ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

STUDY & ANALYSIS OF EXTERNAL SORTING ALGORITHM

Vaishnavi Bihare*, Vishakha Wadhwani, Yamini Rathore, Yash Kumar Jain

ABSTRACT

If we look at various applications, most of them, especially the database involved huge amount of data. For example consider a payroll system of any multinational organization; every department is having many employees. Now if we want to see the salary of any employee, then it is very difficult for us to identify the employee salary record. But if the data is systematically arranged using the employee Id than it will be very much convenient for us to locate the record of the record. In this paper, we will focus and analyzed the external sorting algorithm and their differences.

KEYWORDS: sorting, external, merge sort, quick sort, heap sort.

INTRODUCTION

Algorithm is nothing but the step by step execution of the programs. The steps or the instructions of any program in term of algorithm must be finite. Sorting is the systematic arrangement on the basis of key which help us to fetch the data in quick time. Consider a telephone directory, in that the entire phone numbers are systematically arranged on the basis of key, that key is nothing but the surname of the person.

Commercial computing: *Every organizations, institute and companies actually maintain the huge amount of data of their employees and students. The systematic arrangements make it more convenient to access the any of the record. All these organizations mainly store all the information like name, id, address, contact number and many more. Processing on such data will definitely require an arrangement that arrangement is nothing but the sorting.*

Operations research: Suppose we have some jobs to be available. We need to execute all the jobs and improve the employee satisfaction. Consider an example of priority queue. In priority queue all the jobs will be submitted for execution on the basis of their priorities. So on the basis of that first we need to arrange the data either is ascending order or in descending order and then process the job.

Numerical computations: The accuracy becomes one of the crucial parameter in scientific calculations. It is very important when we are performing billions of computations with projected values such as the integer number, floating point numbers, characters etc. that we commonly use on computers. Here in this also we will use sorting to maintain the accuracy in calculations.

PROCEDURES OF ALGORITHMS

External sorting is a sorting which actually applies on the hard disk. That means that if the data is large enough than we can use external sorting to make the proper and systematic arrangement of data. In external sorting algorithm some of sorting algorithm is commonly used. In this section, we will discuss the procedures of external sorting algorithms.

Merge sort:

The merge sort is the sorting technique that works on the principle of divide and conquers strategy. In this approach, first we will divide the whole dataset into number of partitions and then we will individually sort those partitions and combine them.

Working of Algorithm:

The algorithm is work in two phase. In the first part, we will split the whole array in the number of partition to calculate the mid and in the second phase, we will sort and combine them to get the actually sorting arrangement.

Algorithm:

```
Merge_sort(Array A, Low, High)
{
        If(Low<High)
        {
                Mid=(Low+High)/2;
                Merge_sort(A, Low, Mid);
                Merge_sort(A, Mid+1, High);
                Combine(A, Low, Mid, High);
        }
}
Combine(A, Low, Mid, High)
{
        K=i=Low;
        J=Mid+1;
        While(i<=Mid AND j<=high)
        {
                If(A[i]< A[j])
                {
                        Temp[k++]=A[i++];
                }
                else
                {
                        Temp[k++]=A[j++];
                }
        }
        While(i<=mid)
        {
                Temp[k++]=A[i++]
        }
        While(j<=high)
        {
                Temp[k++]=A[j++];
        }
}
```

Quick Sort

Quick sort is another sorting algorithm which follows the principal called divide and conquer. In this approach, we will divide the whole dataset into two partitions with the help of Pivot Element. Pivot can be any one element in the given list. All the elements which are smaller than the pivot elements will make the left sub list and all elements which are larger than pivot element will make the right sub list. We will divide the whole list in this manner and then we will sort the individual sub list and then combine to get the proper arrangement.

Algorithm:

```
void quickSort(int arr[], int left, int right) {
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];
     /* partition */
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
```

```
            arr[j] = tmp;
            i++;
            j--;
        }
    };
     /* recursion */
    if (left < j)
        quickSort(arr, left, j);
    if (i < right)
        quickSort(arr, i, right);
```

Heap Sort

Heap sort is a comparison-based sorting algorithm to create a sorted array (or list), and is part of the Selection sort family. Although somewhat slower in practice on most machines than a well-implemented Quick sort, it has the advantage of a more favorable worst-case O(*n* log *n*) runtime. Heap sort is an in-place algorithm, but it is not a stable sort. The heap sort algorithm can be divided into two parts. In the first step, a heap is built out of the data. In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one. Heap sort can be performed in place. The array can be split into two parts, the sorted array and the heap. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

Algorithm:

```
function heapSort(a, count) is
    input:  an unordered array a of length count
     (first place a in max-heap order)
    heapify(a, count)
     end := count-1
   /* In languages with zero-based arrays the children are 2*i+1 and 2*i+2 */
    while end > 0 do
       (swap the root(maximum value) of the heap with the last element of the heap)
       swap(a[end], a[0])
       (decrease the size of the heap by one so that the previous max value will
       stay in its proper placement)
       end := end - 1
       (put the heap back in max-heap order)
       siftDown(a, 0, end)
function heapify(a, count) is
    (start is assigned the index in a of the last parent node)
    start := (count - 1) / 2
       while start ≥ 0 do
       (sift down the node at index start to the proper place such that all nodes
        below the start index are in heap order)
       siftDown(a, start, count-1)
       start := start - 1
    (after sifting down the root all nodes/elements are in heap order)
  function siftDown(a, start, end) is
    input:  end represents the limit of how far down the heap
              to sift.
    root := start
 while root * 2 + 1 ≤ end do          (While the root has at least one child)
       child := root * 2 + 1       (root*2 + 1 points to the left child)
       swap := root       (keeps track of child to swap with)
       (check if root is smaller than left child)
       if a[swap] < a[child]
          swap := child
```

```
(check if right child exists, and if it's bigger than what we're
 currently swapping with)
if child+1 ≤ end and a[swap] < a[child+1]
   swap := child + 1
(check if we need to swap at all)
if swap != root
   swap(a[root], a[swap])
   root := swap          (repeat to continue sifting down the child now)
else
   return
```

ANALYSIS OF ALGORITHM

Merge sort is an efficient sorting method when compared to simple sorting methods and also some sophisticated methods such as heap sort and quick sort. The worst case and average case running time complexity of merge sort is O(n log n). However it considered not efficient for the reason that it requires twice the number of additional memory for the second array than any other sophisticated algorithms. Merge sort use a separate array to store the entire sub array along with the main array. Generally it is an external sorting algorithm which needs O(n) additional memory space for n elements. This fact makes merge sort inefficient for the application that run on machines with limited memory resource. Therefore, merge sort recommends for large data sets with external memory.

One disadvantage of merge sort is it requires twice as much additional memory than other sophisticated sorting algorithms and likewise it is not recommended for smaller arrays for the reason that it works recursively and it required O(n) auxiliary space for sorting. In addition, it is difficult to implement the merge operation. However merge sort is considered for the inputs with their data elements stored in a linked list, because merging does not require random access to the list elements. Even though it does not know the length of the list in the beginning in some cases, it requires a single scan through the list to figure it out.

Quick sort is the fastest sort on the average running time complexity of O(n log n) when compared to other sophisticated algorithms. Usually, selecting the leftmost or rightmost element as a pivot causes the worst case running time of $O(n^2)$ when the array is already sorted. Likewise, it is not efficient if all the input elements are equal, the algorithm will take quadratic time $O(n^2)$ to sort an array of equal elements. However, these worst case scenarios are infrequent. There are more advanced version of quick sort are evolved with a solution to selecting pivot. Qsort [6] is one of the variants of quick sort, which is faster and more robust than the standard method. Quick sort is the better option if speed is greatly significant and also for large data sets. It is quite complicated to sort an array of smaller size, so we can implement a quick sort often with insertion sort to sort smaller arrays.

One of the disadvantages of heap sort is that it works slower than other sorting methods with same computational complexity. Moreover, it is not efficient for parallelization. However, heap sort is considered often for large data sets for the reason that it does not work recursively all the time. Sorting data stored in linked list data structure, heap sort is not recommended due to the fact that it's difficult to convert the linked list to heap structure.

No.of Data Elements	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	0.0028	0.0018	0.0084	0.0104
2000	0.004	0.0034	0.0072	0.0076
3000	0.0066	0.0058	0.013	0.018
5000	0.011	0.01	0.21	0.0376
10000	0.0254	0.0214	0.041	0.0768
20000	0.0534	0.0456	0.0884	0.1652
30000	0.0842	0.0692	0.1332	0.2632
50000	0.148	0.121	0.2364	0.454
100000	0.3228	0.257	0.5086	0.9974
200000	0.6782	0.5496	1.3014	2.27
300000	1.0566	0.85	1.944	3.3462

CONCLUSION

This paper presented the survey and comparison of different sorting algorithms along with the results of practical performance. The main findings of this study are that three factors such as running time, number of swaps and memory used that are critical to efficiency of sorting algorithms. Experiment results clearly show that the theoretical performance behavior is relevant to the practical performance of each sorting algorithm. However there is a slight difference in the relative performance of algorithms in some test cases. In addition, the experiment results proved that algorithm's behavior will change according to the size of elements to be sorted and type of input elements. Which conclude that each algorithm has its own advantage and disadvantage? In this study, three different types of performance behavior investigated such as O(n2) class and O(n log n) class and non-comparison or linear class. Each sorting class outperform the other class under certain conditions such as non-comparison based sorting algorithms outperforms comparison based sorting algorithm in large data-sets and O(n log n) class algorithm outperforms the O(n2) class algorithms in large data sets. With the evolution of new technology and the increased usage of the Internet, the data available on the Internet also increased. This creates the demand for fast and efficient sorting algorithm. However, this does not reduce the need of simple sorting algorithms in some cases due to the fact that many sophisticated algorithms use simple sorting algorithms such as insertion sort and shell sort despite of its quadratic performance [16]. As a result, there are more than two algorithms are used to solve a problem as a hybrid method. Thus, selecting efficient algorithm depends on t

REFERENCES

[1] Heap sort. http://info.mcip.ro/?t=ts&p=7, March 2014.
[2] Sorting in linear time. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap09.htm, March 2014.
[3] Sorting: Runestone interactive. http://interactivepython.org/runestone/static/pythonds/SortSearch/sorting.html, March 2014.
[4] Arne Andersson and Stefan Nilsson. A new efficient radix sort. In FOCS, pages 714–721. IEEE Computer Society, 1994.
[5] Kenneth E. Batcher. Sorting networks and their applications. Spring Joint Computer Conference, AFIPS Proc, 32:307–314, 1968.
[6] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. Softw., Pract. Exper., 23(11):1249–1265, 1993.
[7] Coenraad Bron. Merge sort algorithm [m1] (algorithm 426). Commun. ACM, 15(5):357–358, 1972.
[8] M.S. Garai Canaan.C and M. Daya. Popular sorting algorithms. World Applied Programming, 1:62–71, April 2011.
[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2009.
[10] Ian J. Davis. A fast radix sort. Comput. J., 35(6):636–642, 1992.
[11] E.H.Friend. Sorting on electronic computers. JACM 3(2), pages 34–168, 1956.
[12] Donald E.Knuth. The Art of Computer Programming Second Edition, volume 3. ADDISON-WESLEY, 1998.
[13] C.A.R. Hoare. Quicksort. Commun.ACM, 4:321, 1961.
[14] Daniel Jimenez. Complexity analysis. http://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture2.html, June 1998.
[15] Omar khan Durrani, Shreelakshmi V, Sushma Shetty, and Vinutha D C. Analysis and determination of asymptotic behavior range for popular sorting algorithms. Special Issue of International Journal of Computer Science Informatics(IJCSI), ISSN(PRINT):2231-5292, Vol-2, Issue-1,2.