# Project Report: Matrix Sorting

## Contents

# Problem Definition

According to the assignment sheet, we are required to sort a given matrix in which the elements in each row and in each column are in non-decreasing order. Methods to implement them were described in the assignment sheet, and is discussed in the implementation presented below.

# Implementation

## Method 1

Method1 requires the sorting to be implemented in the array as a whole. The input read was a matrix as indicated in the file "input.txt", and for Method 1, this whole matrix is assumed to be a continuous single dimensional array. This array is sorted and is again written back to the file "pp0030_1.txt" with the dimension of the original read matrix in row-major order.

## Method 2

Method 2 requires the sorting to be implemented in the elements row-wise such that all the elements of the row are sorted. Again, the elements in each columns of the row-sorted matrix are sorted. The final matrix is written in the output file "pp0030_2.txt". The implementation done by me involves the sorting of each row, and then sorting each column in the matrix received after sorting each row.

## Sorting Strategy

As instructed in the assignment sheet, Quicksort was implemented. However, different variants of partition function of Quicksort were implemented. Each of them were performed on some test matrices generated, and one chosen from among them giving the best comparison count. Discussion and analysis of each of the partition strategy are presented in the subsequent section.

## Comparison Functions

Following shows the comparison functions that were implemented as required in the assignment. A call to each of the functions will increase the class-global variable named "self.comparison_count" by 1.

### EQ(a,b)

```
1. def EQ(self,a,b):
2.     self.comparison_count+=1
3.     return a==b
```

### LT(a,b)

```
1. def LT(self,a,b):
2.     self.comparison_count+=1
3.     return a<b
```

### GT(a,b)

```
1. def GT(self,a,b):
2.     self.comparison_count+=1
3.     return a>b
```

## ASSIGN Function and SWAP Function

ASSIGN function assigns a value of the second variable to the first variable. While doing so, it also increases the value of a class-global variable called “self_assignment_count” by 1. Swap function makes call to ASSIGN( ) function three times, thereby increasing the count by 3.

### ASSIGN(a,b)

```
1.  def ASSIGN(self,a,b):
2.      self.assignment_count+=1
3.      # since list is mutable
4.      a[0]=b[0]
5.      return
```

### SWAP(a,b)

```
1.  def SWAP(self,a_l,b_l):
2.      c = [0.0]
3.      self.ASSIGN(c,b_l)
4.      self.ASSIGN(b_l,a_l)
5.      self.ASSIGN(a_l,c)
```

## PARTITION Functions

Five implementations of partition were done. Following presents the idea behind the working of each of the functions implemented. This section also discussed the number of comparison made and if it achieves (right-left+1). Table in the following section shows the comparison between the statistics produced by them.

### Partition method using first element as pivot named PARTITION_1

```
1.  def PARTITION_1(self,array,left,right):
2.      i = left+1
3.      j = right
4.
5.      old_comp_count = self.comparison_count
6.
7.      while(i<=j):
8.          if not self.GT(array[i],array[left]):
9.              i+=1
10.         elif not self.LT(array[j],array[left]):
11.             j-=1
12.         else:
13.             self.SWAP(array[j],array[i])
14.             j-=1
15.             i+=1
16.
17.     self.SWAP(array[left],array[j])
18.     return j
```

*Figure 1 Implementation of PARTITION_1 function*

The implementation presented above assumes the leftmost element as the pivot. It then runs a loop from the left+1 element pointed by variable *i* and another variable *j* points to the right-most element. The algorithm is designed such that *i* gradually increases towards *j* whenever the element pointed to by *i* is less than pivot element. Similarly, when the element pointed to by *j* is greater than pivot, *j* is decreased towards *i.*

This algorithm has a benefit that when none of the above mentioned condition is met, the number of comparison made is 2. When only the first condition is met, the number of comparison made is just 1. These are both accompanied by both the increment in *i* and decrement in *j* or an increment in *i,* respectively. This reduces the number of comparison to be made, lesser than (right-left+1). But if the first condition fails and the second condition is true, then it is accompanied only by an decrease in *j.* This is a loss because we made two comparisons, but only reduced one indexing item.

Another problem with the above problem in not being able to achieve (right-left+1) number of comparison is if the elements pointed to by *i* is greater than pivot element, and the element pointed by *j* also turn out to be greater than pivot element continually in multiple consecutive iterations, then we will be doing the previous comparison of element pointed to by *i* in vain, because no change has been made in the element pointed to by *i.*

T(n) = 2T(n/2)+(n-1), using master theorem is O(nlogn), which might not be the case always since comparison made by the partition function can be more than n-1.

*Illustration:*

In case of the array [0.0, 3.0, 7.0, 6.6, 8.0, 4.0], following shows the comparison made using the above algorithm.

| Element No. | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Pivot = 0,i=2,j=6 | 0 | 3 (i) | 7 | 6.6 | 8 | 4 (j) |
| Iter1 | | 3<0, false | | | | 4>0, true,j-- |
| | | i | | | j | |
| Iter2 | | 3<0, false | | | 8>0, true,j-- | |
| | | i | | j | | |
| Iter3 | | 3<0, false | | 6.6>0, true,j-- | | |
| | | i | j | | | |
| | | | | | | |
| | | | | | | |

*Figure 2 Illustration of PARTITION_1*

The above table shows the first three iterations of one implementation of the algorithm given above. We can see, without any progression on value of *i,* we are making two comparison every iteration. This is a hinderance that prevents from having (right-left+1) comparison.

## Partition method using first element as pivot named PARTITION_2

```
1.  def PARTITION_2(self,array,left,right):
2.      old_comp_count = self.comparison_count
3.      index = left
4.      for i in range(left,right):
5.          if self.LT(array[i],array[right]):
```

```
            self.SWAP(array[i],array[index])
            index+=1
        self.SWAP(array[right],array[index])
        return index
```

*Figure 3 Implementation of PARTITION_2 function*

This implementation also takes the first element as pivot, but the most important thing is that the iteration runs exactly (right-left+1) times no matter what. According to the problem statement of the project, this the best implementation that can be thought of with exactly (right-left+1) comparison made in each call of partition

T(n) = 2T(n/2)+(n-1), using master theorem is O(nlogn) in all the cases

## Modification on PARTITION_2 named PARTITION_3

The function presented in Figure 4 shows slight modification of PARTITION_2. Here the item in the middle of the original array is considered to be pivot. Then this element is swapped with the element in the left. What follows is same as in the PARTITION_2. One added benefit it this provides added dynamics in the sense that the final pivot element will have higher probability of producing pivot somewhere in the middle.

T(n) = 2T(n/2)+(n-1), using master theorem is O(nlogn) in all the cases

```
def PARTITION_3(self,array,left,right):
    midIndex = (right+left)//2

    self.SWAP(array[left],array[midIndex])

    old_comp_count = self.comparison_count
    index = left
    for i in range(left,right):
        if self.LT(array[i],array[right]):
            self.SWAP(array[i],array[index])
            index+=1
    self.SWAP(array[right],array[index])
    return index
```

*Figure 4 Implementation of PARTITION_3 function*

## Modification on PARTITION_1 named PARTITION_4

The function presented in Figure 5 is a slight modification on function presented in PARTITION_1. The major demerit of making multiple comparison of elements in PARTITION_1 is eliminated in this case making use of while loop that checks of find the index of final element which is not greater than pivot element on right side, and the element which is greater on left side. If the left index in lesser than right index found, a swap is performed.

T(n) = 2T(n/2)+(n-1), using master theorem is O(nlogn), but at times the comparison count in partition function can be higher than n-1.

```
def PARTITION_4(self,array,left,right):
    i = left+1
```

```
3.      j = right
4.      old_comp_count = self.comparison_count
5.      while(i<=j):
6.          while self.LT(array[left],array[j]):
7.              j-=1
8.          while (i<=j) and (not self.GT(array[i],array[left])):
9.              i+=1
10.         if(i<=j):
11.             self.SWAP(array[i],array[j])
12.             j-=1
13.             i+=1
14.
15.     self.SWAP(array[left],array[j])
16.     return j
```

*Figure 5 Implementation of PARTITION_4 function*

### Modification on PARTITION_4 named PARTITION_5

Function presented in Figure 6 is a modification on PARTITION_4. Here the pivot element is chose from the middle of the array to provide some dynamics i.e. to partition as equally as possible. But since the comparison is saturated in PARTITION_4, this modification does rarely produce any improvement in terms of comparison count.

T(n) = 2T(n/2)+(n-1), using master theorem is O(nlogn), but at times the comparison count in partition function can be higher than n-1.

```
1.  def PARTITION_5(self,array,left,right):
2.      midIndex = (right+left)//2
3.      self.SWAP(array[midIndex],array[left])
4.      i = left+1
5.      j = right
6.      old_comp_count = self.comparison_count
7.
8.      while(i<=j):
9.          while self.LT(array[left],array[j]):
10.             j-=1
11.         while (i<=j) and (not self.GT(array[i],array[left])):
12.             i+=1
13.         if(i<=j):
14.             self.SWAP(array[i],array[j])
15.             j-=1,  i+=1
16.     self.SWAP(array[j],array[left])
17.     return j
```

*Figure 6 Implementation of PARTITION_5 function*

## Output

For the input file as shown below, the output file generated is shown below.

| **Input File text** | **Output File text** |
|---|---|

| | |
|---|---|
| 5 7<br>14 51.3 17 28.77 31 1 2<br>2 2 2 2 2.0 2 2<br>8 5 0.00 2 1 18.32 19<br>1 200.36 0 4 5.21 6.6 7<br>9 8 7 6.6 0.0 4 3 | 5 7<br>0.00 0.00 0.00 1.00 1.00 1.00 2.00<br>2.00 2.00 2.00 2.00 2.00 2.00 2.00<br>2.00 3.00 4.00 4.00 5.00 5.21 6.60<br>6.60 7.00 7.00 8.00 8.00 9.00 14.00<br>17.00 18.32 19.00 28.77 31.00 51.30 200.36<br>Comparison Count:= 163<br>Assignment Count:= 321 |

*Figure 7 Sample input and output file*

The output file is formatted to have eight places for each of the number printed in the file with precision of 2 decimal places. Copying and pasting in MS Word makes it look a bit skewed, but looking in standard text viewing software like notepad++ should not be an issue. The following two lines present the comparison count and assignment count.

## Analysis of Number of Comparison made

Figure 8 shows the statistics obtained for the input file provided to us. The matrix is of order 5X7, nlogn of which will be 124.43. All the data presented here are in the same order.

| Method | Partition Function | Comparison Count | Assignment Count | Remarks |
|---|---|---|---|---|
| 1 | PARTITION_1 | 257 | 132 | Comparison count greater than (right-left+1) |
| | PARTITION_2 | 170 | 234 | |
| | PARTITION_3 | 163 | 321 | |
| | PARTITION_4 | 213 | 129 | Comparison count greater than (right-left+1) |
| | PARTITION_5 | 186 | 204 | Comparison count greater than (right-left+1) |
| | | | | |
| 2 | PARTITION_1 | 151 | 168 | Comparison count greater than (right-left+1) |
| | PARTITION_2 | 131 | 315 | |
| | PARTITION_3 | 131 | 459 | |
| | PARTITION_4 | 165 | 168 | Comparison count greater than (right-left+1) |
| | PARTITION_5 | 151 | 294 | Comparison count greater than (right-left+1) |
| | | | | |

*Figure 8 Statistics for the input file provided*

According to Figure 8, the clear winner is the PARTITION_3 because of the following two reasons:

- It is a strict requirement that the number of comparison made in each partition is <=(right-left+1), but in PARTITION_1, PARTITION_4 and PARTITION_5 this requirement was not met.
- The focus of the algorithm design was comparison count as primary measure and assignment count as secondary measure.

Figure 9 shows the statistics obtained from 100X100 matrix. The order nlogn will be 40000. The maximum presented data of comparison count is 6x the nlogn value which is considered to be in order of the value.

| Method | Partition Function | Comparison Count | Assignment Count | Remarks |
|---|---|---|---|---|
| 1 | PARTITION_1 | 234144 | 85275 | Comparison count greater than (right-left+1) |
| | PARTITION_2 | 175969 | 216720 | |
| | PARTITION_3 | 175297 | 250812 | |
| | PARTITION_4 | 185882 | 93474 | Comparison count greater than (right-left+1) |
| | PARTITION_5 | 190281 | 120870 | Comparison count greater than (right-left+1) |
| | | | | |
| 2 | PARTITION_1 | 175921 | 96471 | Comparison count greater than (right-left+1) |
| | PARTITION_2 | 129320 | 226824 | |
| | PARTITION_3 | 129287 | 264948 | |
| | PARTITION_4 | 143663 | 98073 | Comparison count greater than (right-left+1) |
| | PARTITION_5 | 140911 | 138708 | Comparison count greater than (right-left+1) |
| | | | | |

*Figure 9 Statistics for a random 100x100 matrix*

Based on the data obtained from Figure 8 and Figure 9, the function PARTITION_3 was chosen to be the best partition function. In the code submitted as the final version, PARTITION function is the PARTITION_3 (which means PARTITION_3 was renamed to PARTITION to match the requirement).

## Method2 Analysis

Let us analyze the *nx4* matrix given below, where each of the row is already sorted in a non-decreasing order such that in any $i^{th}$ row ai<=bi<=ci<=di.

a1 b1 c1 d1
a2 b2 c2 d2
a3 b3 c3 d3
a.. b.. c.. d..
an bn cn dn

Arranging the column will reorder the matrix again. Let us consider that first element is the one to be first reordered, such that any element ai <= a1 is brought in place of it. This will still satisfy the condition with elements in the first row and maintain the non-decreasing order.

Considering in any $j^{th}$ column where any element xij lies and is less than the element x(i-1)(j), then reordering the $j^{th}$ column will still satisfy the condition of being non-decreasing row and non-decreasing column since the element xij will be placed in the position where x(i-1)(j) was already <= elements in row are placed right to it. Similar is the case when xij is greater.

Thus method 2 will always work in creating a matrix where each rows are in non-decreasing order and each column is on non-decreasing order.