_1, n_2, n_3, \ldots\ldots\ldots\ldots, n_N$ which satisfies equation (2). So the average iterations $T(N)$ required to sort the array is given by,

$$T(N) = N + \frac{\sum[T(n_1)+T(n_2)+T(n_3)+\ldots\ldots+T(n_N)]}{M} \quad (3)$$

where

$M \rightarrow$ Number of possible combinations of $n_1, n_2, n_3, \ldots\ldots\ldots\ldots, n_N$ satisfying equation (2).

So M is basically the number of solutions of equation (2) such that $n_1, n_2, n_3, \ldots\ldots\ldots\ldots, n_N$ is between 0 to (N-1). Now the number of solution of the equation

$$n_1 + n_2 + n_{3+} \ldots\ldots\ldots + n_M = N$$

where $n_1, n_2, n_3, \ldots\ldots\ldots\ldots, n_M$ are integers ranging from 0 to N is given by following formula [6],

$$\underset{M-1}{\overset{M+N-1}{C}} \quad (4)$$

Here M=N so,

$$L = \underset{N-1}{\overset{2N-1}{C}} - N \quad (5)$$

where, $\underset{m}{\overset{n}{C}} = \frac{n!}{m! \times (n-m)!}$

Now,

$$\sum[T(n_1)+T(n_2)+T(n_3)+\ldots\ldots+T(n_N)]$$
$$= \sum[K_0 \times T(0) + K_1 \times T(1) + K_2 \times T(2) + \ldots\ldots + K_{N-1} \times T(N-1)]$$
$$= \sum_{i=0}^{N-1} K_i \times T(i) \quad (6)$$

where

K_i for i=0,1,2,……,(N-1) are constants given by

$$K_i = N \times \underset{N-2}{\overset{2N-2-i}{C}} \quad \text{for} \quad i=1,2,3,\ldots\ldots\ldots,(N-1) \quad (7)$$

$$K_i = N \times \left[\underset{N-2}{\overset{2N-2-i}{C}} - (N-1) \right] \quad \text{for} \quad i=0 \quad (8)$$

The above formula can be derived by $n1$ as i in equation (2) and than finding the number of possible integer solution of the equation by formula. The result should be multiplied by N as the solution of equation (2) is symmetric around $n_1, n_2, n_3, \ldots\ldots\ldots\ldots, n_N$.

So using equations (4), (5), (6), (7), (8) we get,

$$T(N) = N + \frac{N}{\underset{N-1}{\overset{2N-1}{C}} - N} \left[\sum_{i=0}^{N-1} \underset{N-2}{\overset{2N-2-i}{C}} \times T(i) - (N-1) \times T(0) \right] \quad (9)$$

Making a fair guess that the average time complexity of the algorithm is better than $O(N$*Log(N)) and worse than $O(N)$ we conclude that the graph of $T(N)$ vs N for the given algorithm will be between the plot of N*Log(N) and N. Since the graph of N*Log(N) is fairly linear for large values of N it will be fair to infer that the plot of $T(N)$ which lies between N*Log(N) and N is fairly linear for large values of N. So by One-Point Straight Line formula,

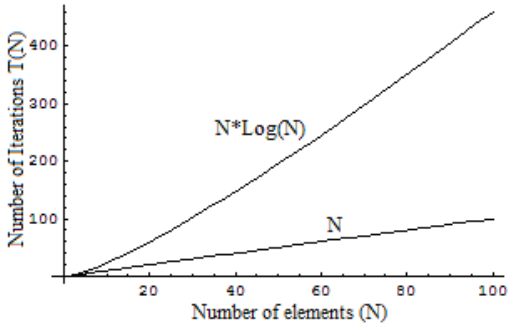


Fig. 1. Graph showing number if iterations needed to sort and array with algorithm of $O(N)$ and O(N*Log(N)) complexity.

$$T(i) = T(N) - (N-i) \times T'(N) \quad (10)$$

where, $T'(N)$ is the derivative of $T(N)$ with respect to N.

Substituting equation (10) in (9) we get,

$$T(N) = N + \frac{N}{\underset{N-1}{\overset{2N-1}{C}} - N} [T(N) \times \sum_{i=0}^{N-1} \underset{N-2}{\overset{2N-2-i}{C}} - T'(N) \times \sum_{i=0}^{N-1} (N-i) \times \underset{N-2}{\overset{2N-2-i}{C}}$$
$$-(N-1)[T(N) - N \times T'(N)]]$$

$$T(N) = N + \frac{N}{\underset{N-1}{\overset{2N-1}{C}} - N} [T(N) \times \sum_{i=0}^{N-1} \underset{N-2}{\overset{2N-2-i}{C}} - T'(N) \times (N-1) \times \sum_{i=0}^{N-1} \underset{N-1}{\overset{2N-2-i}{C}}$$
$$-N \times T(N) + T(N) + N \times (N-1) \times T'(N)]$$

As shown in [6], $\sum_{i=a}^{b} \underset{a}{\overset{i}{C}} = \underset{a+1}{\overset{b+1}{C}}$, hence,

$$T(N) = N + \frac{N}{\underset{N-1}{\overset{2N-1}{C}} - N} [T(N) \times [\underset{N-1}{\overset{2N-1}{C}} - 1] - T'(N) \times (N-1) \times \underset{N}{\overset{2N-1}{C}}$$
$$-N \times T(N) + T(N) + N \times (N-1) \times T'(N)]$$

$$T(N) = N + \frac{N}{\underset{N-1}{\overset{2N-1}{C}} - N} [T(N) \times [\underset{N-1}{\overset{2N-1}{C}} - N] - T'(N) \times (N-1) \times [\underset{N}{\overset{2N-1}{C}} - N]]$$

$$T(N) = N + N \times [T(N) - (N-1) \times T'(N)]$$

$$N \times (N-1) \times T'(N) = N + (N-1) \times T(N)$$

$$T'(N) - \frac{T(N)}{N} = \frac{1}{N-1}$$

Solving the above differential equation gives,

$$T(N) = N \times Log_e[1 - \frac{1}{N}] + \beta \times N$$

where, β is an arbitrary constant.

For large values of N, $N \times Log_e[1-\frac{1}{N}] \approx -1$

So,

$T(N) = \beta \times N - 1$ for large values of N.

So the average time complexity of the algorithm is $O(N)$.

VIII. Results and Discussion

A code implementing the above algorithm has been made and the working of the algorithm is checked. The algorithm works fine. Also the number of iterations to sort an array of given size is counted using the same code. The number of iterations is compared with that of Quick Sort and found to be less most of the time.

TABLE I: TABLE COMAPRING THE AVERAGE NUMBER OF ITERATIONS (AVERAGED OVER 100 RANDOM TEST CASES) OF INTERPOLATION SORT, HEAP SORT AND QUICK SORT.

Number of Elements	Average Number of Iterations		
	Interpolation Sort	Heap Sort	Quick Sort
50	77	269	257
100	164	637	712
150	271	1033	1195
200	380	1468	1677
250	486	1911	2347
300	590	2378	3166
350	698	2838	3478
400	800	3327	4242
450	900	3822	5019
500	1000	4300	5615

A graph is plotted taking size of the array in the X-axis and number of iterations in the Y-axis. A Least Square Fit of the above graph verifies that the algorithm is $O(n)$ time complexity.

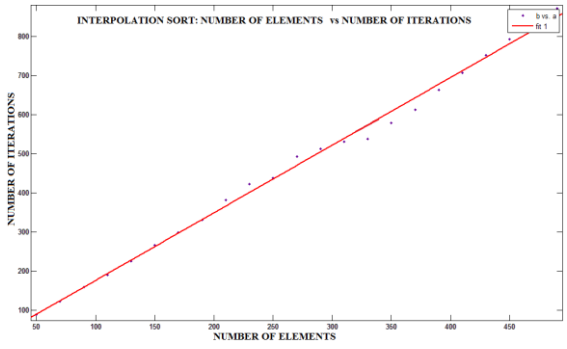


Fig. 2. Graph between number of elements and average number of iterations required to sort an array using interpolation sort. Red line is the Least Square Fit of the data obtained.

It should be noted that the number of iterations required to sort an array using Interpolation Sort depends a lot on the variance of the data present in the array. Lesser the variance, lesser the number of iterations required to sort the array. In other words better the distribution of data (lesser variance), faster will be the sorting.

IX. Conclusion and Future Work

Hence an algorithm has been developed using the principle of interpolation which sorts an array of in $O(n)$ time complexity. Also a way has been proposed to implement the same algorithm for Strings.

An important thing to note about the algorithm is that an arbitrary array size of 3 is chosen after which the array is sorted using bubble sort. But this might not give optimum result always. For small arrays bubble sort will work better while for large arrays interpolation sort works better. Our future work will consist of determining the array size below which bubble sort will give better performance than interpolation sort. To do this we first aim at finding the constants A and B of the following equations:

$$T_{bub}(N) = A \times N^2$$

$$T_{\text{int}}(N) = B \times N$$

where

$T_{bub}(N) \rightarrow$ Average number of Iterations require to sort an array using Bubble Sort

$T_{\text{int}}(N) \rightarrow$ Average number of Iterations required to sort an array using Interpolation Sort

The value N_o we are interested in can be obtained by equating the above equations. i.e.

$$T_{bub}(N_o) = T_{\text{int}}(N_o)$$

$$A \times N_o^{\,2} = B \times N_o$$

$$N_o = \frac{B}{A}$$

References

[1] D. E. Knuth, *The Art of Computer Programming*, 2nd ed., Addison Wesley, ch. 5 and 6, vol. 3, 1998.

[2] G. H. Gonnet, "Interpolation and interpolation hash searching," Ph. D. dissertation, Univ. of Waterloo, Ontario, Canada, 1976.

[3] D. E. Willard, "Searching unindexed and nonuniformly generated files in loglog N time," *SIAM Journal on Computing*, vol. 14, pp. 1013-1029, 1985.

[4] A. C. Yao and F. F. Yao, "The complexity of searching an ordered random table," in *Proc. of 17th Annual Symp on Foundations of Computer Science, Houston, Texas*, 1976, pp. 173-177.

[5] Andersson, Arne, and C. Mattsson, "Dynamic interpolation search in o(log log n) time," *Proc. of the 20th Int Colloquium on Automata, Languages and Programming*, London, UK, 1993, pp. 15-27

[6] S. M. Ross, *A First Course in Probability*, 3rd ed., Macmillan, New York, 1988.