Sliding Window Sort for Multidimension Array

Krishanu Majumder [a], Dipankar Das [a]

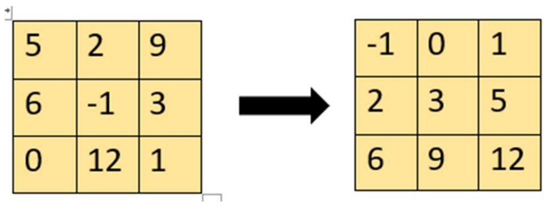
[a] *Dept of Computer Science & Engineering, Jadavpur University, Kolkata, India*

Abstract
Sorting numbers in an array are one of the most basic problems that we learn to solve and have always proved useful. Though there are many algorithms for sorting in 1 dimensional array of numbers, there are not many algorithms which can be applied to 2 dimension or higher dimension arrays. In this paper, we will discuss a new algorithm for sorting 2-dimensional array and will see how it can be modified to sort n-dimensional array.

Keywords
Sorting, Multidimensional array

1. Introduction

Sorting have been one of the oldest solved problems but conventionally limited to 1 dimensional array of numbers. But with growing complexities with matrices, tensors and multidimensional arrays, we might face situations where we need to sort a collection of numbers which are in form of a n-dimensional matrix. We normally do it by converting the multidimensional array into 1 dimensional array, apply the known sorting algorithms to it and restore the array into its previous form. But the question comes can we do it without transforming the array into a 1-dimensional array with a time complexity which will not be worse than the worst time complexities of the known algorithms? Yes, we definitely can. We will see how we can sort a 2D matrix and then we will extend the algorithm to multidimensional array.



$$\begin{array}{|c|c|c|}\hline 5 & 2 & 9 \\ \hline 6 & -1 & 3 \\ \hline 0 & 12 & 1 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline 2 & 3 & 5 \\ \hline 6 & 9 & 12 \\ \hline \end{array}$$

Figure 1: Starting state on the left and target state on the right.

2. Related Works

Sorting is one of the very first problem that we learn. There are many sorting algorithms proposed and some of the widely accepted ones include bubble sort, selection sort, insertion sort, quick sort [1], merge sort [2] to name a few. These algorithms are fit for linear collection of data but mostly not for complex data structures like tensors or multidimensional arrays. Among this, bubble sort, insertion sort and selection sort have the overall time complexity of O(n^2). Quick sort has an average case time complexity as ϴ(nlogn) but have a worst time complexity of O(n^2). On the other hand, merge sort has both average and worst time complexity of O(nlogn) but it uses extra space with worst case space complexity as O(n). Many authors like Oyelami et. al. [3,4] have proposed modifications over the existing algorithms by still they are confined to one-dimension arrays only.

Algorithms, Computing and Mathematics Conference, August 19 – 20, 2021, Chennai, India.
EMAIL: majumder.krishanu95@gmail.com (Krishanu Majumder)
ORCID: 0000-0002-6537-0403 (Krishanu Majumder)

But very few works have been done in the field of multidimensional arrays. One of the interesting works have been presented by Sandeep Sen et. al. [5,6]. They proposed a new algorithm called shear sort which sorts a 2-D array by alternatingly sorting the rows and columns. The authors used it to sort networks. Other works on 2-dimensional sorting was done in the field of VLSI and networks by Thompson and Kung [7], Nissimi and Sahni [8], Hans-Warner et. al. [11] which mainly uses bitonic sort [9] or odd even merge sort [10] using parallel computing. Douglas [12] tried to implement bubble sort in 2-dimension array. Hanan Samet [13,14] discussed the applications of multidimensional sorting in real world like in graphics, game theory, etc.

3. Algorithm

We take a window of 2x2 for a 2-dimensional matrix of size m x n. So, the window will have the elements (i,j), (i,j+1), (i+1,j) and (i+1,j+1) for all 0<=i<m and 0<=j<n. We start from the top left corner. In doing so, we have four numbers under the window. We sort the four numbers and then rearrange the four numbers within the window such that the smallest number is on the top left of the window, next smallest number on the top right, next on the bottom left and the largest number on the bottom right of the window. Now we slide the window one place to the right along the row and will repeat the process until the we reach the end of the row, i.e., top right cell of the window is the last cell along the row. We repeat this for all the rows. This total process forms one pass. We again start form the top left corner of the new matrix until we have done max(m,n) passes. There will be some corner cases like when the window moves to the end of a row or column, i+1 (or j+1) will be out of index range. For simplification, we use mod number_of_rows and mod number_of_columns to locate the elements under the window. A depiction of the process is shown in Fig. 2 and Fig. 3.

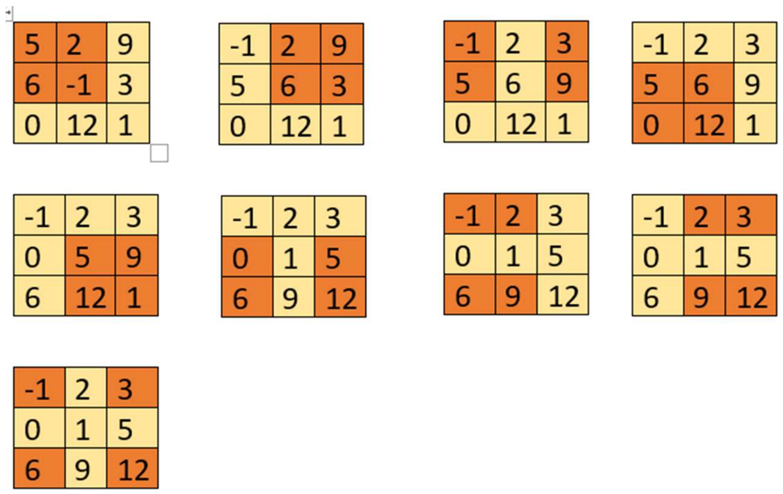


Figure 2: First pass of the algorithm.

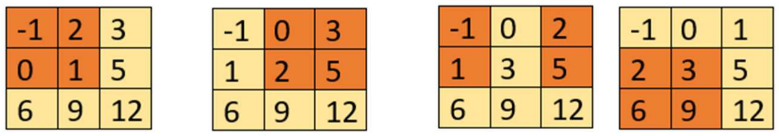


Figure 3: Second pass of the algorithm.

4. Pseudo Code

```
function window_sort(array[][], r, c):
//array[][] is the given array
// r is the number of rows
// c is the number of columns
  flag := 0
  for x := 0 to max(r,c) do:
    for i := 0 to r-1 do:
      for j := 0 to c-1 do:

        k := (i+1) % r
        l := (j+1) % c
        a := min(i,k)
        b := max(i,k)
        c := min(j,l)
        d := max(j,l)

        window := [array[i][j], array[i][l], array[k][j], array[k][l]]
        window := sort(window)

        if array[a][c] ≠ window[0] or array[a][d] ≠ window[1] or array[b][c] ≠ window[2] or
array[b][d] ≠ window[3] and flag = 0 do:
          flag := 1
        end if

        array[a][c] := window[0]
        array[a][d] := window[1]
        array[b][c] := window[2]
        array[b][d] := window[3]

      end for
    end for

    if flag = 0 do:
      break loop
    else do:
      flag = 0
    end if

  end for
end function
```

5. Time and Space Complexity Analysis

When sorting the numbers in the window, we are every time sorting four number and place them at their respective positions in the window. It will take O(1) time. At every pass, we are sliding the window m*n times and this takes O(m*n). We are repeating the above steps for getting the array sorted and it will take a maximum of max(m,n) passes since a misplaced element needs to be moved a maximum of this distance to be placed in its right place. We may even stop the execution if there is no rearrangement during a pass by using a flag variable. Thus, the total time complexity in the worst case is O(m*n*max(m,n)). This can also be written as O((max(m,n))3) in the worst case. Since, we did not use any additional space, space complexity is O(1).

6. Extension to N-Dimension Arrays

Let us extend this to 3-D array first with dimensions d1, d2 and d3. We take the window size as 2 x 2 x 2. The window will be in a shape of smaller cube where the 3-D array will be a bigger cube. The total number of elements under the window to be sorted first is 8. For now we can take the sorting time of the elements under the window be constant, i.e., O(1) as the number of elements under the window to be sorted is constant every time. Now, we move the smaller window cube over the whole array cube such that the (0,0,0) element of the smaller window cube coincides will all (i,j,k) of the bigger array for all 0<=i<d1, 0<=j<d2, 0<=k<d3. Thus, the time complexity of each pass will be O(d1 x d2 x d3). We have to do max(d1, d2, d3) passes as this will be the longest distance that a misplaced element have to be moved to its correct place. So, the total time complexity will be O(d1 x d2 x d3 x max(d1, d2, d3)). But as the dimension increases, the size of the window increases and hence the number of elements to be sorted each time. If we have n dimensions, the window size will be 2n and hence, we need to sort 2n elements each time.

Let us generalize for n-D array, we can choose the window size as 2 x 2 x 2 x… upto n times. Here, the total number of elements in the window will be 2^n. Now, we can use the best existing algorithm for sorting the elements in the window and getting it to their proper positions within the window which will take O(n x logn) for n elements. In this case, this will take in the worst case, we have 2^n elements, hence

$O(2^n \text{ x } \log 2^n)$
or, $O(2^n \text{ x } n \text{ x } \log 2)$
or, $O(2^n \text{ x } n)$

Then we can follow the above explained procedure to first sort within the window and then slide the window. If di, for i from 1 to n, are dimension sizes of the n dimensions, then the total worse case time complexity of the given algorithm is

$O(2^n \text{ x } n) \text{ x } O(\max(d1,d2,\ldots,dn)\hat{}(n+1))$

When this array is implemented for 1-D array, it will be similar to Bubble sort. But as it will go in higher dimension, its complexity will be better compared to other general sorting algorithm.

7. Conclusion

We know that most of the 1D sorting algorithms including quick sort [1] has worst time complexity O(n2). If we sort the 2-D array of size m x n by transforming it into a linear array, the number of elements become m*n and thus the time complexity becomes $O(m^{2\text{ x}} n^2)$. On the other hand, the above proposed algorithm can do it more efficiently. Only merge sort [2] can do it differently with worst time complexity O(m x n x log(m x n)) but it will use extra space with space complexity O(m x n).

8. Acknowledgement

I would also like to thank Department of Computer Science & Engineering, Jadavpur University, Kolkata for giving me the platform to learn, grow, explore and research. Last but not the least, I would like to thank my parents and family for being the pillars of my life, without whom, nothing would have been possible.

9. References

[1] C. A. R. Hoare, Quicksort, The Computer Journal, Volume 5, Issue 1, 1962, Pages 10–16 https://doi.org/10.1093/comjnl/5.1.10
[2] Qin, S. (2008). Merge Sort Algorithm.

[3] Oyelami MO (2008). A Modified Diminishing Increment Sort for Overcoming the Search for Best Sequence of Increment for Shellsort”. J. Appl. Sci. Res., 4 (6): 760- 766.

[4] Oyelami MO, Azeta AA, Ayo CK (2007). Improved Shellsort for the Worst-Case, the Best-Case and a Subset of the Average-Case Scenarios. J. Comput. Sci Appl. 14 (2): 73-84.

[5] Sen, Sandeep & Scherson, Isaac & Shamir, Adi. (1986). Shear Sort: A True Two-Dimensional Sorting Techniques for VLSI Networks.. 903-908.

[6] I.D. Scherson and Sandeep sen, “A characterization of a parallel row-column sorting technique for rectangular arrays” ECE Technical Report No. 85-14, U.C.S.B. August 1985.

[7] C.D. Thompson & H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," Communications or the ACM, vol 20, no. 4,April 1977.

[8] D. Nassimi & S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Transactions on Computers, vol C-27, no\, January 1979

[9] K. Batcher, "Sorting Networks and their applications," in Proc. AFIPS Spring Joint Comput. Conf, vol 32, 1968

[10] M. Kumar & D.S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," IEEE Transaction on Computers, vol C-32, March.198

[11] Lang Hans-Werner et al., "Systolic sorting on a Mesh Connected Network," IEEE Transactions on Computers, vol C-34, no. 7, July1985.

[12] Ierardi, Douglas. (1994). 2d-bubblesorting in average time O (√ N lg N)*. 36-45. 10.1145/181014.181025.

[13] Hanan Samet. 2016. Sorting in space: multidimensional data structures for computer graphics and vision applications. In SIGGRAPH ASIA 2016 Courses (SA '16). Association for Computing Machinery, New York, NY, USA, Article 16, 1–42. DOI:https://doi.org/10.1145/2988458.2988503

[14] Samet, Hanan. (2010). Sorting in space: multidimensional, spatial, and metric data structures for computer graphics applications. 3. 10.1145/1900520.1900523.

[15] D.E. Knuth, "The Art of Computer Programming," vol. 3,Addison-Wesley, 1973