**Sparse matrix (SM)**

**Author:** George Vlassis **Version:** 1

**Definitions:**Suppose a matrix A of size r\*c (r= number or rows, c= number of columns). Also: z=number of 0s in A.

We call **sparsity** of A, and denote with s, the fraction:   
s= number of zero elements/number of elements = z/(r\*c)

We call **density** of A, and denote with d, the fraction:  
d= number of non zero elements/number of elements = [(r\*c)-z]/(r\*c) =1- z/(r\*c)= 1- s

A is called **sparse** if: number of zero elements>number of non zero elements <=> z>(r\*c-z) <=> 2z > r\*c <=> z> r\*c/ 2 (which means that more than half of A’s elements are 0s) <=>  
<=> z/(r\*c) > 0.5 <=> s> 0.5 <=> 1-d > 0.5 <=> d<0.5.

On the other hand, A is called **dense** if: number of zero elements<number of non zero elements <=> z<(r\*c-z) <=> 2z < r\*c <=> z< r\*c/ 2 (which means that less than half of A’s elements are 0s) <=>  
<=> z/(r\*c) < 0.5 <=> s< 0.5 <=> 1-d < 0.5 <=> d>0.5.

Notes:  
\*The above definitions can be easily generalized for any data structure.

\*The above definitions can be easily modified for the most common element of A, instead of 0s.

**Examples:**

A close up of a white background

Description automatically generated s=9/15=0.6=60%, d=1-0.6=0.4=40% => sparse

A close up of a keyboard

Description automatically generated s=4/16=0.25=25%, d=1-0.25=0.75=75% => dense

A 400\*400 matrix, converted to an image by associating each zero element to a white pixel, and each non-zero element to a black pixel. Code for the conversion (in **Java**), as well as the creation, of the matrix is included in the source files.  
A picture containing rain

Description automatically generateds=81.25%, d=18.75% => sparse

A 256\*512 matrix, converted to an image in the same way.  
A picture containing rain, nature, group, tower

Description automatically generateds=2.29%, d=97.71% => dense

**History:**

Sparsity and density are simply properties of matrices and thus it would not be correct to say that they were “invented”. What were indeed invented, are ways to utilize these properties in order to reduce both the storage space of sparse matrices, and the time complexity of some operations with them.

The first one who developed such ways, is Harry Max Markowitz. He is also the one who coined the term “sparse matrices”. Markowitz, while working on industrial capabilities analysis in the 1950s, came across systems with a large amount of equations, represented with matrices. These matrices were bigger than 200\*200, leading to systems that were unsolvable with the then known methods (and of course that’s age hardware). This provoked Markowitz to find a more efficient way to solve them. He observed that the corresponding matrices were mostly full of 0s. This observation, in combination with the fact that in such arrays, if you choose your pivots in a specific way (first choose as pivots the elements of the row/column with most 0s) you can solve them manually, led him to search for ways to take advantage of sparsity in order to make the operations in such matrices more efficient. And indeed he found them.

Albeit Markowitz did not himself implement these techniques in code, he laid the theoretical foundations in which latter implementations and further research were built upon. The earliest applications of sparse matrices(SM) were in the field of electrical engineering, in particular in power systems and electrical networks, and in chemical engineering.

\*Markowitz was awarded a Nobel prize in 1990 for having developed the theory of portfolio choice.

**Data structures:**Several data structures have been proposed for the more efficient storing of sparse matrices and/or faster operations on them. Some of the most commonly encountered are:  
1) The coordinate list (COO) structure. This structure consists of a list that contains 3-tuples of the form (row, column, value) for each non-zero element of the matrix.  
2) The list of lists (LIL) structure. This structure consists of one list per row, which contains 2-tuples of the form (column, value) for each non-zero element on the corresponding row.  
3) The dictionary of keys (DOK) structure. This structure is a dictionary, with keys 2-tuples of the form (row, column) and values the values of the non-zero elements of the matrix.  
4) The compressed sparse row (CSR) or the compressed sparse column (CSC) structures. They both consist of 2 lists and 1 fixed-length array, whose significance will be analyzed below.  
5) The modified compressed sparse row (MCSR) or the modified compressed sparse column (MCSC) structures. They are a modification of the CSR/ CSC structures and they both consist of 3 lists and 1 fixed-length array. Their significance will also be analyzed below.

\*All of the above structures are based on the simple premise that zero elements need not to be stored explicitly, like the non-zero ones. In case you are wondering why we don’t simply store them in an array of size (r\*c-z) the answer is that then, while we would have the non-zero elements, we would not have a way to refer to them by their indexes. So, the above structures only differ in how they tackle this problem.

**Compressed sparse row (CSR)/ Compressed sparse column (CSC):**We will first analyze CSR and then explain how CSC differs. In CSR we traverse the matrix row per row, starting from row 0, from left to right. We save each non-zero element we encounter in a list called **Values,** clearly of final size (r\*c-z). If we did not do anything more we would not have a way to refer to the elements by their indexes, as mentioned previously. So, we are looking for a way to modify the structure in order to be able to access the element on (row, column) with the corresponding 2-tuple.

We observe that, owing to how we traverse the matrix, elements of the same row will appear sequentially in *Values*, sorted by their columns. So, if we somehow knew where in *Values* each row r started, which would mean that we would also know where the next row (r+1) started and thus where the first row r ends, we would be able to access the values of each row. Special care should be given to the last row. Since there is no next row for the last row we would not know where the last row ended. Or wouldn’t we? Since the last row is, well, last, it ends in the last index of *Values*. In order to avoid checking if a row is last and make our code cleaner, we will simply also save the index in *Values* where the next row of the matrix WOULD start, had it one more row. So, we also store an array called **StartingIndexInValuesOfRow** with size (r+1), which will contain in each position r the starting index of row r in *Values*. As we traverse the matrix we have a variable named **Counter** (initially 0) and every time we encounter a non-zero element we increment it by one. Each time we start to traverse a new row since there will be *Counter* non-zero elements before it in *Values*, that row’s elements will start from *Counter.* Thus, this is what we store in *StartingIndexInValuesOfRow*’s position corresponding to that row.

\* To find the number of non-zero elements in a row r we can simple do: (StartingIndexInValuesOfRow[r+1]-1)- StartingIndexInValuesOfRow[r]+1=  
=(StartingIndexInValuesOfRow[r+1]- StartingIndexInValuesOfRow[r].

\* A row starts from StartingIndexInValuesOfRow[r] inclusive and ends at StartingIndexInValuesOfRow[r+1] exclusive.

\* If a row has no non-zero elements: StartingIndexInValuesOfRow[r+1]= StartingIndexInValuesOfRow[r].

Given a row, we can now use *Values* and *StartingIndexInValuesOfRow* to find all the non-zero elements in that row. From those elements, what is the element at column c (that is the element on (row, column) )? If we also store a list **ColumnsOfValues**, which stores at position p the column of the value=*Values*[p], we can easily answer that question. The final length of that list is also (r\*c-z).

Our structure is now complete. To recap, it consists of 2 lists and 1 array:  
1) Values, of size (r\*c-z)  
2) ColumnsOfValues, of size (r\*c-z)  
3) StartingIndexInValuesOfRow, of r+1  
, with a total size: 2(r\*c-z)+(r+1).

\* As is the case with the last element of StartingIndexInValuesOfRow, the first element of that array can also be omitted. This is due to the fact that it is always 0, and thus can be implicitly obtained. However, this would lead to messier code and therefore it is avoided. In both of the above cases, the choice I made is that **code complexity is more important than the storage savings** (a common decision in coding).

\*Given a CSR there is no way to infer the number of columns of the original matrix. If, and only if, the original matrix has at least one non-zero element in its last column, then the maximum value in ColumnsOfValues is this last column, but in any other case this does not hold.

Regarding CSC, the main ideas behind the structure are the same as CSR. This time, we traverse the matrix column per column, starting from column 0, from top to bottom, saving each non-zero element we encounter to **Values**, with size r\*c-z. We also store an array called **StartingIndexInValuesOfColumn**, with size c+1, which will contain in each position c the starting index of column c in *Values.* Lastly, we store a list **RowsOfValues**, with size r\*c-z, for each position p of which, holds the equality: (row of *Values*[i])= *RowOfValues*[i].

CSC consists of 2 lists and 1 array:  
1) Values, of size (r\*c-z)  
2) RowsOfValues, of size (r\*c-z)  
3) StartingIndexInValuesOfColumn, of c+1  
, with a total size: 2(r\*c-z)+(c+1)

\*It is obvious that CSR and CSC can also be used for non arithmetic matrices.

**Matters of space:**Having in mind that, typically, a matrix is saved as an array of size r, of arrays of size c, leading to a total size of r\*c, we can determine when CSR is beneficial ( as far as storage space is concerned). In these cases:

2(r\*c-z)+(r+1)< (r\*c) <=> 2(r\*c)-2z+r+1< (r\*c) <=> (r\*c)+r+1< 2z <=> [r(c+1) +1]/2 < z <=>

<=> -(r\*c)-r-1>-2z <=> +2(r\*c) -(r\*c)-r-1>+2(r\*c) -2z <=> (r\*c)-r-1 > 2[(r\*c)-z] <=>

[r(c-1)-1]/2>[ (r\*c)-z ] .

\*Making the reasonable approximations that c+1~c and r\*c+1~r\*c, we arrive at the rule of thumb that **CSR is beneficial** **if** r\*c/2<z, that is to say that **more than half the elements are zeros**.

Proceeding similarly for the case of CSC we get:  
2(r\*c-z)+(c+1)< (r\*c) <=> 2(r\*c)-2z+c+1< (r\*c) <=> (r\*c)+c+1< 2z <=> [c(r+1) +1]/2 < z <=>

<=> -(r\*c)-c-1>-2z <=> +2(r\*c) -(r\*c)-c-1>+2(r\*c) -2z <=> (r\*c)-c-1 > 2[(r\*c)-z] <=>

[c(r-1)-1]/2>[ (r\*c)-z ] .

We can also see that the total sizes of CSR and CSC are not equal. Using their total sizes’ formulas we can easily obtain that **if c<r, CSC is more beneficial, if r<c, CSR is more beneficial**, and lastly if r=c, both sizes are the same. Since both data structures have 2 lists of the same size, their size difference is due to the size difference of their arrays: StartingIndexInValuesOfRow (with size r+1) and StartingIndexInValuesOfColumn (with size c+1).

Examples:  
CSR:  
A close up of a keyboard

Description automatically generated \*Do note that 0-based array notation is used.  
Start to traverse through row 0: counter=0  
Finish to traverse through row 0: counter=0+2=2 (we encounter 2 and 8)  
Start to traverse through row 1: counter=2  
Finish to traverse through row 1: counter=2+0=2 (we do not encounter non-zero elements)  
Start to traverse through row 2 : counter=2  
Finish to traverse through row 2: counter=2+2=4 (we encounter 1 and 1)  
Start to traverse through row 3: counter=4  
Finish to traverse through row 3: counter=4+1=5 (we encounter 6)  
Start to traverse through row 4: counter=5  
Finish to traverse through row 4: counter=5+0=5 (we do not encounter non-zero elements)



Thus:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Values | 2 | 8 | 1 | 1 | 6 |
| ColumnsOfValues | 0 | 4 | 1 | 4 | 1 |

1. (1) (2) (3) (4)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| StartingIndexInValuesOfRow | 0 | 2 | 2 | 4 | 5 | 5 |

1. (1) (2) (3) (4) (5)

CSC:  
  
Start to traverse through column 0: counter=0  
Finish to traverse through column 0: counter=0+2=2 (we encounter 7 and 2)  
Start to traverse through column 1: counter=2  
Finish to traverse through column 1: counter=2+2=4 (we encounter 7 and 1)



Thus:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Values | 7 | 2 | 7 | 1 |
| RowsOfValues | 2 | 5 | 2 | 4 |

(0) (1) (2) (3)

|  |  |  |  |
| --- | --- | --- | --- |
| StartingIndexInValuesOfColumn | 0 | 2 | 4 |

(0) (1) (2)

**Modified compressed sparse row (MCSR)/ Modified compressed sparse column (MCSC):**Let’s suppose we already store a matrix in CSR. We want to set a previously zero element (r,c) to a non zero value v. How would we do that?

We will first find the correct new element’s position in Values. We first suppose that the correct position is StartingIndexInValuesOfRow[r], that is that the new non zero element of row r is the first non-zero element in that row. If StartingIndexInValuesOfRow[r]= StartingIndexInValuesOfRow[r+1], that is if the row previously was empty, our assumption is correct. If the row is not empty, but ColumnsOfValues[StartingIndexInValuesOfRow[r]]>c, our assumption is again correct. In any other case, our assumption is wrong. We make a new assumption, that the correct position is StartingIndexInValuesOfRow[r]+1. If the row previously had only 1 non-zero element then our assumption is correct. If not, but ColumnsOfValues[StartingIndexInValuesOfRow[r]+1]>c is again correct. In any other case ,our assumption is wrong and we make a new assumption that the correct position is StartingIndexInValuesOfRow[r]+2. The procedure continues similarly, until one assumption of ours is the correct one. The variable that will hold our assumptions will be called **AssumedIndex**.

Having done that, we insert v and c in their correct positions in Values and ColumnsOfValues. Now Values and ColumnsOfValues are correct. The only thing that remains is to check the correctness of StartingIndexInValuesOfRow.

Now, if we would traverse the resulting matrix like we did before, we would see that the *Counter* ,at the beginning of each row after the one in which the addition happened, would be bigger by one. This would be reflected in all of the values of StartingIndexInValuesOfRow after StartingIndexInValuesOfRow[r] (excluding StartingIndexInValuesOfRow[r]), and they would all be bigger by one. So, by performing this change manually, we, at last, arrive at the CSR of the new matrix.

This procedure can be sped up, and simplified, if we utilize another data structure, similar to the CSR, the MCSR. In this data structure we use 3 lists and 1 array:  
1) Values (list): Stores the non-zero elements.  
2) ColumnsOfValues (list): Stores the columns of the non-zero elements. Obviously, of the same size as Values.   
3) IndexOfFirstWithRow (list): Has a size r (not r+1 like CSR). Each element IndexOfFirstWithRow[i] is equal to the index in Values of the first element (in Values) of the row i. If there is no element in Values of the row i, IndexOfFirstWithRow[i]=-1 (by definition- this is utilized in our code)   
4) IndexOfNextWithSameRow (list): Has the same size as Values and ColumnsOfValues. For each element with value=Value[i], and column=ColumsOfValues[i], IndexOfNextWithSameRow[i] is equal to the index in Values of the next element (in Values) of the same row. If there is no such next element: IndexOfNextWithSameRow[i]=-1.

For MCSR, the procedure for setting a previously zero element (r,c) to v is as follows:  
1) Starting from i= IndexOfFirstWithRow[r], we check if IndexOfNextWithSameRow[i]=-1, and if not we make the assignment i= IndexOfNextWithSameRow[i] and proceed similarly. This continues until IndexOfNextWithSameRow[i]=-1. Then, *i* will be the index in Values of the final element in the “chain” of non-zero elements corresponding to row r. Reaching this point means that we, consecutively, went through all of the current non-zero elements of row r. (Note: If the IndexOfFirstWithRow[r]=-1 then the row r does not, currently, have any non zero elements)  
2) We add v,c, -1 in Values, ColumnsOfValues, IndexOfNextWithSameRow respectively.  
3) We set IndexOfNextWithSameRow[i] (or IndexOfFirstWithRow[r] if IndexOfFirstWithRow[r]=-1) to the last index of Values

This procedure is clearly simpler. Speed comparison between the procedures on the two data structures is performed below.

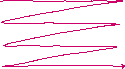
\* We approach the conversion of a normal matrix to MCSR cleverly, as follows:   
i) We first convert a matrix of the same size but with zero in each cell to MCSR. The conversion is trivial. The resulting MCSR will simply have empty Values, ColumnsOfValues, IndexOfNextWithSameRow arrays, and the array IndexOfFirstWithRow will be full of -1s.  
ii) We traverse the given matrix in any arbitrary way. Let’s say we choose to traverse it like we did in the case of CSR. For each non-zero element we encounter, we apply the procedure for the addition of a new element described above.

\*MCSC can be easily inferred from the descriptions of CSC and MCSR. To make things clearer, an example is given below.

\*We can modify the procedures described above so they enable us to set any (r,c) to any v. There are 4 distinct cases:   
i) (r,c)=0, v!=0 (which was described above)  
ii) (r,c)!=0, v!=0   
iii) (r,c)!=0, v=0  
iv) (r,c)=0, v=0  
For the case of CSR, the modifications are as follows:  
ii) If at any moment ColumnsOfValues[StartingIndexInValuesOfRow[r]]=c, then this means that (r,c)!=0, and if v!=0, we simply set Values[assumedIndex] to the new value v.  
iii) If at any moment (r,c)!=0 and v=0, we remove Values[assumedIndex], ColumnsOfValues[assumedIndex], and decrement each value of StartingIndexInValuesOfRow after StartingIndexInValuesOfRow[r] (excluding StartingIndexInValuesOfRow[r]), since the corresponding *Counter* value would be less by one.  
iv) In case i) we detect when (r,c)=0, find the position of the added non-zero element in Values and ColumnsOfValues and proceed accordingly. To cover case iv), we only slightly change our approach. After detecting (r,c)=0, and finding where a non-zero element would go in Values and ColumnsOfValues, we just do nothing (return), if v=0.   
For the case of MCSR, the modifications are as follows:  
ii) If at any moment, while going through the non-zero elements of row r, ColumnsOfValues[i]=c then this means that (r,c)!=0, and if v!=0, we simply set Values[i] to the new value v.  
iii) If at moment (r,c)!=0 and v=0 we remove Values[i], ColumnsOfValues[i], IndexOfNextWithSameRow[i]. Due to this removal each of the elements of Values, RowsOfValues, IndexOfNextWithSameColumn AFTER the one we just removed is 1 cell closer to the beginning of the arrays. This means that its index is smaller by one. So, we decrement each value in IndexOfNextWithSameColumn and in IndexOfFirstWithColumn, that is greater than i.  
iv) If indeed (r,c)=0 (detected in the first case) and v=0, then we do not have to do anything (return)

Examples:  
MCSR:  
A close up of a clock

Description automatically generated \*Do note that 0-based array notation is used.



Initially, Values, ColumnsOfValues, IndexOfNextWithSameRow are all empty, while IndexOfFirstWithRow is 1\*4, full of -1s.

First, we encounter (0,0,10). We extend Values, ColumnsOfValues, IndexOfNextWithSameRow by one, set the last element of each 10, 0 and -1 respectively, and, since IndexOfFirstWithRow[0]=-1, se set IndexOfFirstWithRow[0] to 0.

Then, we encounter (0,2,-3). We extend Values, ColumnsOfValues, IndexOfNextWithSameRow by one, set the last element of each -3, 2 and -1 respectively, and, since IndexOfFirstWithRow[0]=0 and IndexOfNextWithSameRow[0]=-1, we set IndexOfNextWithSameRow[0] to 1.

After that we encounter (0,3,-2). We extend Values, ColumnsOfValues, IndexOfNextWithSameRow by one, set the last element of each -2, 3 and -1 respectively, and, since IndexOfFirstWithRow[0]=0, IndexOfNextWithSameRow[0]=1 and IndexOfNextWithSameRow[1]=-1, we set IndexOfNextWithSameRow[1] to 2.

We continue similarly(\*->\*->\*->\*->\*), to arrive at:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Values | 10 | -3 | -2 | 15 | -1 | 20 | -1 | 50 |
| ColumnsOfValues | 0 | 2 | 3 | 1 | 2 | 2 | 0 | 3 |
| IndexOfNextWithSameRow | 1 | 2 | -1 | -1, 4 | -1 | -1 | -1, 7 | -1 |

1. (1) (2) (3) (4) (5) (6) (7)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IndexOfFirstWithRow | 0 | -1,3 | -1,5 | -1,6 |

1. (1) (2) (3)

If we set (2,3) (previously 0) to -2, we get:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Values | 10 | -3 | -2 | 15 | -1 | 20 | -1 | 50 | -2 |
| ColumnsOfValues | 0 | 2 | 3 | 1 | 2 | 2 | 0 | 3 | 3 |
| IndexOfNextWithSameRow | 1 | 2 | -1 | 4 | -1 | -1, 8 | 7 | -1 | -1 |

1. (1) (2) (3) (4) (5) (6) (7) (8)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IndexOfFirstWithRow | 0 | 3 | 5 | 6 |

(0) (1) (2) (3)

If we set (0,1) (previously 0) to 96, we get:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Values | 10 | -3 | -2 | 15 | -1 | 20 | -1 | 50 | -2 | 96 |
| ColumnsOfValues | 0 | 2 | 3 | 1 | 2 | 2 | 0 | 3 | 3 | 1 |
| IndexOfNextWithSameRow | 1 | 2 | -1,9 | 4 | -1 | 8 | 7 | -1 | -1 | -1 |

1. (1) (2) (3) (4) (5) (6) (7) (8) (9)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IndexOfFirstWithRow | 0 | 3 | 5 | 6 |

(0) (1) (2) (3)

Lastly, if we set (1,1) (previously 15) to 66 we get:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Values | 10 | -3 | -2 | 15,66 | -1 | 20 | -1 | 50 | -2 | 96 |
| ColumnsOfValues | 0 | 2 | 3 | 1 | 2 | 2 | 0 | 3 | 3 | 1 |
| IndexOfNextWithSameRow | 1 | 2 | 9 | 4 | -1 | 8 | 7 | -1 | -1 | -1 |

1. (1) (2) (3) (4) (5) (6) (7) (8) (9)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IndexOfFirstWithRow | 0 | 3 | 5 | 6 |

(0) (1) (2) (3)

MCSC:  
A close up of a clock

Description automatically generated



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Values | 1 | 1 | 1 | 1 |
| RowsOfValues | 1 | 0 | 2 | 1 |
| IndexOfNextWithSameColumn | -1 | 2 | -1 | -1 |

1. (1) (2) (3)

|  |  |  |  |
| --- | --- | --- | --- |
| IndexOfFirstWithColumn | 0 | 1 | 3 |

1. (1) (2)

If we set (2,1) (previously 1) to 0 we get the following result.

|  |  |  |  |
| --- | --- | --- | --- |
| Values | 1 | 1 | 1 |
| RowsOfValues | 1 | 0 | 1 |
| IndexOfNextWithSameColumn | -1 | -1 | -1 |

1. (1) (2)

|  |  |  |  |
| --- | --- | --- | --- |
| IndexOfFirstWithColumn | 0 | 1 | 2 |

(0)(1) (2)

If we set (1,0) (previously 1) to 0, following the same steps as above, we arrive at:

|  |  |  |
| --- | --- | --- |
| Values | 1 | 1 |
| RowsOfValues | 0 | 1 |
| IndexOfNextWithSameColumn | -1 | -1 |

1. (1)

|  |  |  |  |
| --- | --- | --- | --- |
| IndexOfFirstWithColumn | -1 | 0 | 1 |

(0)(1) (2)

**Algorithms:**In the source code(**Java**) we implemented only CSR and MCSR (as separate classes), since CSC and MCSC are highly similar (in a way “symmetrical”). The constructors and the methods to set (r,c)=v of those structures have already been studied. While the majority of the other methods are trivial,, or explained in the comments there are a few that need further clarification:

CSR:  
toMatrix(): We first create a matrix of the same size as the matrix that CSR represents. We fill this matrix with 0s. We then iterate over the rows of the matrices. Consecutively, for every row, we iterate over the non-zero elements of that row adding each one of them to the matrix we just initialized. These non-zero elements are located between the indexes StartingIndexInValuesOfRow[row] and StartingIndexInValuesOfRow[row+1] in Values and ColumnsOfValues. After all the iterations, every non-zero elements in CSR will have been added in the matrix. All the other elements are zero, and thus correct from the initialization.

CSRxM(): This calculates the product in normal matrix format of the Multiplier, represented in CSR, and Multiplicand, which is in normal matrix format. First, we create a matrix of size  
multiplier.rows\*multiplicand.columns. We will consecutively calculate every element of that matrix, row by row and column by column. Every element is calculated via the following simple procedure: Initially curProduct=0. We iterate over every non-zero element of the element’s row in Multiplier. We calculate the subproduct between every one of them and the corresponding element in Multiplicand. We add this subproduct in curProduct. By the end the curProduct contains the true value of the product’s element.

transpose(): Works nearly the same as toMatrix(). We again initially have a matrix full of 0s and then iterate over every row and every non-zero element of that row. However, this time, we assign these elements to matrix[c][r] instead of matrix[r][c], like we did in toMatrix().

MCSR:  
toMatrix(): We first create a matrix of the same size as the matrix that MCSR represents. We fill this matrix with 0s. We then iterate over the rows of the matrices. Consecutively, for every row, we iterate over the non-zero elements of that row adding each one of them to the matrix we just initialized. The iterations happen differently from CSR. Starting from index i= indexOfFirstWithRow[row], we check if i != 0, and if yes we add Values[i] to the matrix, set i= IndexOfNextWithSameRow[i] and proceed similarly. Otherwise, we proceed to the next row. After all the iterations, every non-zero elements in MCSR will have been added in the matrix. All the other elements are zero, and thus correct from the initialization.

MCSRxM(): This calculates the product in normal matrix format of the Multiplier, represented in MCSR, and Multiplicand, which is in normal matrix format. First, we create a matrix of size multiplier.rows\*multiplicand.columns. We will consecutively calculate every element of that matrix, row by row and column by column. Every element is calculated via the following simple procedure: Initially curProduct=0. We iterate over every non-zero element of the element’s row in Multiplier (the iterations happen like they did in toMatrix()). We calculate the subproduct between every one of them and the corresponding element in Multiplicand. We add this subproduct in curProduct. By the end the curProduct contains the true value of the product’s element.

transpose(): Similar to toMatrix() but this time we assign every element to matrix[c][r] instead of matrix[r][c], like we did in toMatrix().

\* In the Main class (which contains the main() method) there are also a lot of algorithms implemented, all of them simple or sufficiently documented in the comments.

**Array multiplication: ???**