Performance Study of Improved Heap Sort Algorithm and Other Sorting Algorithms on Different Platforms

Vandana Sharma[†], Satwinder Singh[††] and Dr. K. S. Kahlon [†††]

Department of Computer science & Engineering Chitkara Institute of Engg. & Technology, Rajpur a, Punjab , India
Department of Computer science & Engineering, Baba banda Singh Bahadur engineering College, Fatehgarh Sahib, Punjab, India
Department of Computer science & Engineering ,GNDU Amritsar , Punjab, India

Summary

Today there are several efficient algorithms that cope with the popular task of sorting. This paper titled Comparative Performance Study of Improved Heap Sort Algorithm and other sorting Algorithms presents a comparison between classical sorting algorithms and improved heap sort algorithm. To have some experimental data to sustain these comparisons three representative algorithms were chosen (classical Heap sort, quick sort and merge sort). The improved Heap sort algorithm was compared with some experimental data of classical algorithms on two different platforms that lead to final conclusions.

Key words: *Complexity, Performance of algorithms, Asymptotic notation*

1. Introduction

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for producing human-readable output. Sorting algorithms are classified by several other criteria such as Computational complexity (worst, average and best number of comparisons for several typical test cases) in terms of the size of the list, Stability (Memory usage and use of other computer resources), The difference between worst case and average behavior, behaviors on practically important data sets (completely sorted, inversely sorted and almost sorted. In this paper, a comparative performance evaluation of improved heap sort with three different sorting algorithms: heap sort, quick sort, and merge sort is presented. In order to study the interaction between the algorithms and the platform, all the algorithms were implemented on two different platforms**.**

2. Comparison based sorting algorithms

A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator) that determines which of two elements should occur first in the final sorted list. The only requirement is that the operator obey the three defining properties of a total order:

if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry)

if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)

$a \leq b$ or $b \leq a$ (totalness or trichotomy)

Example: Quick sort, Heap sort, Merge sort, Insertion sort, Selection sort Bubble sort

In this paper algorithms based on comparisons are studied. Theoretical lower bound [1] for general sorting algorithms is

$$\log(n!) = n\log n - n\log e + \theta(\log n)$$

$$\approx n \log n - 1.442695n,$$

for the worst case numbers of comparisons. This lower Bound makes sorting by merging, sorting by insertion and binary search very efficient. Merge Sort performs at most nlogn – n+1 key comparisons and requires O(n) extra space Quick sort consumes $\Theta(n^2)$ comparisons in worst case. Hoare proposed CLEVER-QUICKSORT in worst case still it has $\Theta(n^2)$ comparisons and in average case number of comparisons are reduced to 1.188nlogn-2.255n[2]. Heap sort needs 2nlogn comparisons and upper bound for comparisons in Bottom-up-heap sort of 1.5nlogn [3]. Carlson's variant of Heap sort [4] needs nlogn + (nlog logn) comparisons. Wegner showed that it McDiarmid and Reed's variant of Bottom-up-heap sort needs nlogn+1.1 n comparisons [5]. An improved HEAP SORT algorithm is proposed by XiaoDong Wang and Ying -Jie wu with nlogn-0.788928n comparisons in Worst case [6].

Manuscript received April 5, 2008
Manuscript revised April 20, 2008

2.1. Heap sort

Heapsort is a comparison-based sorting algorithm Heapsort is an in-place algorithm, but is not a stable sort. a worst-case **O**(n log n) runtimeHeapsort inserts the input list elements into a heap data structure. The largest value (in a max-heap) or the smallest value (in a min-heap) are extracted until none remain, the values having been extracted in sorted order. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.During extraction, the only space required is that needed to store the heap. In order to achieve constant space overhead, the heap is stored in the part of the input array that has not yet been sorted Heapsort uses two heap operations: *insertion* and *root deletion*. Each extraction places an element in the last empty location of the array. The remaining prefix of the array stores the unsorted elements

2.2. Quick sort

The quick sort is an in-place, divide-and-conquer, massively recursive sort. The quick sort is by far the fastest of the common sorting algorithms Quicksort runs in $O(n \log(n))$ on the average and worst case behavior of $\Theta(n^2)$.

2..3. Merge sort

Mergesort is an $O(n \log n)$ comparison-based sorting algorithm. It is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm. Conceptually, merge sort works as follows:

i) Divide the unsorted list into two sublists of about half the size

ii) Divide each of the two sublists recursively until we have list sizes of length 1, in which case the list itself is returned

iii) Merge the two sublists back into one sorted list.

2.4. Modified Heap Sort

A new variant of Heap Sort is modified heap sort [6] Basic idea of new algorithm is similar to classical Heap sort algorithm but it builds heap in another way. This new algorithm requires nlogn-0.788928n comparisons for worst case and nlogn-n comparisons in average case. This algorithm uses only one comparison at each node. With one comparison we can decide which child of node contains larger element. This child is directly promoted to its parent position In this way algorithm walks down the path until a leaf is reached

Algorithm

Procedure Rank Heap Sort ()

Step I: Building : Call to Build Heap (H)

Step II: Loop for shifting

For i← length(H) down to 2

Call to Shift(i)

Step III: Call to Rearrange()

Fig. 1.1 Rank Heap sort

The above Procedure in fig 1.1 sorts the element of a given array using modified heap sort technique . The procedure makes a call to build heap , shift and rearrange.

Procedure Shift(index)

Step I: Set k←0

Step II: Call to internal(K)

Step III: If K is an internal node

Call to maxchild(k)

Set k ←maxchild(k)

Swap a[k/2] and a[k]

Step IV Repeat step III till K is internal node

Step V Set Rank[k] ← index

Fig 1.2. Shift Procedure

Shift procedure in fig.1.2 shift the root element downward till it becomes leaf node . The procedure makes a call to internal and maxchild procedures

```
Function   boolean internal(i)

Step I:   Set   k ← 2*I

Step II: Return   k<=n and !rank[k] or ( k<n) and(!rank[k+1])
```

Fig 1.3 . Internal Procedure

Internal function in fig 1.3 return whether a node is internal node or not.

```
Function integer maxchild( i )

Step I:              Set   k ←  2 * i

Step II:  Set        left ←  (k<=n) AND  (!rank[k])

                     Set right = (k<n) AND(!rank[k+1])

Step III:    If(!left  OR  left  AND  right  AND a[k]<a[k+1])

                        Set k ← k+1

Step IV:  return k
```

Fig. 1.4 Maxchild Procedure

maxchild function in fig 1.4 returns index of child of node i having maximum value.

```
Procedure Rearrange()

Step I:              For I=2 to n

Step II:  Repeat step III /& IV while rank[I] ← I

Step III             Swap a[i] and a[rank[I]

Step IV              Swap rank[i]and rank[rank[i]
```

Fig 1.5. Rearrange Procedure

Rearrange procedure in fig 1.5 rearranges elements of array in such way that order becomes sorted.

3. Performance study

In the previous section, heap sort, quick sort, merge sort and improved heap sort algorithms were described. A detailed study to assess the performance of improved heap sort algorithm with respect to the heap, quick and merge sort algorithms was conducted . The performance metric in all the experiments is the total execution time taken by the sorting operation.

The experiments were conducted in two categories

Category I:
AMD 1.8 GHz workstation running Windows XP Professional Service Pack2 operating system configured with 512MB main memory and one 80 GB hard disk using Microsoft Visual C++ compiler.

Category II:
Celeron 2.5 GHz running Windows 2000 Professional Service Pack 2 operating system configured with 512 MB main memory and one 40 GB hard disk using Dev C++ compiler.
For the experiments integer numbers have been used, which were generated randomly. To obtain results data files were used. To study the performance of the algorithms generated data sets with 1000 to 100000 items were used and code was executed 50 times and average execution time in clock ticks was recorded and converted in ms.

Results of Category I are shown in Table 1. It shows the execution times of all the four algorithms for no. of data items ranging from 1K to 100K. It is observed that that modified heap sort algorithm takes less time than other sorting algorithms for data items 100K . In case of 100K data items it beats quick sort also as shown in Fig 2. For other data sizes the execution time is greater than other sorts. While for larger no. of data items the execution time difference increases.

TABLE 1
AVERAGE SORTING TIME (IN MSEC) OF ALGORITHMS ON RANDOM DATA AVERAGED 50 RUNS(CATEGORY I)

No.of data item----->	**1000**	**5000**	**10000**	**50000**	**100000**
Heap sort	0.0003	0.0045	0.0054	0.04962	0.08486
Merge sort	0.0009	0.0045	0.0054	0.94564	0.08426
Quick sort	0.0009	0.0021	0.0027	0.02116	0.04734
Modified heap sort	0.0015	0.0066	0.0154	0.07428	0.0162

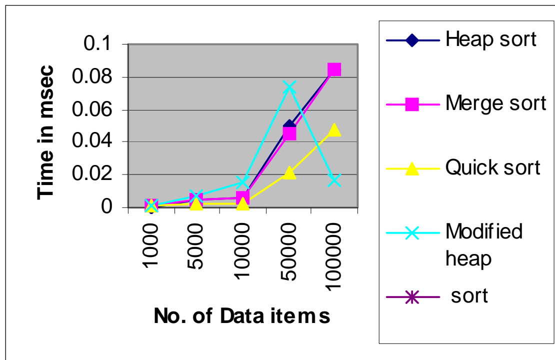

Fig 2 .Comparison of sorting algorithms performance

Results of category II are shown in Table 2. It represents the execution times of all the four algorithms for no. of data items ranging from10 to 100K similar as in category I. It shows poor performance of modified heap sort algorithms as compared to other algorithms for larger data items of 50K and 100K.Fig 3. Represents performance of algorithms in Category II.

TABLE 2
AVERAGE SORTING TIME(IN MS) OF ALGORITHMS ON RANDOM DATA AVERAGED 50 RUNS(CATEGORY II)

No.of data item------>	**1000**	**5000**	**10000**	**50000**	**100000**
Heap sort	0.00122	0.0036	0.0083	0.04994	0.1229
Merge sort	0.00061	0.0012	0.0153	0.05287	0.12308
Quick sort	0.0009	0.0003	0.0027	0.01469	0.0298
Modified heap sort	0.00030	0.0067	0.0092	0.11718	0.29665

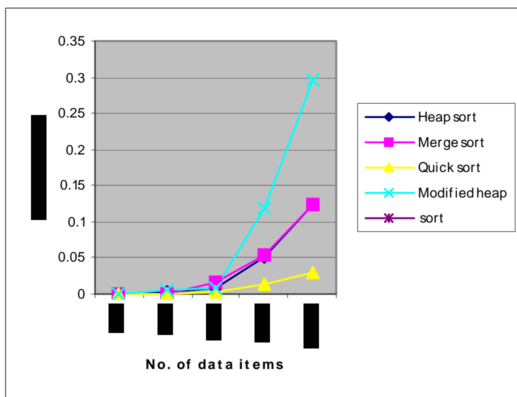

Fig32 : Sorting Algorithms Performance (category II)

4. Conclusions

This paper presented the comparative performance study of four sorting algorithms on different platform. For each machine, It is found that the choice of algorithm depends upon the number of elements to be sorted. In addition, as expected, results show that the relative performance of the algorithms differed on the various machines. In category I for large number of data items performance of improved heap sort is better. But for small number of data items performance of modified heap sort is poor than other algorithms. But in Category II platform for each data size except 1K ,modified heap sort algorithm takes more execution time than any other algorithm in question.

5. Future work

Considering the performance of algorithms based on hardware and operating system better analysis can lead to more efficient algorithm. In future purposed work is to improve algorithm considering these factors.

References

[1] Knuth D.E *The art of programming-sorting and searching*.2nd edition addison wesley.

[2] Hoare C A R Quicksort.Computer journal 5(1):10~15.

[3] I.Wegner:*BOTTOM-UP-HEAPSORT beating on average QUICKSORT(if n is not very small).* Proceedings of the MFCS90,LNCS 452,516-522,1990

[4] S.Carlsson:*Avariant of HEAPSORT with almost optimal number of comparisons*.Information Processing Letters 24:247-250,1987.

[5] I.Wegner:*The worst case complexity of Mc diarmid and Reed's variant of BOTTOM-UP-HEAP SORT is less than nlogn+1*.1n. Proceedings of the STACS91,LNCS 480:137-147,1991.

[6] Xio dong wang,ying jie wu *An improved heap sort algorithm with nlogn –0.788928n comparisons in worst case.* journal of computer science and technology22(6):898~903

Vandana Sharma is a Senior Lecturer in Department of Computer Science & Engineering at Chitkara Institute of Engineering and Technology, Jansla , Punjab, India. Her research interests are in design and analysis of algorithms, datamining and applications of IT. Ms. Vandana is member of ISTE.

Satwinder Singh is Lecturer in Department of Computer Science & Engineering and IT at Baba Banda Singh Bahadur Engineering College, Fatehgarh Sahib,Punjab, India. .His research interests are design and analysis of algorithms, object oriented analysis & design.

Dr. K.S. Kahlon is Professor &Head of Department of Computer Science and Engineering, Guru Nanak dev University, Amritsar (India)