# 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 contiguous 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”.

## 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.

### 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 Function

Multiple partition functions were tried. 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 (One Implementation)

|  |
| --- |
| 1. def PARTITION\_1(self,array,left,right): 2. i = left+1 3. j = right 4. old\_comp\_count = self.comparison\_count 5. while(i<=j): 6. if not self.GT(array[i],array[left]): 7. i+=1 8. elif not self.LT(array[j],array[left]): 9. j-=1 10. else: 11. # make a list and pass, since list is mutable 12. a\_l = [array[i]] 13. b\_l = [array[j]] 14. self.SWAP(a\_l,b\_l) 15. array[i] = a\_l[0] 16. array[j] = b\_l[0] 17. j-=1 18. i+=1 20. # make a list and pass, since list is mutable 21. a\_l = [array[left]] 22. b\_l = [array[j]] 23. self.SWAP(a\_l,b\_l) 24. array[left] = a\_l[0] 25. array[j] = b\_l[0] 26. return j |

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.*

#### 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 |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

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 (Second Implementation)

|  |
| --- |
| 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]): 6. # make a list and pass, since list is mutable 7. a\_l = [array[i]] 8. b\_l = [array[index]] 9. self.SWAP(a\_l,b\_l) 10. array[i] = a\_l[0] 11. array[index] = b\_l[0] 12. index+=1 13. # make a list and pass, since list is mutable 14. a\_l = [array[right]] 15. b\_l = [array[index]] 16. self.SWAP(a\_l,b\_l) 17. array[right] = a\_l[0] 18. array[index] = b\_l[0] 19. return index |

This implementation also takes the first element as pivot, but the thing is 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.

# Output

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

|  |  |
| --- | --- |
| **Input File text** | **Output File text** |
| 5 7  14 51.3 17 28.77 31 1 2  2 2 2 2 2.0 2 2  8 5 0.00 2 1 18.32 19  1 200.36 0 4 5.21 6.6 7  9 8 7 6.6 0.0 4 3 | 5 7  0.00 0.00 0.00 1.00 1.00 1.00 2.00  2.00 2.00 2.00 2.00 2.00 2.00 2.00  2.00 3.00 4.00 4.00 5.00 5.21 6.60  6.60 7.00 7.00 8.00 8.00 9.00 14.00  17.00 18.32 19.00 28.77 31.00 51.30 200.36  170 234 |

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.

## Analysis of Number of Comparison made