**CSEL- – Algorithm Design Lab**

**Laboratory 01**: **C++ STL and Java Collection Class.**

**Submission Due**: End of laboratory class, submit the file on Google Classroom/Google Form before the beginning of next laboratory class. The class teacher will provide solutions in Java Programming. However, you can solve the problem using any programming language (Java/C++/ Python).

**Total Marks** = X marks for Y weeks

Marks will be given only to students who attend and participate during 2/3-hour laboratory class. Submission on Google Classroom/Google Form is mandatory as evidence of participation.

**Description of the laboratory exercise**:

**Learning Outcome:**

By the end of this lab, you’ll know how to:

1. C++ STL
2. Java Collection Class
3. Graph

**C++ STL:**  To learn C++ STL, follow the links below

1. <https://sites.google.com/site/smilitude/stl>
2. [Zobayer’s STL](https://zobayer2009.wordpress.com/2011/02/16/c-stl-vector/?fbclid=IwAR1HfMgquox2dzcu1Y0NVyK7y9jsSY0z14Y3ZCOMYC__zp8m3Fv-C9aooSI)
3. [Bangla Programming Resource](https://github.com/me-shaon/bangla-programming-resources)
4. [Algorithm Resources in Bangla](https://www.shafaetsplanet.com/)

**Java Collection Class**

Chapter 20, Introduction to Java Programming by Y. Daniel Liang.

**Introduction to Collections and Class ArrayList**

The Java API provides several predefined data structures, called **collections**, used to store groups of related objects in memory. These classes provide efficient methods that organize, store and retrieve your data *without* requiring knowledge of how the data is being *stored*. This reduces application-development time. You’ve used arrays to store sequences of objects. Arrays do not automatically change their size at execution time to accommodate additional elements. The collection class

**ArrayList<T>** (package java.util) provides a convenient solution to this problem—it can *dynamically* change its size to accommodate more elements. The T (by convention) is a *placeholder*—when declaring a new ArrayList, replace it with the type of elements that you want the ArrayList to hold. For example,

ArrayList<String> list;

declares list as an ArrayList collection that can store only Strings. Classes with this kind of placeholder that can be used with any type are called **generic classes**. *Only* *nonprimitive types can be used to declare variables and create objects of generic classes*. However, Java provides a mechanism—known as *boxing*—that allows primitive values to be wrapped as objects for use with generic classes. So, for example,

ArrayList<Integer> integers;

declares integers as an ArrayList that can store only Integers. When you place an int value into an ArrayList<Integer>, the int value is *boxed* (wrapped) as an Integer object, and when you get an Integer object from an ArrayList<Integer>, then assign the object to an int variable, the int value inside the object is *unboxed* (unwrapped).

Graphical user interface, text, application

Description automatically generated

**Example for practice:**

|  |
| --- |
|  |

The **ArrayList** Class

*An* **ArrayList** *object can be used to store a list of objects.* Now we are ready to introduce a very useful class for storing objects. You can create an array to store objects. But, once the array is created, its size is fixed. Java provides the **ArrayList.**

ArrayList<String> cities = **new** ArrayList<String>();

Text

Description automatically generated with medium confidence

*The* **Collection** *interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.*

The Java Collections Framework supports two types of containers:

■ One for storing a collection of elements is simply called a *collection*.

■ The other, for storing key/value pairs, is called a *map*.

Maps are efficient data structures for quickly searching an element using a key. We will introduce maps in the next chapter. Now we turn our attention to the following collections.

■ **Set**s store a group of nonduplicate elements.

■ **List**s store an ordered collection of elements.

■ **Stack**s store objects that are processed in a last-in, first-out fashion.

■ **Queue**s store objects that are processed in a first-in, first-out fashion.

■ **PriorityQueue**s store objects that are processed in the order of their priorities.

Queues and Priority Queues

*In a priority queue, the element with the highest priority is removed first.*

A *queue* is a first-in, first-out data structure. Elements are appended to the end of the queue

and are removed from the beginning of the queue. In a *priority queue*, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

The **Queue** Interface

The **Queue** interface extends **java.util.Collection** with additional insertion, extraction, and inspection operations

Graphical user interface, application

Description automatically generated

**Deque** and **LinkedList**

The **LinkedList** class implements the **Deque** interface, which extends the **Queue** interface. Therefore, you can use **LinkedList** to create a queue. **LinkedList** is ideal for queue operations because it is efficient for inserting and removing elements from both ends of a list.

**Deque** supports element insertion and removal at both ends. The name *deque* is short for “double-ended queue” and is usually pronounced “deck.” The **Deque** interface extends **Queue** with additional methods for inserting and removing elements from both ends of the queue. The methods **addFirst(e)**, **removeFirst()**, **addLast(e)**, **removeLast()**, **getFirst()**, and **getLast()** are defined in the **Deque** interface.

Example:

Graphical user interface, text, application, chat or text message

Description automatically generated

**Arrays**

An array is a group of variables (called **elements** or **components**) containing values that all have the *same* type. Arrays are *objects*, so they’re considered *reference types*.

The *elements* of an array can be either *primitive types* or *reference types*

To refer to a particular element in an array, we specify the *name* of the reference to the array and the *position number* of the element in the array. The position number of the element is called the element’s **index** or **subscript**.

*Once an array is created, its size is fixed. An array reference variable is used to access the elements in an array using an* index*.*

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as **number0**, **number1**, . . . , and **number99**, you declare one array variable such as **numbers** and use **numbers[0]**, **numbers[1]**, . . . , and **numbers[99]** to represent individual variables. This section introduces how to declare array variables, create arrays, and process arrays using indexes.

**Declaring Array Variables**

To use an array in a program, you must declare a variable to reference the array and specify the array’s *element type*. Here is the syntax for declaring an array variable:

elementType[] arrayRefVar;

The **elementType** can be any data type, and all elements in the array will have the same data type. For example, the following code declares a variable **myList** that references an array of double elements.

**double**[] myList;

**Creating Arrays**

Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is **null**.

You cannot assign elements to an array unless it has already been created. After an array variable is declared, you can create an array by using the **new** operator and assign its reference to the variable with the following syntax:

arrayRefVar = **new** elementType[arraySize];

This statement does two things: (1) it creates an array using **new elementType[arraySize]**; (2) it assigns the reference of the newly created array to the variable **arrayRefVar**. Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:

elementType[] arrayRefVar = **new** elementType[arraySize];

or

elementType arrayRefVar[] = **new** elementType[arraySize];

Here is an example of such a statement:

**double**[] myList = **new double**[**10**];

This statement declares an array variable, **myList**, creates an array of ten elements of **double** type, and assigns its reference to **myList**. To assign values to the elements, use the syntax:

arrayRefVar[index] = value;

For example, the following code initializes the array.

myList[**0**] = **5.6**;

myList[**1**] = **4.5**;

myList[**2**] = **3.3**;

myList[**3**] = **13.2**;

myList[**4**] = **4.0**;

myList[**5**] = **34.33**;

myList[**6**] = **34.0**;

myList[**7**] = **45.45**;

myList[**8**] = **99.993**;

myList[**9**] = **11123**;

Table

Description automatically generated

**Array Size and Default Values**

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using **arrayRefVar.length**. For example, **myList.length** is **10**. When an array is created, its elements are assigned the default value of **0** for the numeric primitive data types, **\u0000** for **char** types, and **false** for **boolean** types.

**Array Initializers**

Java has a shorthand notation, known as the *array initializer*, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:

elementType[] arrayRefVar = {value0, value1, ..., value*k*};

For example, the statement

**double**[] myList = {**1.9**, **2.9**, **3.4**, **3.5**};

declares, creates, and initializes the array **myList** with four elements, which is equivalent to the following statements:

**double**[] myList = **new double**[**4**];

myList[**0**] = **1.9**;

myList[**1**] = **2.9**;

**Processing Arrays**

When processing array elements, you will often use a **for** loop—for two reasons:

* All of the elements in an array are of the same type. They are evenly processed in the same fashion repeatedly using a loop.
* Since the size of the array is known, it is natural to use a **for** loop.

Assume the array is created as follows:

**double**[] myList = **new double**[**10**];

The following are some examples of processing arrays.

1. *Initializing arrays with input values:* The following loop initializes the array **myList** with user input values.

java.util.Scanner input = **new** java.util.Scanner(System.in);

System.out.print(**"Enter "** + myList.length + **" values: "**);

**for** (**int** i = **0**; i < myList.length; i++)

myList[i] = input.nextDouble();

2. *Initializing arrays with random values:* The following loop initializes the array **myList** with random values between **0.0** and **100.0**, but less than **100.0**.

**for** (**int** i = **0**; i < myList.length; i++) {

myList[i] = Math.random() \* **100**;

}

3. *Displaying arrays:* To print an array, you have to print each element in the array using a

loop like the following:

**for** (**int** i = **0**; i < myList.length; i++) {

System.out.print(myList[i] + **" "**);

}

4. *Summing all elements:* Use a variable named **total** to store the sum. Initially **total** is **0**. Add each element in the array to **total** using a loop like this:

**double** total = **0**;

**for** (**int** i = **0**; i < myList.length; i++) {

total += myList[i];

}

5. *Finding the largest element:* Use a variable named **max** to store the largest element. Initially **max** is **myList[0]**. To find the largest element in the array **myList**, compare each element with **max**, and update **max** if the element is greater than **max**.

**double** max = myList[**0**];

**for** (**int** i = **1**; i < myList.length; i++) {

**if** (myList[i] > max) max = myList[i];

}

1. *Finding the smallest index of the largest element:* Often you need to locate the largest element in an array. If an array has multiple elements with the same largest value, find the smallest index of such an element. Suppose the array **myList** is {**1**, **5**, **3**, **4**, **5**, **5**}. The largest element is **5** and the smallest index for **5** is **1**. Use a variable named **max** to store the largest element and a variable named **indexOfMax** to denote the index of the largest element. Initially **max** is **myList[0]**, and **indexOfMax** is **0**. Compare each element in **myList** with **max**, and update **max** and **indexOfMax** if the element is greater than **max**.

**double** max = myList[**0**];

**int** indexOfMax = **0**;

**for** (**int** i = **1**; i < myList.length; i++) {

**if** (myList[i] > max) {

max = myList[i];

indexOfMax = i;

}

}

**Foreach Loops**

Java supports a convenient **for** loop, known as a *foreach loop*, which enables you to traversethe array sequentially without using an index variable. For example, the following code displaysall the elements in the array **myList**:

**for** (**double** e: myList) {

System.out.println(e);

}

You can read the code as “for each element **e** in **myList**, do the following.” Note that the variable, **e**, must be declared as the same type as the elements in **myList**. In general, the syntax for a foreach loop is

**for** (elementType element: arrayRefVar) {

// Process the element}

You still have to use an index variable if you wish to traverse the array in a different order or

change the elements in the array.

**Passing Arrays to Methods**

*When passing an array to a method, the reference of the array is passed to the method.* Just as you can pass primitive type values to methods, you can also pass arrays to methods.For example, the following method displays the elements in an **int** array:

**public static void** printArray(**int**[] array) {

**for** (**int** i = **0**; i < array.length; i++) {

System.out.print(array[i] + **" "**);

}

}

You can invoke it by passing an array. For example, the following statement invokes the **printArray** method to display **3**, **1**, **2**, **6**, **4**, and **2**.

printArray(**new int**[]{**3**, **1**, **2**, **6**, **4**, **2**});

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

* For an argument of a primitive type, the argument’s value is passed.
* For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as

*pass-by-sharing*, that is, the array in the method is the same as the array being passed. Thus, if you change the array in the method, you will see the change outside the method. Take the following code, for example:

Example for practice:

|  |
| --- |
| Text  Description automatically generated |

Example for practice:

|  |
| --- |
| Text  Description automatically generated |

**Returning an Array from a Method**

*When a method returns an array, the reference of the array is returned.* You can pass arrays when invoking a method. A method may also return an array. For example,the following method returns an array that is the reversal of another array.

Diagram

Description automatically generated

**Counting the Occurrences of Each Letter**

*This section presents a program to count the occurrences of each letter in an array of characters.*

**Example for practice:**

|  |
| --- |
| Graphical user interface, text, application, chat or text message  Description automatically generatedText  Description automatically generatedGraphical user interface, text, application  Description automatically generated |

**The Arrays Class**

*The* **java.util.Arrays** *class contains useful methods for common array operations such as sorting and searching.*

The **java.util.Arrays** class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types. You can use the **sort** or **parallelSort** method to sort a whole array or a partial array.

For example, the following code sorts an array of numbers and an array of characters.

**double**[] numbers = {**6.0**, **4.4**, **1.9**, **2.9**, **3.4**, **3.5**};

java.util.Arrays.sort(numbers); // Sort the whole array

java.util.Arrays.parallelSort(numbers); // Sort the whole array

**char**[] chars = {**'a'**, **'A'**, **'4'**, **'F'**, **'D'**, **'P'**};

java.util.Arrays.sort(chars, **1**, **3**); // Sort part of the array

java.util.Arrays.parallelSort(chars, **1**, **3**); // Sort part of the array

Invoking **sort(numbers)** sorts the whole array **numbers**. Invoking **sort(chars, 1, 3)** sorts a partial array from **chars[1]** to **chars[3-1]**. **parallelSort** is more efficient ifyour computer has multiple processors.You can use the **binarySearch** method to search for a key in an array. The array must be presortedin increasing order. If the key is not in the array, the method returns **–(insertionIndex + 1)**. For example, the following code searches the keys in an array of integers and an arrayof characters.

**int**[] list = {**2**, **4**, **7**, **10**, **11**, **45**, **50**, **59**, **60**, **66**, **69**, **70**, **79**};

System.out.println(**"1. Index is "** +

java.util.Arrays.binarySearch(list, **11**));

System.out.println(**"2. Index is "** +

java.util.Arrays.binarySearch(list, **12**));

**char**[] chars = {**'a'**, **'c'**, **'g'**, **'x'**, **'y'**, **'z'**};

System.out.println(**"3. Index is "** +

java.util.Arrays.binarySearch(chars, **'a'**));

System.out.println(**"4. Index is "** +

java.util.Arrays.binarySearch(chars, **'t'**));

The output of the preceding code is

1. Index is 4

2. Index is -6

3. Index is 0

4. Index is -4

You can use the **equals** method to check whether two arrays are strictly equal. Two arrays are strictly equal if their corresponding elements are the same. In the following code, **list1** and **list2** are equal, but **list2** and **list3** are not.

**int**[] list1 = {**2**, **4**, **7**, **10**};

**int**[] list2 = {**2**, **4**, **7**, **10**};

**int**[] list3 = {**4**, **2**, **7**, **10**};

System.out.println(java.util.Arrays.equals(list1, list2)); // true

System.out.println(java.util.Arrays.equals(list2, list3)); // false

You can use the **fill** method to fill in all or part of the array. For example, the following code fills **list1** with **5** and fills **8** into elements **list2[1]** through **list2[5-1]**.

**int**[] list1 = {**2**, **4**, **7**, **10**};

**int**[] list2 = {**2**, **4**, **7**, **7**, **7**, **10**};

java.util.Arrays.fill(list1, **5**); // Fill 5 to the whole array

java.util.Arrays.fill(list2, **1**, **5**, **8**); // Fill 8 to a partial array

You can also use the **toString** method to return a string that represents all elements in the array. This is a quick and simple way to display all elements in the array. For example, the following code

**int**[] list = {**2**, **4**, **7**, **10**};

System.out.println(Arrays.toString(list));

displays **[2, 4, 7, 10]**.

**Data in a table or a matrix can be represented using a two-dimensional array.**

You can use a two-dimensional array to store a matrix or a table. For example, the following table that lists the distances between cities can be stored using a two-dimensional array named distances.

Table

Description automatically generated

Two-Dimensional Array Basics

An element in a two-dimensional array is accessed through a row and column index. How do you declare a variable for two-dimensional arrays? How do you create a two-dimensional array? How do you access elements in a two-dimensional array? This section addresses these issues.

Declaring Variables of Two-Dimensional Arrays and Creating

Two-Dimensional Arrays

The syntax for declaring a two-dimensional array is:

elementType[][] arrayRefVar;

or

elementType arrayRefVar[][]; // Allowed, but not preferred

As an example, here is how you would declare a two-dimensional array variable **matrix** of **int** values:

**int**[][] matrix;

or

**int** matrix[][]; // This style is allowed, but not preferred

You can create a two-dimensional array of 5-by-5 **int** values and assign it to **matrix** using this syntax:

matrix = **new int**[**5**][**5**];

Two subscripts are used in a two-dimensional array, one for the row and the other for the

column. As in a one-dimensional array, the index for each subscript is of the **int** type and

starts from **0**

To assign the value **7** to a specific element at row **2** and column **1**, as shown in Figure b, you can use the following syntax:

matrix[**2**][**1**] = **7**;

A picture containing text, remote, game

Description automatically generated

You can also use an array initializer to declare, create, and initialize a two-dimensional array. For example, the following code in (a) creates an array with the specified initial values, as shown in Figure 8.1c. This is equivalent to the code in (b).

Text

Description automatically generated

Obtaining the Lengths of Two-Dimensional Arrays

A two-dimensional array is actually an array in which each element is a one-dimensional array. The length of an array **x** is the number of elements in the array, which can be obtained using **x.length**. **x[0]**, **x[1]**, . . . , and **x[x.length-1]** are arrays. Their lengths can be obtained using **x[0].length**, **x[1].length**, . . . , and **x[x.length-1].length**.

For example, suppose **x = new int[3][4]**, **x[0]**, **x[1]**, and **x[2]** are one-dimensional arrays and each contains four elements, as shown in Figure 8.2. **x.length** is **3**, and **x[0].length**, **x[1].length**, and **x[2].length** are **4**.

Table

Description automatically generated

Ragged Arrays

Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths. An array of this kind is known as a *ragged array*. Here is an example of creating a ragged array:

Chart

Description automatically generated with medium confidence

As you can see, **triangleArray[0].length** is 5, **triangleArray[1].length** is 4, **triangleArray[2].length** is 3, **triangleArray[3].length** is 2, and **triangle-** **Array[4].length** is 1.

If you don’t know the values in a ragged array in advance, but do know the sizes—say, the same as before—you can create a ragged array using the following syntax:

**int**[][] triangleArray = **new int**[**5**][];

triangleArray[**0**] = **new int**[**5**];

triangleArray[**1**] = **new int**[**4**];

triangleArray[**2**] = **new int**[**3**];

triangleArray[**3**] = **new int**[**2**];

triangleArray[**4**] = **new int**[**1**];

You can now assign values to the array. For example,

triangleArray[**0**][**3**] = **50**;

triangleArray[**4**][**0**] = **45**;

Processing Two-Dimensional Arrays

*Nested* **for** *loops are often used to process a two-dimensional array.*

Suppose an array **matrix** is created as follows:

**int**[][] matrix = **new int**[**10**][**10**];

The following are some examples of processing two-dimensional arrays.

1. *Initializing arrays with input values.* The following loop initializes the array with user

input values:

java.util.Scanner input = **new** Scanner(System.in);

System.out.println(**"Enter "** + matrix.length + **" rows and "** +

matrix[**0**].length + **" columns: "**);

**for** (**int** row = **0**; row < matrix.length; row++) {

**for** (**int** column = **0**; column < matrix[row].length; column++) {

matrix[row][column] = input.nextInt();

}

}

2. *Initializing arrays with random values.* The following loop initializes the array with random values between **0** and **99**:

**for** (**int** row = **0**; row < matrix.length; row++) {

**for** (**int** column = **0**; column < matrix[row].length; column++) {

matrix[row][column] = (**int**)(Math.random() \* **100**);

}

}

3. *Printing arrays.* To print a two-dimensional array, you have to print each element in the array using a loop like the following:

**for** (**int** row = **0**; row < matrix.length; row++) {

**for** (**int** column = **0**; column < matrix[row].length; column++) {

System.out.print(matrix[row][column] + **" "**);

}

System.out.println();

}

4. *Summing all elements.* Use a variable named **total** to store the sum. Initially **total** is **0**. Add each element in the array to **total** using a loop like this:

**int** total = **0**;

**for** (**int** row = **0**; row < matrix.length; row++) {

**for** (**int** column = **0**; column < matrix[row].length; column++) {

total += matrix[row][column];

}

}

5. *Summing elements by column.* For each column, use a variable named **total** to store its sum. Add each element in the column to **total** using a loop like this:

**for** (**int** column = **0**; column < matrix[**0**].length; column++) {

**int** total = **0**;

**for** (**int** row = **0**; row < matrix.length; row++)

total += matrix[row][column];

System.out.println(**"Sum for column "** + column + **" is "**

+ total);

}

6. *Which row has the largest sum?* Use variables **maxRow** and **indexOfMaxRow** to track the largest sum and index of the row. For each row, compute its sum and update **maxRow** and **indexOfMaxRow** if the new sum is greater.

**int** maxRow = **0**;

**int** indexOfMaxRow = **0**;

// Get sum of the first row in maxRow

**for** (**int** column = **0**; column < matrix[**0**].length; column++) {

maxRow += matrix[**0**][column];

}

**for** (**int** row = **1**; row < matrix.length; row++) {

**int** totalOfThisRow = **0**;

**for** (**int** column = **0**; column < matrix[row].length; column++)

totalOfThisRow += matrix[row][column];

**if** (totalOfThisRow > maxRow) {

maxRow = totalOfThisRow;

indexOfMaxRow = row;

}

}

System.out.println(**"Row "** + indexOfMaxRow

+ **" has the maximum sum of "** + maxRow);

Passing Two-Dimensional Arrays to Methods

*When passing a two-dimensional array to a method, the reference of the array is passed to the method.*

You can pass a two-dimensional array to a method just as you pass a one-dimensional array. You can also return an array from a method.

Example for practice:

|  |
| --- |
| Text  Description automatically generated |

What is Graph:

G = (V,E)

Graph is a collection of nodes or vertices (V) and edges(E) between them. We can traverse these nodes using the edges. These edges might be weighted or non-weighted.

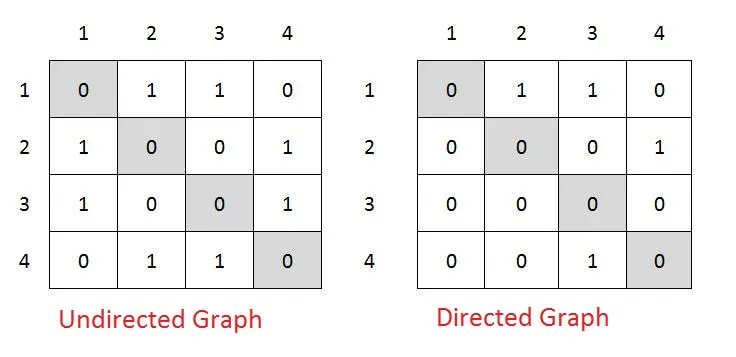
Terminology and Representation of Graph:

[Click me](https://www.techiedelight.com/terminology-and-representations-of-graphs/) to read terminology and representation of graph.

Adjacency Matrix:

Adjacency Matrix is 2-Dimensional Array which has the size VxV, where V are the number of vertices in the graph. See the example below, the Adjacency matrix for the graph shown above.

* adjMaxtrix[i][j] = 1 when there is edge between Vertex i and Vertex j, else 0.
* It’s easy to implement because removing and adding an edge takes only O(1) time.
* But the drawback is that it takes O(V2) space even though there are very less edges in the graph.



Example for practice: This example shows you how to input graph in a program.

|  |
| --- |
|  |
| Graphical user interface, text, application  Description automatically generated |

Now solve the following problem.

1. Is there a direct link between A and B?
2. What is the indegree and outdegree for a vertex A?
3. How many nodes are directly connected to vertex A?
4. Is it an undirected graph or directed graph?
5. Suppose ADJ is an NxN matrix. What will be the result if we create another matrix ADJ2 where ADJ2=ADJxADJ?

Graph Traversing

1. Implement the following graph traversing algorithm using any programming language.

**BFS(G, s) {**

**initialize vertices;**

**Q = {s}; *// Q is a queue (duh); initialize to s***

**while (Q not empty) {**

**u = RemoveTop(Q);**

**for each v ∈ u->adj {**

**if (v->color == WHITE)**

**v->color = GREY;**

**v->d = u->d + 1;**

**v->p = u;**

**Enqueue(Q, v);**

**}**

**u->color = BLACK; }}**

**Data: color[V], prev[V],d[V]**

**BFS(G) // starts from here**

**{**

**for each vertex u ∈ V-{s}**

**{**

**color[u]=WHITE;**

**prev[u]=NIL;**

**d[u]=inf;**

**}**

**color[s]=GRAY;**

**d[s]=0; prev[s]=NIL;**

**Q=empty;**

**ENQUEUE(Q,s);**

**While(Q not empty)**

**{**

**u = DEQUEUE(Q);**

**for each v ∈ adj[u]{**

**if (color[v] == WHITE){**

**color[v] = GREY;**

**d[v] = d[u] + 1;**

**prev[v] = u;**

**Enqueue(Q, v);**

**}**

**}**

**color[u] = BLACK;**

**}**

**}**

1. Find the shortest path in unweighted undirected graph using BFS

**Data: color[V], prev[V],d[V]**

**Print-Path(G, s, v)**

**{**

**if(v==s)**

**print(s)**

**else if(prev[v]==NIL)**

**print(No path);**

**else{**

**Print-Path(G,s,prev[v]);**

**print(v);**

**}**

**}**

[Click me](https://pencilprogrammer.com/algorithms/shortest-path-in-unweighted-graph-using-bfs/) to read about shortest path in unweighted undirected graph using BFS

1. Find the bipartiteness of a graph.

[Click me](https://www.geeksforgeeks.org/bipartite-graph/)

1. Find cycle in a graph.

[Click me](https://www.geeksforgeeks.org/detect-cycle-in-an-undirected-graph-using-bfs/)

1. Find the connectedness of a graph.

**References**

1. <https://www.techiedelight.com/graph-implementation-java-using-collections/>
2. <https://www.techiedelight.com/terminology-and-representations-of-graphs/>