International Journal of Scientific & Engineering Research, Volume 4, Issue 6, June-2013
ISSN 2229-5518

DAMS SORT

Sudeep Bisht[1]

[1] Department of Computer Science, Maharaja Surajmal Institute of Technology, Janakpuri, N.delhi 110058, India
Email:sudeep.28.march@gmail.com

ABSTRACT

Sorting is a fundamental data structure which can find its applications in all our daily life. It is considered to be the most fundamental problem in study of algorithms. A sorting algorithm can be defined as an algorithm that arranges data in a sequential order. There are many sorts, some of which are bubble sort, cocktail sort, insertion sort, selection sort, quick sort, radix sort etc. Sorting finds its use in many operations such as cpu job scheduling, integer sorting etc.

In this paper the bubble sort is modified and has been presented with a new approach. The paper puts forth a modified algorithm called Dams algorithm that combines the technique of bubble sort with partitioning and modified diminishing sort. The results obtained by implementing this sort and comparing with bubble sort and cocktail sort showed that the proposed algorithm performed better than the two mentioned above. Hence a conclusion has been reached through experimental result that the proposed algorithm is better than the bubble and its bi-directional sort.

Key Words: Algorithm, Bubble Sort, Cocktail Sort, Bi directional Bubble Sort, Dams Sort, Internal Sort, External sort

INTRODUCTION

Any program is written and implemented only after its algorithm is chalked down. An algorithm is a well defined computational procedure that transforms the input to output **[1]**. It can also be described as a step by step representation of a problem.

Donald Knuth listed 5 properties that differentiate an algorithm:

- Finiteness: An algorithm must always terminate after a finite number of steps. Similar procedures which differ only in that they do not terminate can be described as computational methods.
- Definiteness: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously 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.
- Output: An algorithm has one or more outputs; quantities that have a specified relation to the 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.**[2]**

Sorting finds its widespread use in job search, web search, telephone companies, libraries etc. Sorting can be classified into two parts:

a) Internal sorting b)External sorting

When the list is to be sorted, the list resides completely in main memory then it is internal sort (e.g. bubble sort, selection etc) but if some part of list is in main memory and other part is in secondary memory, then it's called external sort(e.g. two way sort, polyphase sort etc)**[3]**.
In internal sort for sorting larger databases it may be necessary to hold only a chunk of data in memory at a time, since it won't all fit. The rest of the data is normally held on some larger, but slower medium, like a hard-disk. Any reading or writing of data to and from this slower media can slow the sort process considerably. **[4]**

Bubble Sort

Bubble sort is an internal sort in which each pair of adjacent data item are compared and if they are found to be in an incorrect order are swapped, this process continued repeatedly. The pass through the list are repeated until no more swaps are required. In this sort (if sorting in ascending order) the largest element bubbles to the top in

the first pass then in the second pass the second largest element bubbles to its position.

The sorting algorithm is measured in terms of the number of comparisons. There are **n-1 comparisons** for an array of n elements during 1st pass which places largest element in last position; there are **n-2 comparisons** in second step which places second largest element in the second last position and so on.
Thus

$$F(n)=(n-1) + (n-2) + \ldots.. + 2 + 1$$
$$=n(n-1)/2 \quad \textbf{[5]}$$

Time complexity of bubble sort algorithm is $O(n^2)$

Remark: Some programs use 1 bit variable flag to signal that no interchange takes place during pass. For flag=0 after any pass, list is sorted and there is no need to continue. The number of passes is cut down, but it's efficient when list is originally 'almost' in sorted order. **[6]**

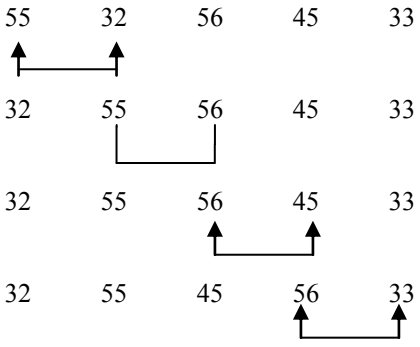
Example

To understand how a bubble let us consider an array containing elements that is to be sorted.
Starting from the first element the adjacent pair of elements are checked and if they are not in a correct order then they are swapped. In this sort in the first pass (if arranging array in ascending order) the largest element is placed in the last position and then in the second pass the second largest element is placed in its position and so on in the subsequent passes. For example let us consider an array below:

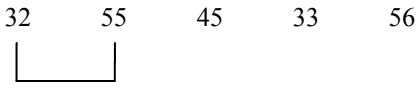
55 32 56 45 33

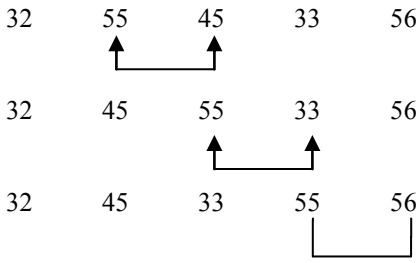
A pictorial representation of implementation of bubble sort on this array is shown in Figure 1

Pass 1

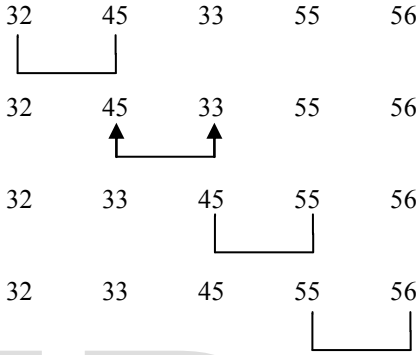


Pass 2

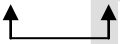




Pass 3



: Represent a comparison between numbers with no swap



: Represent a comparison between numbers then a swap

Figure 1: Illustration of bubble sort

Algorithm (Bubble sort)[7]

```
func bubblesort ( var a as array )
        for i = 1:n,
   swapped = false
   for j = n:i+1,
      if a[j] < a[j-1],
         swap a[j,j-1]
         swapped = true
   → invariant: a[1..i] in final position
   break if not swapped
end
```

Cocktail Sort

One of the variation of bubble sort is bi-directional sort also known as Cocktail sort or Shaker sort. This sorting technique is a stable sort. Cocktail sort is different from bubble sort for it sorts the list in both directions. **[8]** In this sort the largest element is bubbled to its position in the first pass after that the smallest value is bubbled to its position in the second pass. In the next pass the second largest element is bubbled to its position i.e. next – to – last. The

process is repeated until no more swaps are made in the last pass or if n passes are completed.

Algorithm(Cocktail Sort)[8]

```
procedure cocktailSort(A: list of sortable items)
defined as:
do
        swapped := false
  for each i in 0 to length (A) – 2 do:
        if A[i] > A[i+1] then
                swap(A[i], A[i+1])
                swapped :=true
        end if
   end for
   if swapped = false then
        break do-while loop
   end if
   swapped := false
   for each i in length (A) – 2 to 0 do:
        if A[i] > A[i+1] then
                swap( A[i],A[i+1])
                swapped := true
         end if
        end for
  while swapped
  end procedure
```

The Proposed Algorithm

This algorithm (**Dams algorithm**) does not follow the traditional rule of bubble algorithm according to which at each pass the largest element bubbles to its fixed position. Instead in this algorithm the element selected bubbles to its position. The rest of the elements are positioned in the array with respect to the bubbled number.
The proposed algorithm is stated as follows:

a. The initial array (arr[]) is divided into sub arrays just like shell sort **but** here the first element is compared with the last element.
b. If the elements are not found to be in their fixed position they are swapped.
c. Next the second element is compared with the second last element and they are swapped if they are not found to be in the proper order.
d. This comparison and swapping process continues until we have compared the last two elements (i.e. the middle elements) or we are only left with one element that is the middle element.
e. Call Sort(arr[], low, high)

Sort (arr[], low, high)
Here
arr[] represents array to be sorted
b_elepos represents position at which the element exists that is to be bubbled.
high represents the pointer or integer pointing to the last position of arr[]
low represents the pointer pointing to the first position of arr[]
b_element represents the element that is bubbled

1. [Initialization]
   Hi <-high
   b_elepos <- low
2. The value pointed by Hi is compared with b_element. If arr[Hi] > b_element then the pointer Hi is decremented
3. This process continues until Hi points to an element less than or equal to the b_element.
4. The b_element is compared to its next adjacent element if Hi > b_elepos (assigning name **next**). Here 2 cases arise when the comparison is made.
   7.1) if (b_element >= next)
      7.1.1) swap(b_element, next)
   7.2) if(b_element< next)
      7.1.1) temp <- arr[Hi]
      7.1.2) arr[Hi] <- next
      7.1.2) next <- b_element
      7.1.3) b_element<-temp
   In the above step a temp is assigned arr[Hi], then arr[Hi] is assigned value of next.
   Further b_element is placed at the position of next and temp is placed in position of b_element.
5. b_elepos is incremented by 1.
6. The steps 6-8 are repeated until b_elepos < Hi.
7. A recursive call is made as:
   7.1. dams (arr[], low, b_elepos-1)
   7.2. dams (arr[], b_elepos+1,high)
   The recursive call is made if low<high

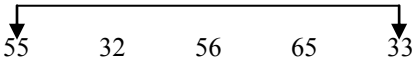
[END]

Figure 2 shows how the sort works

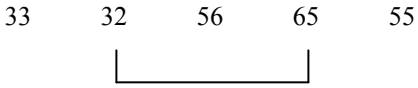
EXAPMPLE
To understand this algorithm let us sort the following array using dams sort.

55 32 56 45 33

A pictorial representation of dams sort is shown in Figure 2
1st Pass



2nd Pass

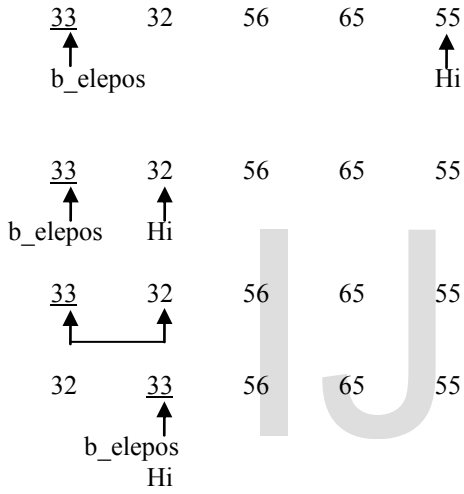


3rd Pass

After the diminishing sort Dams function is called:
Dams(arr[],0,4)

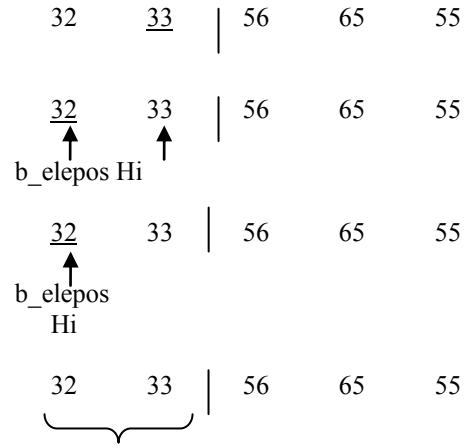
Inside Dams Sort function is called

Sort(arr[],0,4)



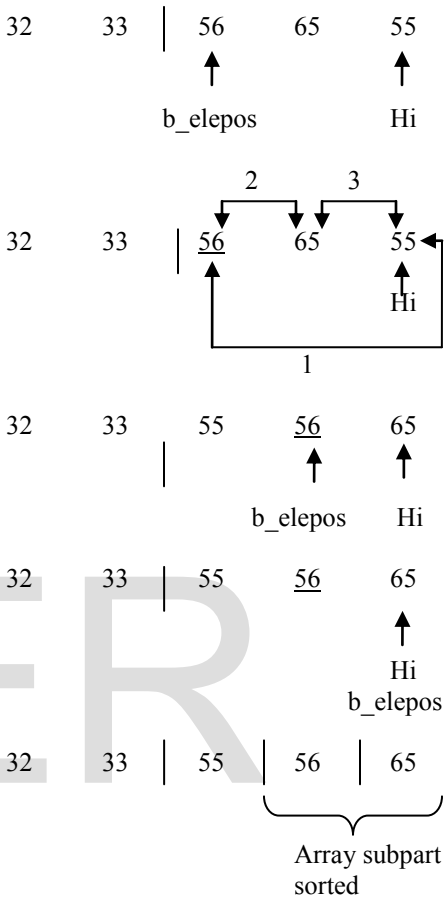
After b_elepos >= Hi Sort gives control to Dams and the function recursively calls itself

Dams(arr[],0,1)
Sort(arr[],0,1)



Array subpart
sorted

Dams(arr[],2,4)
Sort(arr[],2,4)



Dams() recursive call stopped as in all the arrays further formed will have
low >= high

Sorted array:

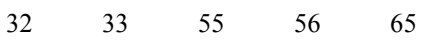


Figure 2: Illustration of Dams Sort

Algorithm

Dams sort (array, low, high)

1. n <- high-low
2. while(i<=(n-1)/2)
3. if(array[i]>array[n-i-1])
4. Swap(a[i],a[n-i-1])
5. i++

6. If array sorted exit.

7. If(low<high)
8. X <- Sort(a, low, high)
9. Dams sort(a,low,x-1)
10. Dams sort(a,x+1, max)

[Sort]
Sort (array, low, high)

1. b_elepos <- low
2. Hi <- high
3. while(b_elepos <Hi)
4. while(a[Hi]>array[b_elepos])
5. Hi—

6. If(b_elepos <Hi)
7. If(array[b_elepos] < array[b_elepos+1])
8. temp=a[Hi]
9. array[Hi]=array[b_elepos+1]
10. array[b_elepos+1]=array[b_elepos]
11. array[b_elepos]=temp
12. If(array[b_elepos]>=array[b_elepos+1])
13. Swap (array(b_elepos),array(b_elepos+1))
14. Return b_elepos

Algorithm Analysis

Algorithms vary in process and output data. Thus there is an appreciable difference in terms of performance and space utilization. Performance of an algorithm is thus based on the time and memory requirement **[10]**. One of the method employed in algorithm analysis is – Experimental analysis **[11]**.

Mostly the worst case and average case are analyzed because:

1. It gives upper bound on running time for any input.
2. For some algorithms worst case occur fairly often.
3. The average case is often as bad as the worst case.

CONCLUSION

Table 1, 2, 3and 4 with the graphs (1, 2, 3 and 4) shown below shows the results are obtained. From the results in Table 1, 2,3and 4 the number of comparison and swaps are less than the Bubble and Cocktail sort. It can also be seen that as the size of array increases the number of swaps and comparisons decreases. Moreover the proposed sort is effective in sorting small as well as large data. Thus it can be concluded that the proposed algorithm is much more efficient that the rest of the two.

REFERENCES

1. (Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. Introduction to Algorithms [1])
2. Donald Knuth. [The Art of Computer Programming, Volume 3: Sorting and Searching]
3. [Data Structures- Deepak Gupta(2012 edition)]
4. [http://en.wikipedia.org/wiki/Internal_sort].
5. [Seymour Lipschutz – Schaum's outlines (Data Structured)(Indian Adapted Edition 2006 edition)]
6. [Seymour Lipschutz – Schaum's outlines (Data Structured)(Indian Adapted Edition 2006 edition)]
7. [http://www.sorting-algorithms.com/bubble-sort]
8. http://en.wikipedia.org/wiki/Cocktail_sort
9. [Knuth, The Art of Computer Programming: Sorting and Searching, 2 ed., vol. 3.Addison-Wesley, 1998.]
10. [Data Structures- Deepak Gupta(2012 edition)]
11. (Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. Introduction to Algorithms [1])

Experimental Analysis

WORST CASE ANALYSIS*

1. Number of swaps

Array Size	Bubble Sort	Cocktail Sort	Dams Sort
100	4950	2500	50
200	19900	10000	100
300	44850	22500	150
400	79800	40000	200

Table 1

2. Number of comparisons

Array Size	Bubble Sort	Cocktail Sort	Dams Sort
100	4950	3774	149
200	19900	15049	299
300	44850	33824	449
400	79800	60099	599

*analysis done on bubble and cocktail's worst case

Table 2

AVERAGE CASE ANALYSIS

1. Number of swaps

Array Size	Bubble Sort	Cocktail Sort	Dams Sort
100	2529	1281	391
200	8744	4485	990
300	22704	10782	1652
400	39659	19399	2465

Table 3

2. Number of comparisons

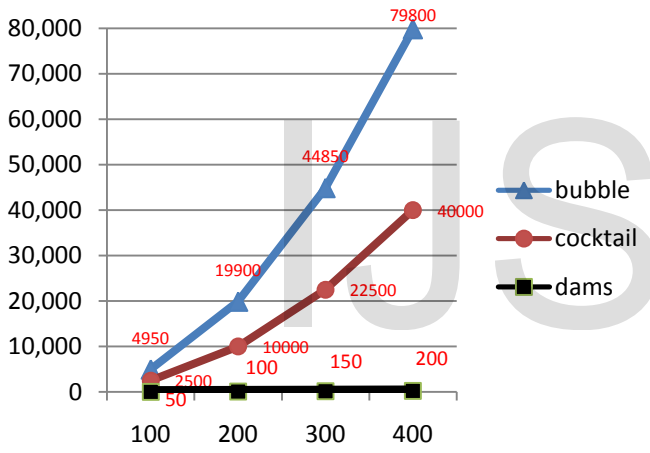
Array Size	Bubble Sort	Cocktail Sort	Dams Sort
100	4929	2535	615
200	19435	7810	1448
300	44525	20540	2304
400	79647	37022	3106

Table 4

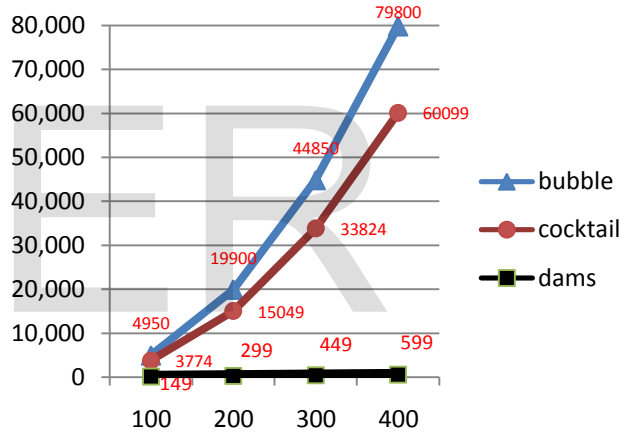
GRAPHS

WORST CASE ANALYSIS

Graph 1: Number of swaps

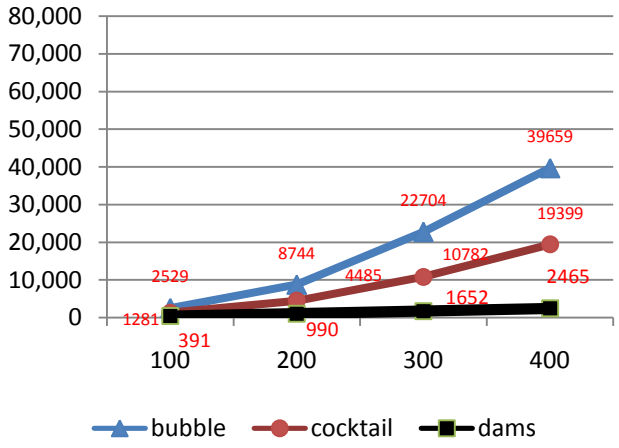


Graph 2: Number of comparisons



AVERAGE CASE ANALYSIS

Graph 3: Number of swaps



Graph 4: Number of comparisons