PageRank Algorithm

By Rami Alkadri and Anass Qneibi

Summary of Datasets

Name	Dimension (N)
3WebPages.csv	3
10WebPages.csv	10
MediumAmountWebpages.csv	200
LargeAmountWebpages.csv	880
ExtraLargeAmountWebPages.csv	3400

The dimension mentioned in the chart is the number of unique web pages in the dataset. All these data sets differ from one another significantly and should help us picture the Big(O) of our algorithm. We decided to go with *N* being the number of unique web pages, because it best demonstrates the size of our dataset in the simplest manner.

Output/Correctness/Test Suite

1) Matrix Multiplication Test Cases

Our Page Rank Algorithm is largely dependent on the success of our matrix multiplication function. We needed to make sure that this function works on a varying size of Matrices and Vectors. We also want this function to raise a warning when a case of invalid multiplication occurs. Our function's success was verified with the use of an online matrix multiplier calculator.

For note:

```
{{4, 0.5}, {0, 2}};
```

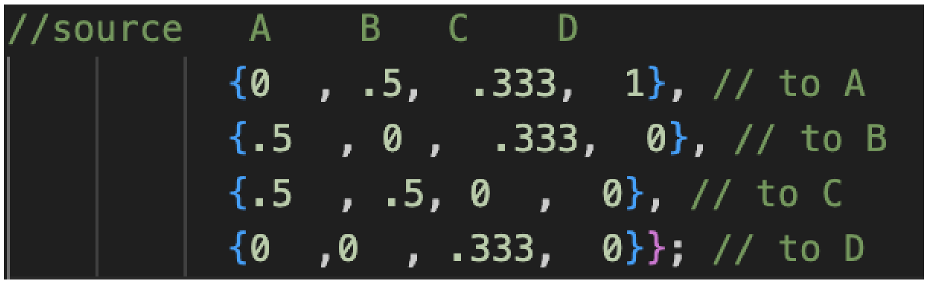
{4, 0.5} is row 0, {0, 2} is row 1.
4 and 0 are a part of the 0th column, 0.5 and 2 are a part of the 1st column.

- **"Small Matrix Vector Multiplication"**: This test case multiplies a 2x2 Identity matrix by a 2x1 vector. We had a REQUIRE statement that makes sure the expected matrix is equivalent to our result vector (obtained from the use of our multiplyMatrixByVector function). We also had another 2x2 matrix that was made randomly multiplied by a 2x1 vector also made randomly. Also, another REQUIRE statement to ensure correct multiplication.
- **"Medium Matrix Vector Multiplication":** Here we multiply a 5x5 Identity matrix with a random 5x1 vector, and a separate randomly made 5x5 matrix by a random 5x1. These are tested with two REQUIRE statements that passed. In this test case, we also have a couple REQUIRE_THROWS_AS statements that make sure our multiplyMatrixByVector function throws an invalid argument error when tasked with multiplying invalid sizes matrices, such as a 5x5 with a 1x1, and a 5x5 with a 4x1. These also pass.
- **"Large Matrix Multiplication":** The goal of this test case was to make sure our function would work for larger matrices, and would be able to handle the larger workload. Here a random 15x15 matrix is multiplied with a random 15x1 vector. This test case passed, and was verified by calculator.

2) Constructor Test Cases

Rami Alkadri, Anass Qneibi, PageRank, CS225

Making sure we had a working constructor that correctly populates the Adjacency Matrix was a priority. To be able to see that our constructor was working, we used 3 CSVs of different sizes. To find the expected adjacency matrix we had to manually populate it ourselves using the CSVs and a notebook. This was particularly challenging for the 10 WebPages Test.
Note: We rounded our vector values to 3 digits for ease of comparison.
Note: For clarity, (imagine this matrix is called Adj)
Adj[0][0], tells you whether there is a connection from A to A.
Adj[0][1], tells you whether there is a connection from B to A.
Ajd[1][0], tells you whether there is a connection from A to B.
Rows are destinations, Columns are sources. Our adjacency matrix is also normalized so each point also serves as a probability (which will help us in calculating the PageRank).



```
//source   A     B    C     D
          {0   , .5,   .333,  1}, // to A
          {.5   , 0 ,   .333,  0}, // to B
          {.5   , .5, 0   ,  0}, // to C
          {0   ,0   , .333,  0}}; // to D
```

- **"Test Constructor on 3 Webpages File"**: (We used the pages.csv)

We first needed to make sure that our adjacency matrix was correctly populated using a small sized csv. There were 3 different aspects we want to test: the number of rows in the populated adjacency matrix was equal to the number of unique web pages; the number of columns in the populated adjacency matrix was equal to the number of unique web pages; and most importantly, the fact that the adjacency matrix was essentially equal to the expected.

- **"Test Constructor on 4 Webpages File"**: (We used the 4WebPages.csv)

This test is almost identical to the 3 Webpages File, except we used 4 web pages instead of 3. We wanted to make sure our constructor populated our adjacency matrix correctly using a slightly larger csv.

- **"Test Constructor on 10 Webpages File"**: (We used the 10WebPages.csv)

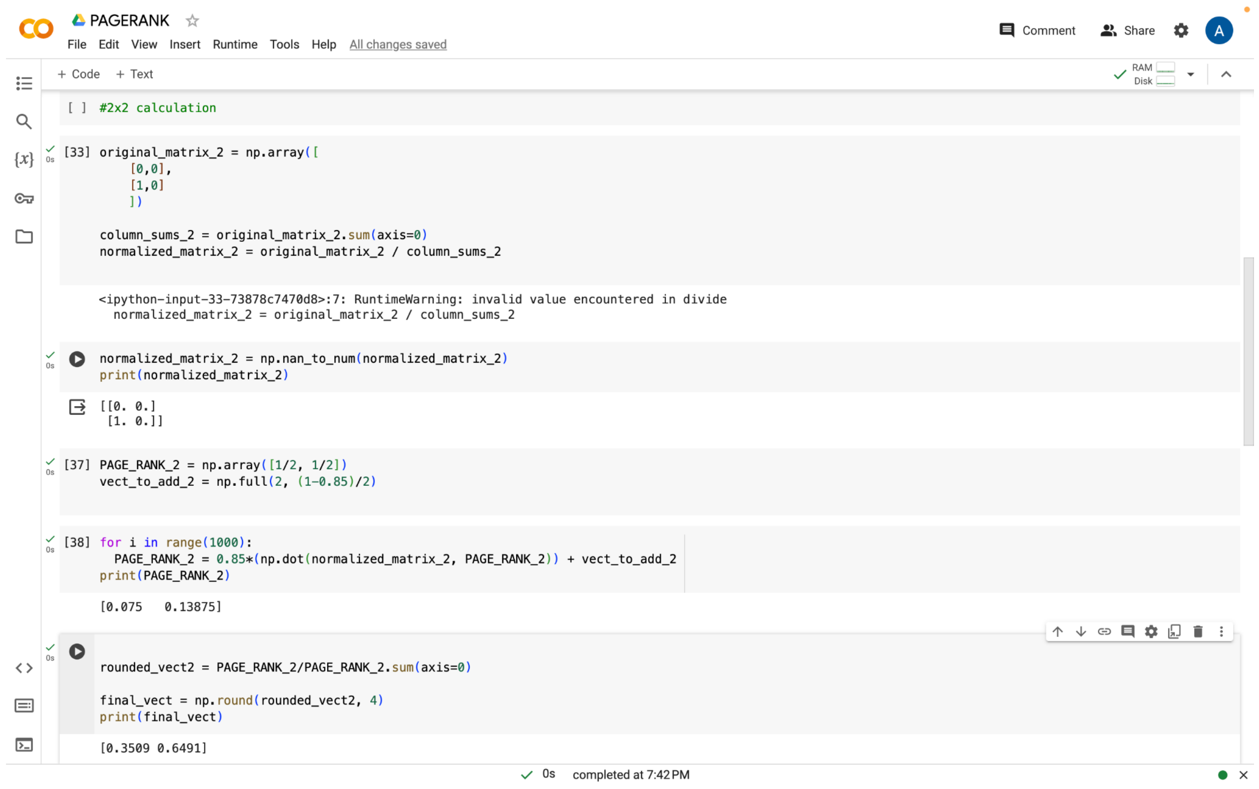
Last but not least, we had to test our constructor using a larger csv with more data points. We checked our result adjacency matrix (from the constructor) against a manually populated adjacency matrix. This test case confirms the constructor's success.

3) Page Rank Calculation Test Cases

The last series of test cases check whether our calculatePageRank works as intended. It also checks whether our getWebsiteRank function correctly outputs the position (regarding pagerank) a specific string/link/code resides relative to its dataset.

These last test cases ensure that our algorithm works correctly from start to finish. We test 3 different datasets here: 2 Webpages, 4 Webpages and 10 Webpages.
To obtain the expected PageRank vector, we wrote an alternative python mini program that calculates the page Rank. We compared the expected PageRank vectors generated from Python against our results. Here are screenshots of the Python code used for the 2,4,10 Webpage tests.
Note: We used the numpy library.
Note: The description of the test cases are right after the screenshots.

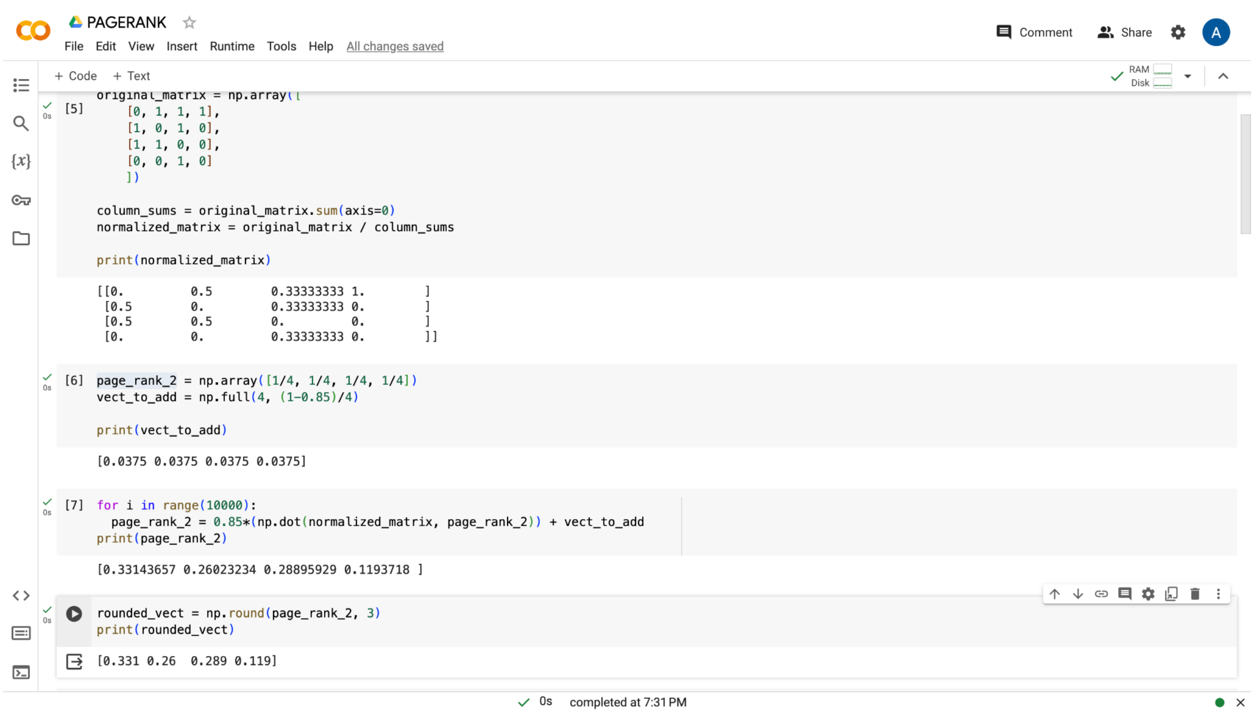
2 WebPages



```
[ ] #2x2 calculation

[33] original_matrix_2 = np.array([
         [0,0],
         [1,0]
         ])

     column_sums_2 = original_matrix_2.sum(axis=0)
     normalized_matrix_2 = original_matrix_2 / column_sums_2

     <ipython-input-33-73878c7470d8>:7: RuntimeWarning: invalid value encountered in divide
       normalized_matrix_2 = original_matrix_2 / column_sums_2

     normalized_matrix_2 = np.nan_to_num(normalized_matrix_2)
     print(normalized_matrix_2)

     [[0. 0.]
      [1. 0.]]

[37] PAGE_RANK_2 = np.array([1/2, 1/2])
     vect_to_add_2 = np.full(2, (1-0.85)/2)

[38] for i in range(1000):
       PAGE_RANK_2 = 0.85*(np.dot(normalized_matrix_2, PAGE_RANK_2)) + vect_to_add_2
     print(PAGE_RANK_2)

     [0.075    0.13875]

     rounded_vect2 = PAGE_RANK_2/PAGE_RANK_2.sum(axis=0)

     final_vect = np.round(rounded_vect2, 4)
     print(final_vect)

     [0.3509 0.6491]
```

Rami Alkadri, Anass Qneibi, PageRank, CS225

4 WebPages



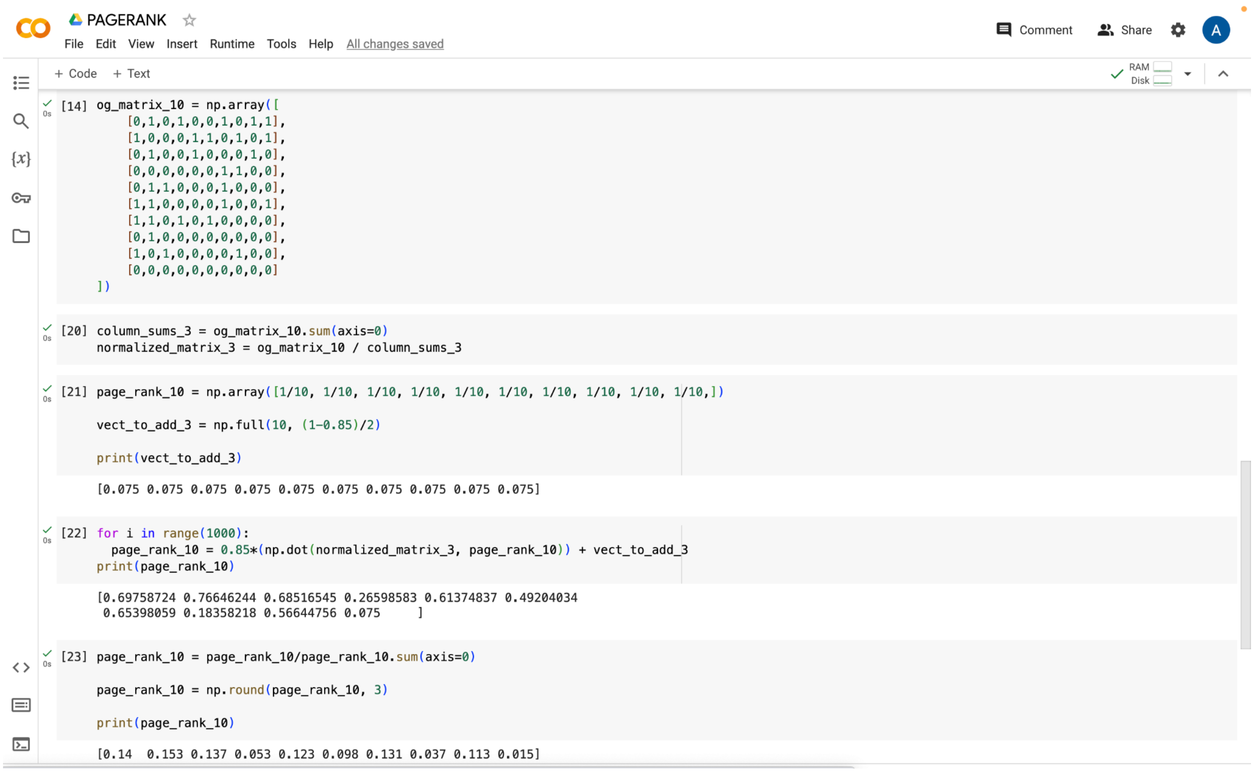
```
[5] original_matrix = np.array([
        [0, 1, 1, 1],
        [1, 0, 1, 0],
        [1, 1, 0, 0],
        [0, 0, 1, 0]
        ])

    column_sums = original_matrix.sum(axis=0)
    normalized_matrix = original_matrix / column_sums

    print(normalized_matrix)

    [[0.         0.5        0.33333333 1.        ]
     [0.5        0.         0.33333333 0.        ]
     [0.5        0.5        0.         0.        ]
     [0.         0.         0.33333333 0.        ]]
```

```
[6] page_rank_2 = np.array([1/4, 1/4, 1/4, 1/4])
    vect_to_add = np.full(4, (1-0.85)/4)

    print(vect_to_add)

    [0.0375 0.0375 0.0375 0.0375]
```

```
[7] for i in range(10000):
      page_rank_2 = 0.85*(np.dot(normalized_matrix, page_rank_2)) + vect_to_add
    print(page_rank_2)

    [0.33143657 0.26023234 0.28895929 0.1193718 ]
```

```
    rounded_vect = np.round(page_rank_2, 3)
    print(rounded_vect)

    [0.331 0.26  0.289 0.119]
```

10 WebPages



```
[14] og_matrix_10 = np.array([
         [0,1,0,1,0,0,1,0,1,1],
         [1,0,0,0,1,1,0,1,0,1],
         [0,1,0,0,1,0,0,0,1,0],
         [0,0,0,0,0,0,1,1,0,0],
         [0,1,1,0,0,0,1,0,0,0],
         [1,1,0,0,0,0,1,0,0,1],
         [1,1,0,1,0,1,0,0,0,0],
         [0,1,0,0,0,0,0,0,0,0],
         [1,0,1,0,0,0,0,1,0,0],
         [0,0,0,0,0,0,0,0,0,0]
     ])
```

```
[20] column_sums_3 = og_matrix_10.sum(axis=0)
     normalized_matrix_3 = og_matrix_10 / column_sums_3
```

```
[21] page_rank_10 = np.array([1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10,])

     vect_to_add_3 = np.full(10, (1-0.85)/2)

     print(vect_to_add_3)

     [0.075 0.075 0.075 0.075 0.075 0.075 0.075 0.075 0.075 0.075]
```

```
[22] for i in range(1000):
       page_rank_10 = 0.85*(np.dot(normalized_matrix_3, page_rank_10)) + vect_to_add_3
     print(page_rank_10)

     [0.69758724 0.76646244 0.68516545 0.26598583 0.61374837 0.49204034
      0.65398059 0.18358218 0.56644756 0.075     ]
```

```
[23] page_rank_10 = page_rank_10/page_rank_10.sum(axis=0)

     page_rank_10 = np.round(page_rank_10, 3)

     print(page_rank_10)

     [0.14  0.153 0.137 0.053 0.123 0.098 0.131 0.037 0.113 0.015]
```

Rami Alkadri, Anass Qneibi, PageRank, CS225

Note: We use a tolerance of 0.001 to measure whether convergence has occurred, we also use a damping factor of 0.85, which is the most commonly used one. Damping factor helps adjust the PageRank vector take into account the probability someone accesses a webpage by searching for it, without being directed to it by another link.

-**"Page Rank Calculation 2x2"**: (Used the 2WebPages.csv) 2 Webpages
This test case tests our calculatePageRank function using a small csv. This test case also uses the PageRank constructor that we test in the second series of tests. After calling the constructor, and then the calculatePageRank function. To get the expected PageRank vector we used the Python code provided above, and to get the expected website rank (for a website/link/string), we just eyeballed the printed PageRank vector from Python.
It was important to test both of these functions to ensure our algorithm was successful.

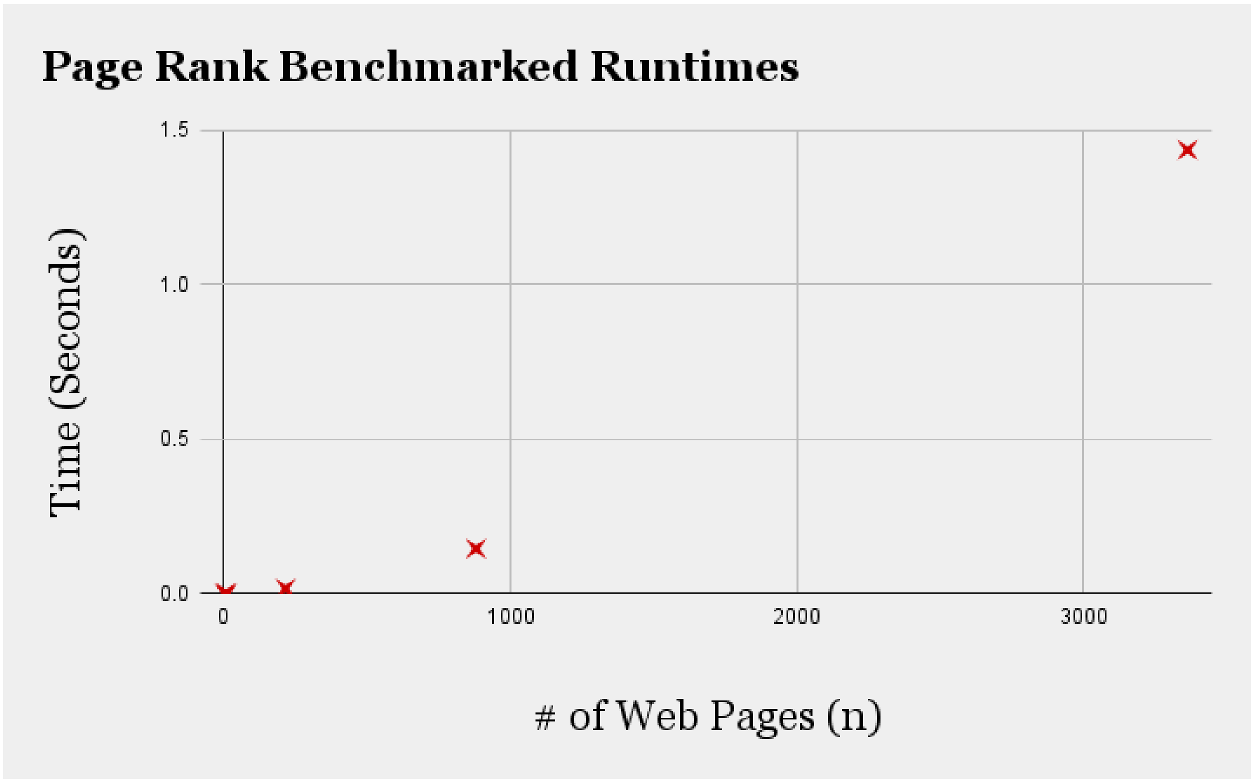
-**"Page Rank Calculation 4x4"**: (Used the 4WebPages.csv) 4 Webpages
This tests calculatePageRank function and the getWebsiteRank. This follows a similar process to the previous test. We make sure that expectedPageRankVect is equivalent to our PageRank vector. We also do 4 separate checks to make sure that the Web Pages are ranked in the correct order.

-**"Page Rank Calculation 10x10"**: (Used the 10WebPages.csv) 10 Webpages
This follows a similar process to the previous tests, but features a much larger csv that ensures the functionality of our algorithm. This was understandably the toughest one to do manually, because of the amount of edges in the adjacency matrix. From there we used the python code provided above.

Why Our Test Suite Works

Our test suite works because it tests each element in our PageRank Algorithm from start to finish. It is comprehensive and thorough. We test CSVs of varying sizes, some small and others much larger. We also obtained our expected vectors/matrices using tried and tested calculators, and using work that was checked for accuracy numerous times.

Benchmarking the Algorithm & Big O



After documenting the runtimes of the different sized data sets at our disposal, we can see a clear convergence towards the O(N^2) runtime. We can prove that the Big O of our algorithm is O(N^2) from the pseudocode of each function:

buildAdjacencyMatrix(filename) pseudocode:

```
Read each line in file and collect unique pages into a set
  Initialize adjacency matrix of size N x N (where N is the number of unique pages)
  For each line in the file:
    Parse source and destination
    Update the adjacency matrix with 1 at row (destination) col (source)
  For each row in adjacency matrix
      For each col in row
          Divide each number by the outlinks count within its column
```

Complexity: It is important to note that the size of each file is exactly the amount of edges that exist in our adjacency matrix. We loop through each line of the file twice, where the first pass is for collecting unique pages (O(E) complexity, with E representing the number of edges) and the second pass is for building the matrix. However, the dominant part in terms of complexity is the normalization of the adjacency matrix, which involves iterating over each of its N^2 elements, leading to an overall time complexity of O(N^2).

multiplyMatrixByVector(matrix, vector) pseudocode:

```
For each row in matrix:
    For each element in the vector:
        Multiply current element in matrix by element in vector
        Add the product to vector which will be returned at element index
```

Complexity: Given a matrix which is size N x N and a vector which is size N, this function contains a nested loop structure which iterates over N rows and N elements in each row. This gives the multiplyMatrixByVector function a time complexity of O(N^2).

calculatePageRank(maxIterations, tolerance) pseudocode:

```
For each maxIterations:
    multiplyMatrixByVector(adjacency_matrix, ranks)
    For each rank in ranks:
        Sum previous rank with product of previous rank and damping factor
    If isConverged
        Break
For each rank in ranks:
    Divide rank by sum of all elements in ranks vector
```

Complexity: maxIterations is a constant number decided by the user, so we can ignore its effect on the time complexity of our algorithm. The primary complexity within each iteration comes from the `multiplyMatrixByVector` function, which is O(N^2). The operations following the matrix multiplication, including applying the damping factor and normalizing the ranks, are O(N) and do not change the overall O(N^2) complexity. This gives the calculatePageRank function a time complexity of O(N^2).

Overall Time Complexity: Based on the pseudocode and the analysis of the key functions in our implementation, it is justified to conclude that the overall time complexity of our PageRank algorithm is O(N^2). This is due to the operations of building the adjacency matrix and multiplying the adjacency matrix by a vector having a time complexity of O(N^2). These time-complexities dominate the overall algorithm's Big O, which gives us a time complexity which converges towards O(N^2) in the long run.

Our PageRank algorithm did not achieve the theoretical Big O of O(E) due to the use of a dense matrix representation. In contrast, a sparse matrix, which only stores and processes non-zero elements, could have potentially reduced the complexity of matrix operations and is optimal for the PageRank algorithm. However, it's important to note that even with a sparse matrix, the complexity might not be exactly O(E) but could be significantly less than O(N^2), depending on the matrix's sparsity and the specific implementation details.