**Introduction**

Everyone works hard to crack their dream company, prepare [Data Structures and Algorithms](https://www.codingninjas.com/studio/guided-paths/data-structures-algorithms), Computer Science Fundamentals, and solve interview problems.

**Source: giphy**

[Graph](https://www.codingninjas.com/studio/library/graph-210) is one of the important topics in [Data Structures](https://www.codingninjas.com/studio/library/data-structure) one should be well versed in.

[Questions related to Graphs](https://www.codingninjas.com/studio/problem-lists/top-graphs-interview-questions?page=2) are asked in all the major product-based companies. Also, Graphs are used everywhere, and directed graphs are used in Google's page ranking algorithms. Social networking sites like Facebook and Linkedin use graphs to represent different users as vertices and edges to represent the connections between them.  Therefore it's quite important to have a solid understanding of Graphs.

The first step to study Graphs is to know the basic terminologies related to graphs. You may refer to this article for more information. After knowing the basic terminologies, it’s time to implement your Graph. This article will discuss the **implementation of graphs in**[Python](https://www.codingninjas.com/studio/library/python-introduction). Before moving on to the implementation, it’s necessary to understand Graphs quickly.

Also see, [Merge Sort Python](https://www.codingninjas.com/studio/library/python-merge-sort-program)

**A Brief Introduction to Graphs**

A **graph** is a non-linear data structure that consists of nodes and edges. Formally, a graph can be defined as an ordered set G(V, E) where V represents the vertices and E represents the set of edges that are used to connect these vertices. (See [Graph Representation](https://www.codingninjas.com/studio/library/graph-representation))

In the above graph, the set of nodes are {1, 2, 3, 4, 5, 6} and the set of edges are {1-2, 1-5, 5-2, 2-3, 3-4, 4-5, 4-6}.

**Graphs may be directed or undirected graphs**. A directed graph is a set of vertices/nodes connected by edges, with each node having a direction associated with it. Edges are represented by arrows pointing to the direction in which the graph can be traversed. If there is an edge between vertex 0 and 1 with an arrow pointing towards 1, then the graph can be [traversed](https://www.codingninjas.com/blog/2020/09/25/graph-traversal-techniques-in-dfs-bfs/) (also see [Traversal](https://www.codingninjas.com/studio/library/traversal)) from 0 to 1 and not from 1 to 0. In contrast, an undirected graph can be traversed in either direction. The edges are bidirectional. If there is an edge between vertex 0 and 1, then the graph can be traversed from 0 to 1 and from 1 to 0.

Some graphs have weights associated with edges between two vertices. These graphs are called **Weighted Graphs**.

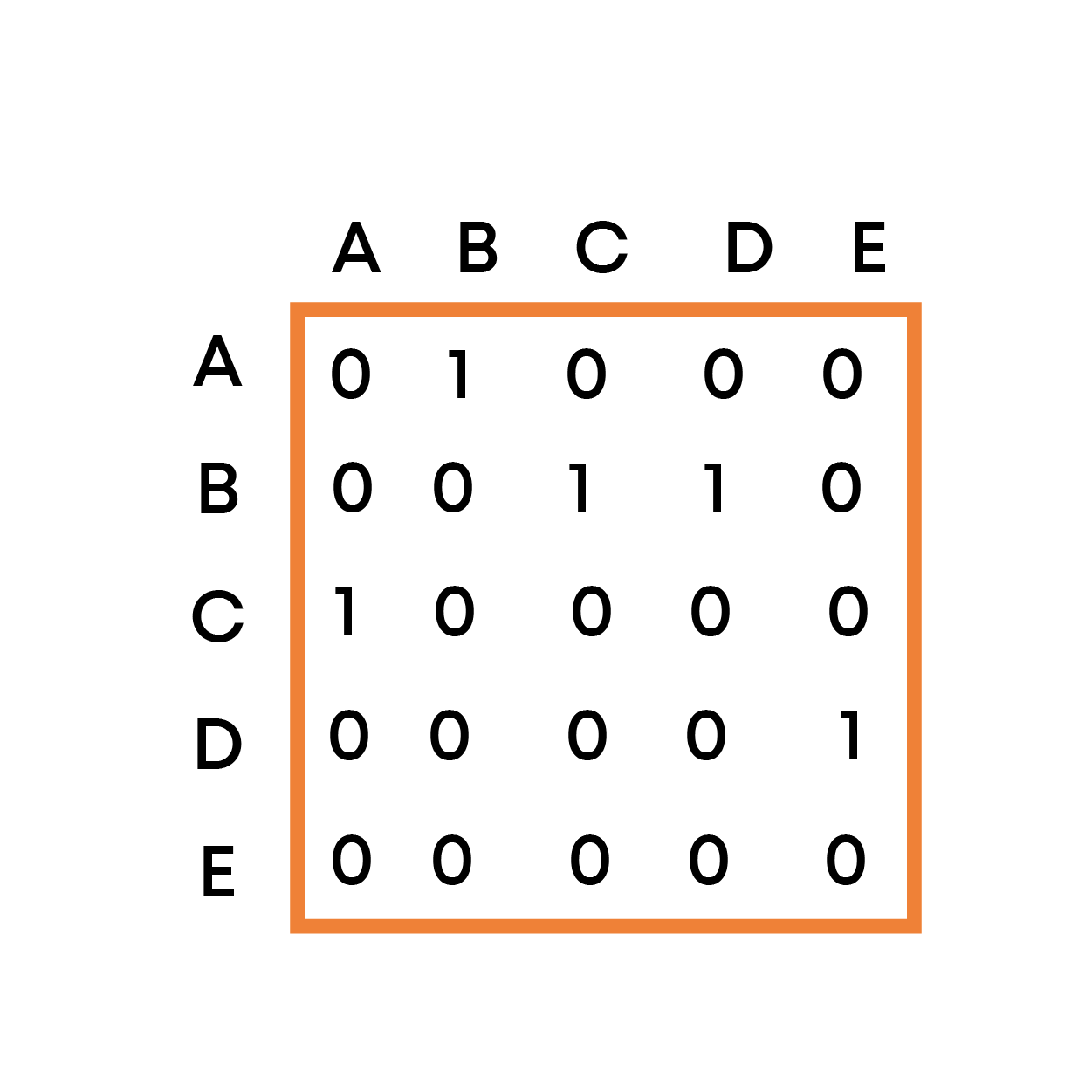
Implementation of the Graph can be done by using either an **adjacency list** or an **adjacency matrix**. Each of the two representations has its pros and cons; the choice of a particular graph representation depends on the requirements.

**Also read:**[**Application of graph data structure**](https://www.codingninjas.com/studio/library/application-of-graph-data-structure)

**Implementation of Graph in Python - Using Adjacency Matrix**

A graph can be represented using an adjacency Matrix. An adjacency matrix is a square matrix of size V \* V, where V is the number of vertices in the graph. The values in the matrix show whether a given pair of nodes are adjacent to each other in the graph structure or not. In the adjacency matrix, 1 represents an edge from node X to node Y, while 0 represents no edge from X to Y.

The adjacency matrix representation of the above-directed graph is:



Note that for an undirected graph, the adjacency matrix is always symmetric. For a weighted graph, instead of 0 and 1, the value of weight w is used to indicate that there is an edge from i to j.

**Implementation**

The **Implementation of Graphs in**[**Python**](https://www.codingninjas.com/studio/guided-paths/basics-of-python)**using Adjacency Matrix** is done in the following program:

# Adjacency Matrix representation of a graph

class Graph:

   # self represents the instance of the class.

   # By using the “self” keyword we can access the

   # attributes and methods of the class in python.

   # It binds the attributes with the given arguments.

   # This constructor is used to initialize the adjacecny matrix

   # with 0.

   def \_\_init\_\_(self, vertices):

       self.V = vertices

       self.graph = [[0 for column in range(vertices)]

                           for row in range(vertices)]

   # This function prints the adjacency matrix of the graph

   # Due to two nested loops, it is O(V^2)

   def printGraph(self):

       print("\nAdjacency Matrix:")

       for i in range(self.V):

           for j in range(self.V):

               print(self.graph[i][j], end = " ")

           print()

   # This function is used to add an edge

   # between vertices v and w.

   # This implementation is for undirected graph

   def addEdge(self, v, w):

       print("Adding an edge between", v , "and", w , "and between", w , "and", v)

       self.graph[v][w] = 1

       self.graph[w][v] = 1

   # This function is used to add a

   # vertex to the graph.

   def addVertex(self, v):

       self.V += 1

       for i in range(self.V):

           self.graph[i].append(0)

       self.graph.append([0 for column in range(self.V)])

if \_\_name\_\_ == "\_\_main\_\_":

   # Initialize the graph with 5 vertices

   g = Graph(5)

   # An edge between 0 and 1 and between 1 and 0 will be created

   g.addEdge(0, 1)

   # An edge between 0 and 2 and between 2 and 0 will be created

   g.addEdge(0, 2)

   # An edge between 1 and 2 and between 2 and 1 will be created

   g.addEdge(1, 2)

   # An edge between 2 and 0 and between 2 and 0 will be created

   g.addEdge(2, 0)

   # An edge between 2 and 3 and between 3 and 2 will be created

   g.addEdge(2, 3)

   # An edge between 3 and 4 and between 4 and 3 will be created

   g.addEdge(3, 4)

   g.printGraph()

The output of the above program is:

Adding an edge between 0 and 1 and between 1 and 0

Adding an edge between 0 and 2 and between 2 and 0

Adding an edge between 1 and 2 and between 2 and 1

Adding an edge between 2 and 0 and between 0 and 2

Adding an edge between 2 and 3 and between 3 and 2

Adding an edge between 3 and 4 and between 4 and 3

Adjacency Matrix:

0 1 1 0 0

1 0 1 0 0

1 1 0 1 0

0 0 1 0 1

0 0 0 1 0

The [time complexity](https://www.codingninjas.com/studio/library/introduction-of-time-complexity) of the addEdge() function is **O(1)** and the [space complexity](https://www.codingninjas.com/studio/library/introduction-to-space-complexity) is **O(V^2)**.

**Advantages of Adjacency Matrix Representation**

The advantages of adjacency Matrix representation are as follows:

* Insertion and deletion of an edge can be done in **O(1) time**.
* We can easily determine if two edges are adjacent to each other in constant time complexity.

**Disadvantages of Adjacency Matrix Representation**

The major disadvantage of Adjacency Matrix Representation is as follows:

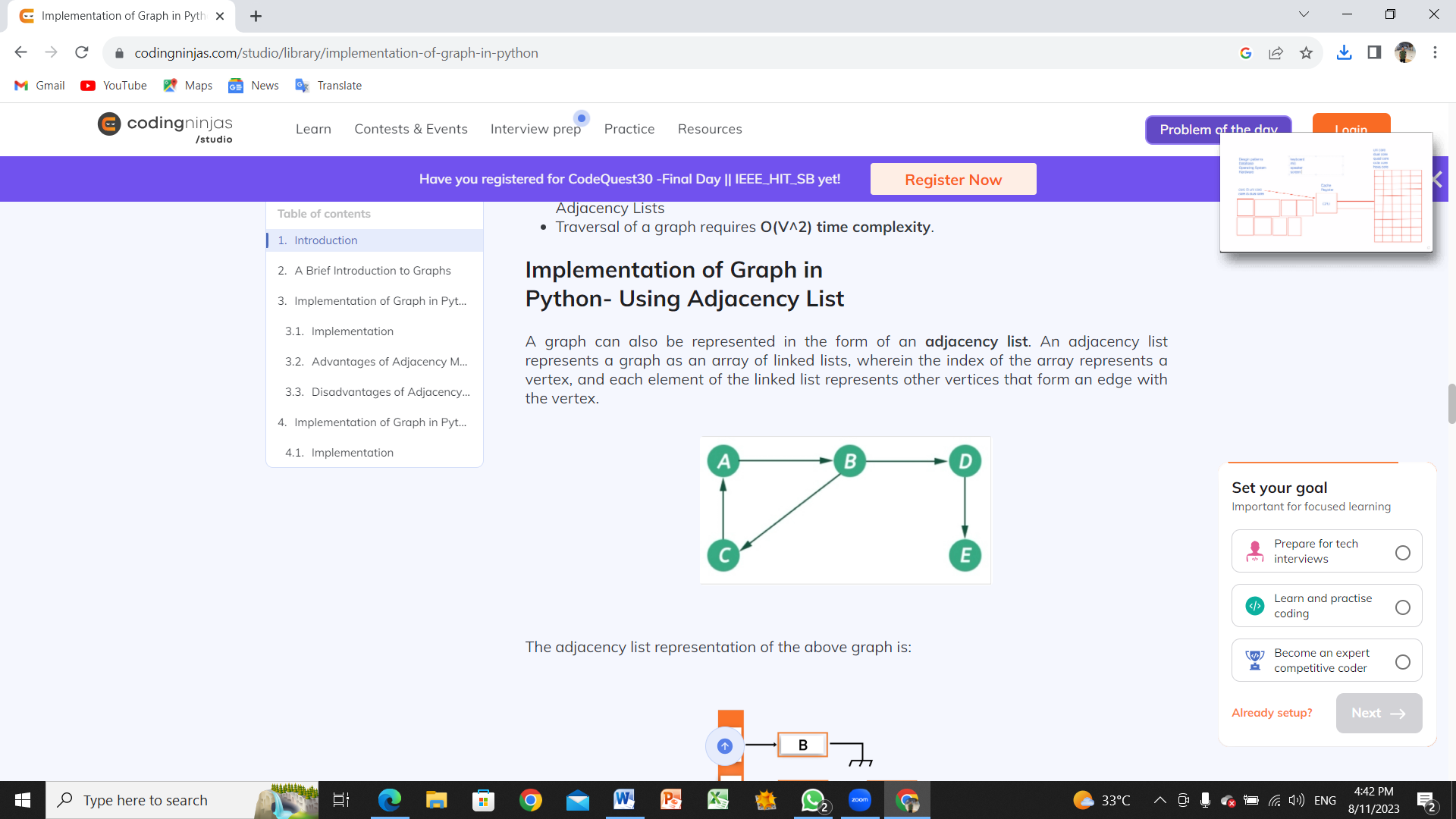
* The memory consumption of adjacency matrix representation is of the order of O(V^2), where V is the number of vertices in the graph, which is way too high as compared to Adjacency Lists
* Traversal of a graph requires **O(V^2) time complexity**.

**Implementation of Graph in Python- Using Adjacency List**

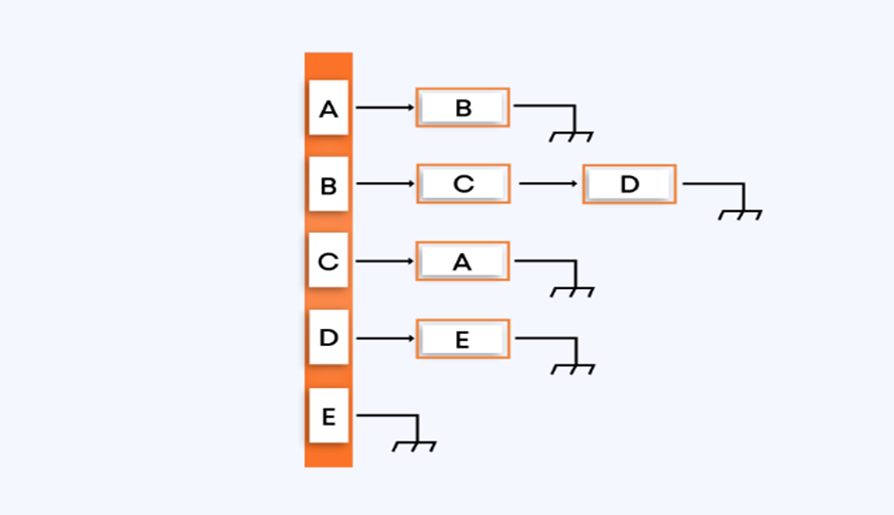
A graph can also be represented in the form of an **adjacency list**. An adjacency list represents a graph as an array of linked lists, wherein the index of the array represents a vertex, and each element of the linked list represents other vertices that form an edge with the vertex.

The adjacency list representation of the above graph is:

In the adjacency list representation shown above, the vertices A, B, C, D, and E will be stored in a list, and each vertex will have a separate list containing the vertices that form an edge with the vertex.



The adjacency list representation of the above graph is:



**Implementation**

The below program shows the**implementation of Graphs in**[**Python**](https://www.codingninjas.com/studio/library/python)**using adjacency list representation**.

# A utility function to add a vertex to the graph

'''

    The purpose of using global variables inside all the

    functions defined below is that global variables can be

    used both inside and outside the functions

'''

def addVertex(vertex):

   global graph

   # This variable is used to count the number of vertices in the graph

   global vertexCount

   # If the vertex already exists in the graph

   if vertex in graph:

       print("Vertex ", vertex, " already exists in the graph ")

   # Otherwise create a new list for this vertex and increase the count of

   # vertices in the vertexCount variable

   else:

       vertexCount += 1

       graph[vertex] = []

# A utility function to add an edge between

# vertex source and destination

def addEdge(source, destination):

   global graph

   # Check if the source vertex exists in the graph

   if source not in the graph:

       print("Source vertex ", source, " does not exist in the graph")

   # Check if the destination vertex exists in the graph

   elif destination not in graph:

       print("Destination vertex ", destination,

             " do not exist in the graph")

   # If both the source and destination vertex exists in the

   # graph, an edge can be added.

   else:

       # Create a new list by passing in the data of the destination vertex

       temp = [destination]

       # The append() method adds a single item to the list. A new list is

       # not created. Instead, the original list is modified by adding the new

       # item to the rear of the original list

       graph[source].append(temp)

# A utility function to print the adjacency list representation of

# a graph

def printGraph():

   global graph

   # Pick each vertex

   for vertex in graph:

   # Pick all the vertices that form an edge with the picked vertex above.

       for edges in graph[vertex]:

           print(vertex, "->", edges[0])

if \_\_name\_\_ == '\_\_main\_\_':

   # Initializing the graph with an empty list

   # This will store all the vertices

   graph = {}

   # This variable is used to store the count of vertices in the graph

   vertexCount = 0

   # Creating 5 vertices in the graph. Initially none

   # of them is connected to each other or have an edge

   # between them

   addVertex(1)

   addVertex(2)

   addVertex(3)

   addVertex(4)

   addVertex(5)

   # Adding the edges between source and destination vertices

   addEdge(1, 2)

   '''

   1 --------> 2

   '''

   addEdge(2, 3)

   '''

   1 ---------> 2

                |

                |

               \|/

                3

   '''

   addEdge(3, 4)

   '''

   1 ----------> 2

                 |

                 |

                \|/

   4 <---------- 3

   '''

   addEdge(4, 1)

   '''

   1 ----------> 2

  /|\  |

   | |

   |  \|/

   4 <---------- 3

   '''

   # Function call to printGraph() function

   printGraph()

   print("Internal representation of graph is: ", graph

The output of the above program is:

1 -> 2

2 -> 3

3 -> 4

4 -> 1

Internal representation of graph is:  {1: [[2]], 2: [[3]], 3: [[4]], 4: [[1]], 5: []}

The **time complexity** of the addVertex() is O(1), addEdge() is O(1) as append() operation takes constant time. The printGraph() function however takes **O(V\*E) time**.

The **space complexity** of the above program is **O(V + E)**, where V is the number of vertices and E is the number of edges. The space complexity of the above program will be**O(V^2)** in the case of a complete graph.