Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Fall 2022

Homework 2 – 2D linked lists for sparse matrix operations

Due: 01/11/2022, Tuesday, 11:55

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

You HAVE TO write down the code on your own.
You CAN NOT HELP any friend while coding.
Plagiarism will not be tolerated!!

Introduction

In this homework, you will implement operations on a very special type of data structure – a sparse matrix. You will read sparse matrices from text files, print them, add them, and check if they're symmetric.

Sparse matrices

As you know, a matrix is simply a table of numbers. It's a bunch of values stored in rows and columns. A *sparse* matrix is a matrix in which most values are zeros. This type of matrix is very common. It is found in graphics applications, scientific simulations, neural networks, and so many more areas of computer science. The following are two matrices, the left one is dense, and the right one is sparse.

32	21	0	19	57	74
-1	10	8	12	44	52
11	0	9	11	-14	19
32	83	0	19	57	85
-1	-185	8	687	0	56
0	58	68	19	765	19

0	0	0	0	12	0
-1	0	0	0	44	0
4	0	0	11	0	5
0	0	0	6	0	0
0	4	8	0	0	-1
0	0	0	19	0	-4

Figure 1: The matrix to the left is dense: most of its cells are not zero. The matrix to the right is sparse. It's mostly zeros with a few cells that are not zero.

Let's ask ourselves: if we have a matrix in our code and we know it's a sparse matrix. How should we store it? I mean, what data structure should we use for it? A vector of vectors? A linked list?

Computer scientists asked themselves this same question. They looked at these matrices and realized that, rather than store sparse matrices like you would a normal matrix, they can instead store *only the values that are not zero* (which we call non-zeros) and that would be enough to represent the entire matrix!

First of all, what does it mean to only store the non-zeros? It means going over every element that is not zero in the matrix and storing its location and value. So, for example, for the sparse matrix on the right of Figure 1, we could store only the following data (we start counting from 0):

```
row 0 column 4 value 12
row 1 column 0 value -1
row 1 column 4 value 44
row 2 column 0 value 4
row 2 column 3 value 11
row 2 column 5 value 5
row 3 column 3 value 6
row 4 column 1 value 4
row 4 column 2 value 8
row 4 column 5 value -1
row 5 column 3 value 19
row 5 column 5 value -4
```

This would be sufficient to express the entire sparse matrix. How? Well, if I want to know what the value of any cell in the matrix is, all I have is to search this list. I will either find the value in the list, or I won't it, and that would mean it's actually 0.

This realization is extremely useful. It saves computer scientists so much space. Imagine a matrix with 1 million rows and 1 million columns and only 10 million non-zeros. If we store it like we store a normal matrix (vector of vectors) we would need 1 million x 1 million x 4 bytes/value = 4 terabytes. If we store it as a sparse matrix, we can do the same with tens of thousands of times less memory!

A sparse data structure is extremely useful due to its memory saving. However, its strange storage pattern makes operations on it more challenging to implement than a normal matrix.

In this homework, you are going to build a sparse matrix data structure using a two-dimensional linked list. It will be based on this idea of only storing the values that are not zero. We will be providing the data structure explanation and you will be implementing a few functions for it, namely reading it from a file, printing it to the console, deleting it, checking if it's symmetric, finding its transpose, and adding sparse matrices together. The following section will explain this data structure in detail.

2D Linked List representation of a Sparse Matrix

We will represent a sparse matrix as a two-dimensional linked list. The following is a graphical representation of the sparse matrix in Figure 1.

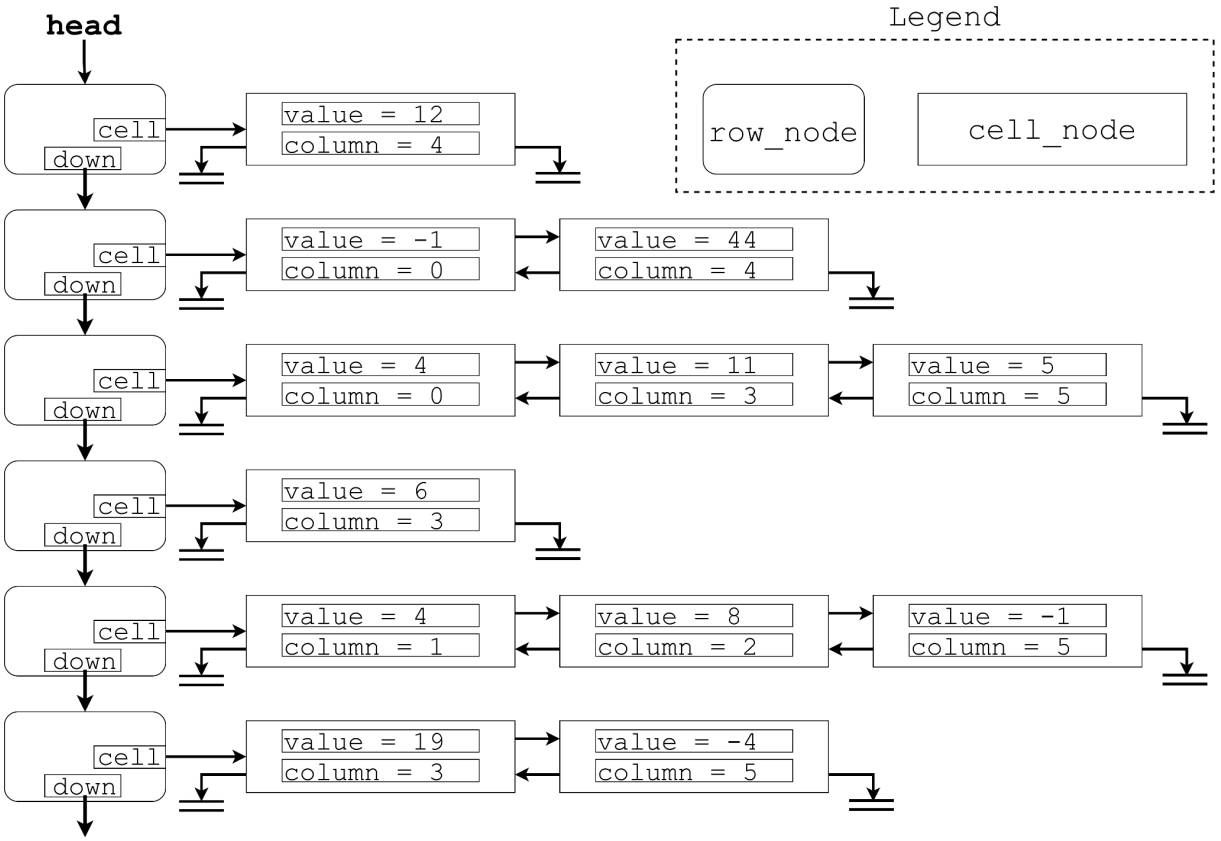

Figure 2: a graphical representation of the 2D linked list representation of sparse matrices.

Each row in the matrix will have a corresponding row in the linked list, There are two types of nodes in this data structure. The first is `cell_node` defined by this struct:

```
struct cell_node{
   cell_node* right;
   cell_node* left;
   int column;
   int value;
};
```

Each cell represents a single non-zero element. A cell contains both the *value* of the non-zero and the *column* in which it occurs. It also points at the `cell_node` to its left, which must have a column value *less* than it, and the cell_node to its right, which must have a column value *greater* than it. In other words, within a single row, the column values of `cell_nodes` increase from left to right. If a `cell_node` doesn't have a `cell_node` to its left or right, then the corresponding pointer should be equal to `nullptr`.

The second type of node is row_node defined by this struct:

```
struct row_node{
     row_node* down;
```

```
    cell_node* cell;
};
```

This node has two jobs:

1. It will point at the first `cell_node` in a row. In other words, the cell of the first non-zero in this row (leftmost `cell_node` in the row).
2. It will point at the `row_node` of the row below it, or `nullptr` if there is no row below it.

These nodes don't store non-zeros themselves but serve as a way to connect rows together.

A matrix will be pointed at by a single `head` pointer that points at the very first `row_node`, i.e., the `row_node` pointing at row 0.

Functions to Implement

In this homework, you will be given a main.cpp file containing a "main" function that executes some code, including a few functions using sparse matrices. However, these functions will not be implemented. Your job is to implement these functions such that the main function runs correctly.

Note that the main.cpp file includes a function that will check the structure of a sparse matrix for correctness. This function is already implemented for you, you don't have to write it yourself.

The functions you must implement are the following:

`row_node* read_matrix(string filename, int& num_rows, int& num_cols)`

This function will take the name of a text file containing a matrix, read the file into a sparse matrix, and return a pointer at its head. In addition, it will set the integer `num_rows` to the number of rows in the matrix, and the integer `num_cols` to the number of columns in the matrix.

The matrix file structure will be identical to the one used in the last homework. You may refer to the previous homework document for the exact definition of the file format.

Important: you may assume that the file structure will be correct. In other words, ***you don't have to do any error checking for the input file.***

Additionally, you may make the following assumptions about the input matrices:

1. They will only contain integers (positive, 0, and negative). No strings, characters, or floating point numbers.
2. They will always have at least one row and at least one column.

`void print_matrix(row_node* head, int num_rows, int num_cols)`

Takes the head pointer at a matrix and prints the matrix to standard output. It will print each row in a separate line and will split the values in each row by a single space.

However, the function must print all the elements *including the zeros*. For example, for the matrix in Figure 2, the printed output should be:

```
0 0 0 0 12 0
-1 0 0 0 44 0
4 0 0 11 0 5
0 0 0 6 0 0
0 4 8 0 0 -1
0 0 0 19 0 -4
```

`void delete_matrix(row_node* head)`

Deletes a matrix pointed at by `head`. Not that it must delete *every node*. The cell nodes and the row nodes.

`bool is_symmetric(row_node* head, int num_rows, int num_columns)`

This function will check whether the matrix pointed at by `head` is symmetric. A matrix is symmetric if:

1. It is square (number of rows == number of columns), and
2. Element at row i, column j == element at row j, column i for every i and j.

As an example, the following are two matrices, one that is symmetric and one that is a non-symmetric matrix:

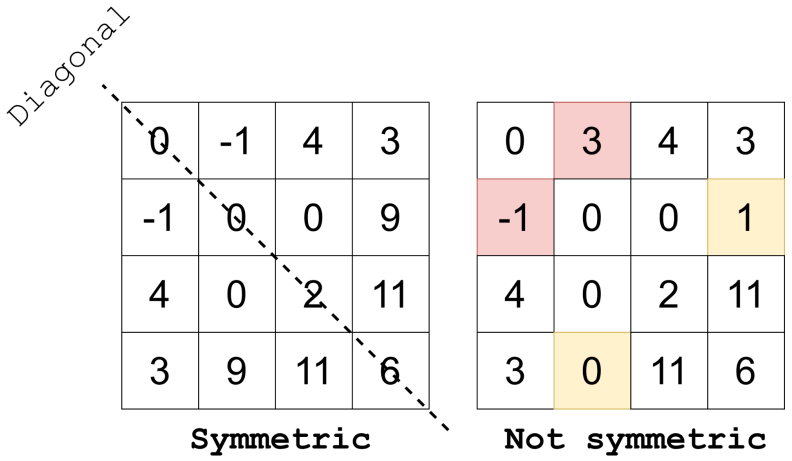

Figure 3: The left matrix is symmetric. The right matrix is not symmetric.

A nice way to understand symmetry is to imagine a mirror down the diagonal of the matrix (shown with zebra lines in the left figure). A symmetric matrix looks as if the triangle above the diagonal is a mirrored version of the triangle below the diagonal.

The right matrix is almost symmetric, however, it has four problematic cells shown in red and yellow. As you can see, the problem is that `Element[0][1] != Element[1][0]`, and `Element[3][1]!=Element[1][3]`.

`row_node* transpose(row_node* head, int num_row, int num_columns)`

This function takes a matrix and finds its transpose. The transpose of a matrix is a matrix generated if we take every row in the original matrix in order and turn it into a column in the new matrix, i.e., row 0 in the original matrix becomes column 0 in the transpose matrix, and row 1 in the original matrix becomes column 1 in the transpose matrix, etc.

Figure 4 is an example of a transpose operation. The right matrix is the transpose of the left matrix.

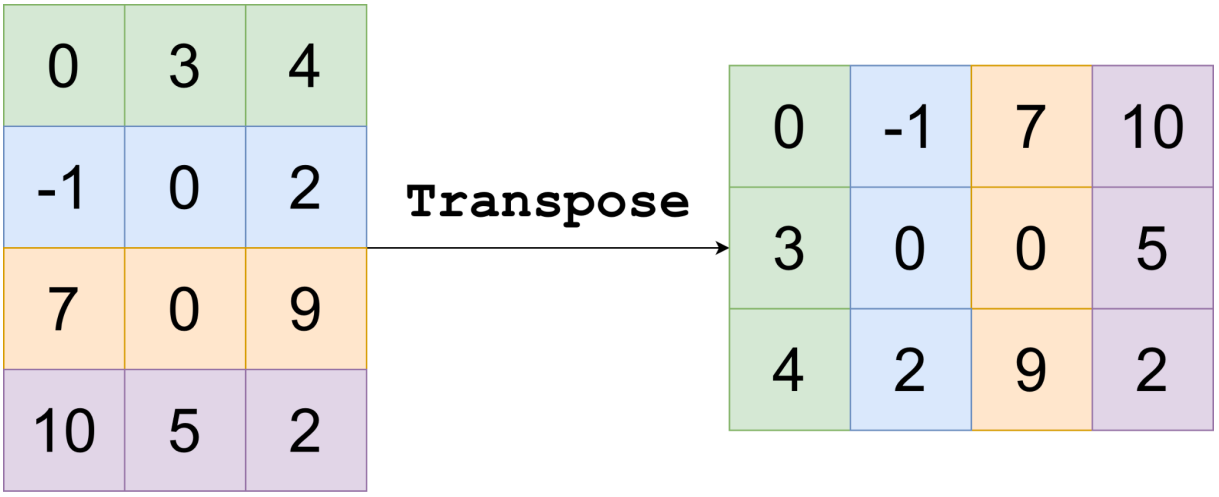

Figure 4: the matrix to the left is transposed to create the matrix on the right.

Mathematically, the transpose of a matrix A is a new matrix B in which the element A[i][j] becomes the element B[j][i].

`row_node* add_matrices(row_node* lhs, row_node* rhs, int num_rows, int num_cols)`

Takes two matrices, one pointed at by `lhs`, and another pointed at by `rhs`, and returns a new matrix that is the result of adding these two matrices together. Note that `lhs` and `rhs` must have the same number of rows and must have the same number of columns. You don't have to check this when you implement this function. We guarantee to you that the function will only be called if the two matrices have the same dimensions (in other words, the fact that `lhs` and `rhs` have the same dimensions is a "precondition" of this function).

Adding matrices is very simple: it will create a result matrix with the same size as the left and right matrices. Every cell in the new matrix will be equal to the addition of the cells in the same position in `lhs` and `rhs`.

So, if our matrices were represented in the vector of vector representation, we would calculate the addition of them as follows:

```
vector<vector<int>>add_matrices(vector<vector<int>>lhs,   vector<vector<int>>   rhs,
int num_rows, int num_columns){
  vector<vector<int>> result(num_rows, vector<int>(num_columns));
  for (int i =0; i < num_rows; i++)
    for (int j =0; j < num_columns; j++)
      result[i][j] = lhs[i][j] + rhs[i][j];
  return result;
```

```
}
```

You're going to implement a function that does the same thing as the one above, but that takes sparse matrices, and returns the result as a different sparse matrix as well.

Program flow

In the homework package, you will find two code files:

1. `main.cpp`: this file contains two things, the main program that will execute the functions you are going to implement, and the correctness checker function that can check if your matrix is structured correctly.
2. `sparse_matrix_2d_linked_list.h`: a header file containing the struct definitions of `row_node` and `cell_node`, as well as the function signatures of the functions you will implement (a function signature is just the name of the function, its return type, and its parameters.)

Your job is to implement the functions in the header file in a *different* .cpp file. In this file, you will include the header file "sparse_matrix_2d_linked_list.h" and you will implement all the functions mentioned in the previous section. For instructions on how to name this file, please refer to the "Submission rules" section at the end of the document).

Important: you are not allowed to change anything in the `main.cpp` and `sparse_matrix_2d_linked_list.h` files. Your code should work with the exact same files we provide. Of course, during the development of the code, you may comment out some code or try different things to help you write your code. However, these changes cannot be made to the code that is submitted.

When grading your code, we are only going to take the .cpp file with function implementations that you wrote and compile it against the `main.cpp` and `sparse_matrix_2d_linked_list.h` files provided in the homework package. We won't use the main that you submit with your submission.

Important rules:

1. **you may not use vectors, arrays, or any other type of container besides the given data structure in your code.**
2. **You may not use any external libraries.**

Violating either of these rules will give your homework an instant 0.

The following is the main function we provide. You will find that we are using some functions from the `chrono` library. These are for measuring how long the `add_matrices` and `is_symmetric` function calls take.

```
int main() {
    /////// Timing data ///////
    chrono::time_point<chrono::system_clock> start, end;
    chrono::duration<double> total_time;
    ////////////////////////////
```

```
    string filename1, filename2;
    int m1_rows, m1_cols, m2_rows, m2_cols;
    cout << "Matrix 1 filename: ";
    cin >>filename1;

    cout << "Matrix 2 filename: ";
    cin >>filename2;

    row_node* m1 = read_matrix(filename1, m1_rows, m1_cols);
    structure_is_okay(m1, m1_rows, m1_cols);

    row_node* m2 = read_matrix(filename2, m2_rows, m2_cols);
    structure_is_okay(m2, m2_rows, m2_cols);

    cout << "m1\n";
    print_matrix(m1, m1_rows, m1_cols);

    cout << "m2\n";
    print_matrix(m2, m2_rows, m2_cols);

    if (m1_rows == m2_rows && m1_cols == m2_cols){
        start = chrono::system_clock::now();
        row_node* m3 = add_matrices(m1, m2, m1_rows, m1_cols);
        end = chrono::system_clock::now();
        total_time = end - start;
        cout << "Time to add matrices: " << total_time.count() << "
seconds\n";

        structure_is_okay(m2, m2_rows, m2_cols);

        cout << "m3\n";
        print_matrix(m3, m2_rows, m2_cols);
        delete_matrix(m3);
    }
    start = chrono::system_clock::now();
    bool symmetric = is_symmetric(m1, m1_rows, m1_cols);
    end = chrono::system_clock::now();
    total_time = end - start;
    cout << "Time to check if m1 is symmetric: " <<
total_time.count() << " seconds\n";

    if (symmetric)
```

```
        cout << "Matrix m1 is symmetric.\n";
    else
        cout << "Matrix m1 is not symmetric.\n";

    start = chrono::system_clock::now();
    symmetric = is_symmetric(m2, m2_rows, m2_cols);
    end = chrono::system_clock::now();
    total_time = end - start;
    cout << "Time to check if m2 is symmetric: " <<
total_time.count() << " seconds\n";

    if (symmetric)
        cout << "Matrix m2 is symmetric.\n";
    else
        cout << "Matrix m2 is not symmetric.\n";

    row_node* m1_transpose = transpose(m1, m1_rows, m1_cols);
    cout << "m1 transposed\n";
    print_matrix(m1_transpose, m1_cols, m1_rows);
    delete_matrix(m1_transpose);

    row_node* m2_transpose = transpose(m2, m2_rows, m2_cols);
    cout << "m2 transposed\n";
    print_matrix(m2_transpose, m2_cols, m2_rows);
    delete_matrix(m2_transpose);

    delete_matrix(m1);
    delete_matrix(m2);
    return 0;
}
```

It starts out by taking the filenames from the user. Then it reads these files into matrix objects and returns the head pointer to each object. Notice how after each read, the matrices are passed to the structure-checking function.

Afterward, if the matrices have the same shape (same rows and same columns), the two matrices are added into a new matrix. That matrix is also run through the structure-checking function, printed, and then deleted.

Following that, both matrices are checked to see if they are symmetric or not. Then, both matrices are transposed.

Finally, the matrices are deleted.

Sample Runs

Sample run 1

m1.txt

```
3 3
1 1 2
1 1 3
2 3 1
```

m2.txt

```
3 3
0 0 1
2 3 1
0 0 0
```

Console output (italicized and bolded text is input by the user):

```
Matrix 1 filename: inputs/tc1/m1.txt
Matrix 2 filename: inputs/tc1/m2.txt
m1
1 1 2
1 1 3
2 3 1
m2
0 0 1
2 3 1
0 0 0
Time to add matrices: 3.6e-06 seconds
m3
1 1 3
3 4 4
2 3 1
Time to check if m1 is symmetric: 4e-07 seconds
Matrix m1 is symmetric.
Time to check if m2 is symmetric: 3e-07 seconds
Matrix m2 is not symmetric.
Time to transpose m1: 2.1e-06 seconds
m1 transposed
1 1 2
1 1 3
2 3 1
Time to transpose m2: 1.2e-06 seconds
m2 transposed
0 2 0
0 3 0
1 1 0
```

Sample run 2

m1.txt

```
3 4
1 2 0 0
4 0 0 3
0 4 0 0
```

m2.txt

```
1 1
1
```

Console output (italicized and bolded text is input by the user):

```
Matrix 1 filename: inputs/tc2/m1.txt
Matrix 2 filename: inputs/tc2/m2.txt
m1
1 2 0 0
4 0 0 3
0 4 0 0
m2
1
Time to check if m1 is symmetric: 3e-07 seconds
Matrix m1 is not symmetric.
Time to check if m2 is symmetric: 2e-07 seconds
Matrix m2 is symmetric.
Time to transpose m1: 8.9e-06 seconds
m1 transposed
1 4 0
2 0 4
0 0 0
0 3 0
Time to transpose m2: 1.6e-06 seconds
m2 transposed
1
```

Sample run 3

m1.txt

```
3 3
0 4 1
4 0 4
1 4 0
```

m2.txt

```
3 3
0 -4 -1
-4 0 -4
-1 -4 0
```

Console output (italicized and bolded text is input by the user):

```
Matrix 1 filename: inputs/tc3/m1.txt
Matrix 2 filename: inputs/tc3/m2.txt
m1
0 4 1
4 0 4
1 4 0
m2
0 -4 -1
-4 0 -4
-1 -4 0
Time to add matrices: 3.9e-06 seconds
m3
0 0 0
0 0 0
0 0 0
Time to check if m1 is symmetric: 6e-07 seconds
Matrix m1 is symmetric.
Time to check if m2 is symmetric: 6e-07 seconds
Matrix m2 is symmetric.
Time to transpose m1: 1.7e-06 seconds
m1 transposed
0 4 1
4 0 4
1 4 0
Time to transpose m2: 1.2e-06 seconds
m2 transposed
0 -4 -1
-4 0 -4
-1 -4 0
```

Submission rules

In order to get full credit, your programs must be efficient and well presented, the presence of any redundant computation or bad indentation, missing comments, or irrelevant comments are going to decrease your grades. You also have to use understandable identifier names and informative prompts. Modularity is also important; you have to use functions wherever needed and appropriate.

When we grade your homework we pay attention to these issues. Moreover, in order to observe the real performance of your codes, we will run your programs in Release mode and we will test your programs with very large test cases.

What and where to submit (PLEASE READ, IMPORTANT)

You must write your solution in C++. It'd be a good idea to write your name and last name in the program (as a comment line of course). Submission guidelines are below. Some parts of the grading process are automatic. Students are expected to strictly follow these guidelines in order to have a smooth grading process. If you do not follow these guidelines, depending on the severity of the problem created during the grading process, 5 or more penalty points are to be deducted from the grade.

Name your .cpp file that contains the implementations of the functions described in the section “Functions to Implement” as follows:

```
SUCourseUserName_YourLastname_YourName_HWnumber.cpp
```

Your SUCourse user name is your SUNet username that is used for your sabanciuniv e-mails. Do NOT use any spaces, non-ASCII and Turkish characters in the file name. For example, if your SUCourse username is cago, your first name is Taha Çağlayan, and your last name is Özbugsizkodyazaroğlu, then the file name must be:

```
cago_Ozbugsızkodyazaroglu_TahaCaglayan_1.cpp
```

If your solution contains other code files, you don’t have to change the other files’ names. Only the file with the function implementations needs to have the naming convention above.

You shouldn’t add any other files besides code files to the submission folder. In other words, don’t add the example inputs or any example outputs to the submission. You may add the files that we provide (main and header files) to the submission package, but we won’t use them when grading anyway.

Place all of your code files inside a folder named with the same naming convention shown above (without the .cpp extension, of course). So, the same student above would place all of his code inside a folder named:

```
cago_Ozbugsızkodyazaroglu_TahaCaglayan_1
```

Compress this folder using a compression program such as WinZip or WinRAR. Please use "zip" compression. "rar", “7z” or any other compression mechanisms are NOT allowed. Our homework processing system only works with zip files. Therefore, make sure that the resulting compressed file has a zip extension. Check that your compressed file opens up correctly and it contains all your code files. You will receive no credits if your compressed folder does not expand or it does not contain the correct files. The name of the zip file follows the same convention. The zip file for the homework submission by the student mentioned above would be:

```
cago_Ozbugsızkodyazaroglu_TahaCaglayan_1
```

Submit via SUCourse ONLY! You will receive no credits if you submit by other means (e-mail, paper, etc.). Successful submission is one of the requirements of the homework. If for some reason you cannot successfully submit your homework and we cannot grade it, your grade will be 0.

Good Luck!

Amro