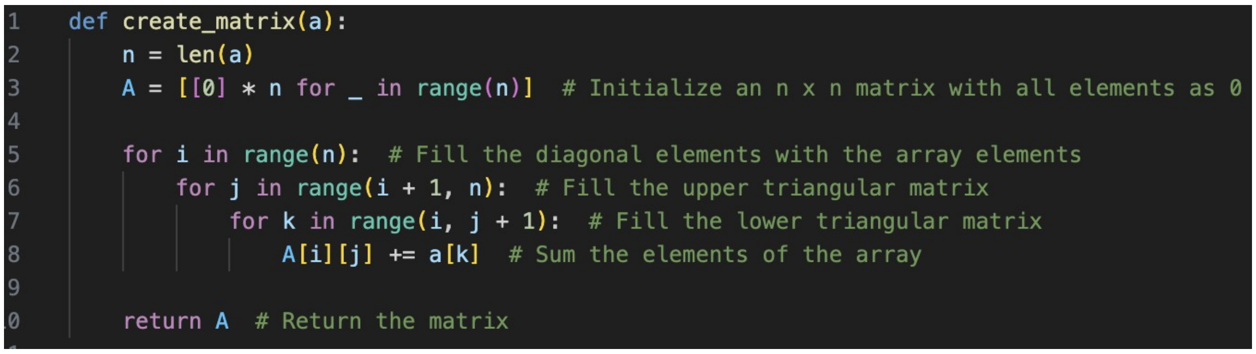
HW1 Yaroslav Popryho
ypopry2@uic.edu

Problem 1: (24 points)
For each pair of A and B in the list below, indicate their asymptotic relation (O, Ω, Θ). No justification is needed. (Assume that k ≥ 1, c > 1 are constants.)

1. A = O(B)
2. A = O(B)
3. -
4. A = O(B); A = Ω(B); A = Θ(B)

Problem 2: (25 points) You are given a list of integers a1, a2, . . . , an. You need to output an n × n matrix A in which the entries A[i, j] = ai + ai+1 + · · · + aj for i < j (for i ≥ j, A[i, j] is 0). Consider the following algorithm for this problem.

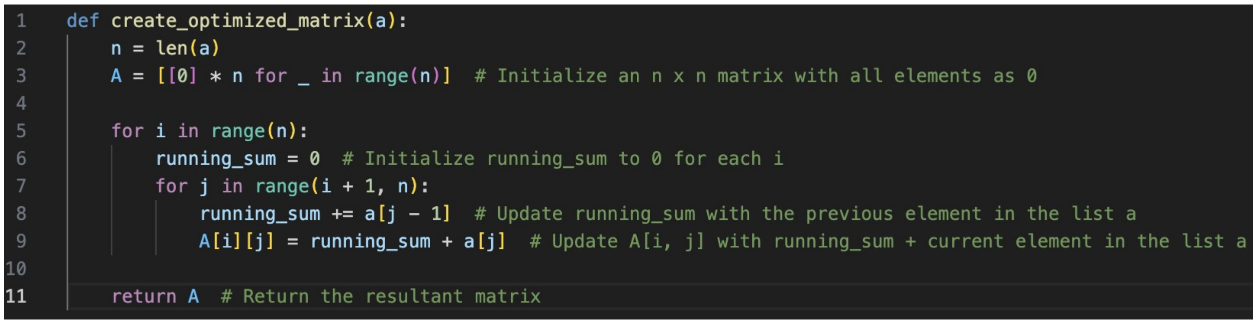


```
def create_matrix(a):
    n = len(a)
    A = [[0] * n for _ in range(n)]  # Initialize an n x n matrix with all elements as 0

    for i in range(n):  # Fill the diagonal elements with the array elements
        for j in range(i + 1, n):  # Fill the upper triangular matrix
            for k in range(i, j + 1):  # Fill the lower triangular matrix
                A[i][j] += a[k]  # Sum the elements of the array

    return A  # Return the matrix
```

1. What is the worst-case running time of above algorithm? Give the running time as a function of n.

$\boxed{O(n^3)}$

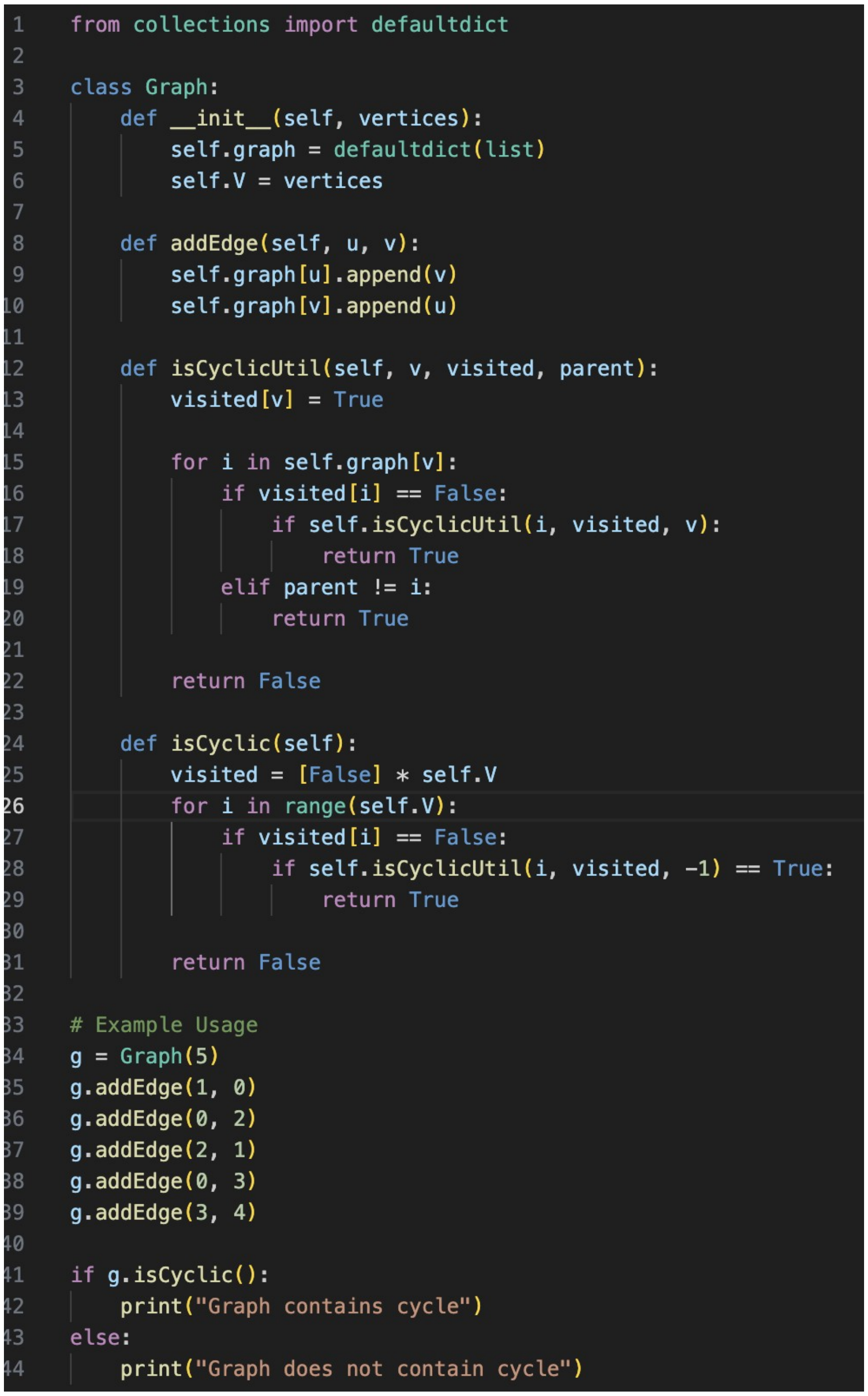
2. Design an algorithm for this problem with asymptotically faster running time than above algorithm. What is the running time of your new algorithm?



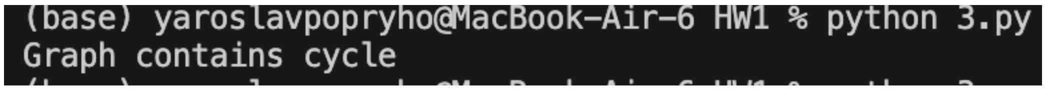
```
def create_optimized_matrix(a):
    n = len(a)
    A = [[0] * n for _ in range(n)]  # Initialize an n x n matrix with all elements as 0

    for i in range(n):
        running_sum = 0  # Initialize running_sum to 0 for each i
        for j in range(i + 1, n):
            running_sum += a[j - 1]  # Update running_sum with the previous element in the list a
            A[i][j] = running_sum + a[j]  # Update A[i, j] with running_sum + current element in the list a

    return A  # Return the resultant matrix
```

time complexity of the optimized algorithm is $\boxed{\text{O(n2)}}$, which is asymptotically faster than the original algorithm with a time complexity of $\boxed{\text{O(n3)}}$.

Problem 3: (25 points) Give an algorithm to detect whether a given undirected connected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one cycle (do not output all cycles in the graph, just any one of them). Justify the running time bound of your algorithm. You will receive 25 points if the running time of your algorithm is O(m + n) for a graph with n nodes and m edges. You will receive 18 points if your solution is correct, and the running time of your algorithm is polynomial in n and m.



```python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def isCyclicUtil(self, v, visited, parent):
        visited[v] = True

        for i in self.graph[v]:
            if visited[i] == False:
                if self.isCyclicUtil(i, visited, v):
                    return True
            elif parent != i:
                return True

        return False

    def isCyclic(self):
        visited = [False] * self.V
        for i in range(self.V):
            if visited[i] == False:
                if self.isCyclicUtil(i, visited, -1) == True:
                    return True

        return False

# Example Usage
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 1)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print("Graph contains cycle")
else:
    print("Graph does not contain cycle")
```

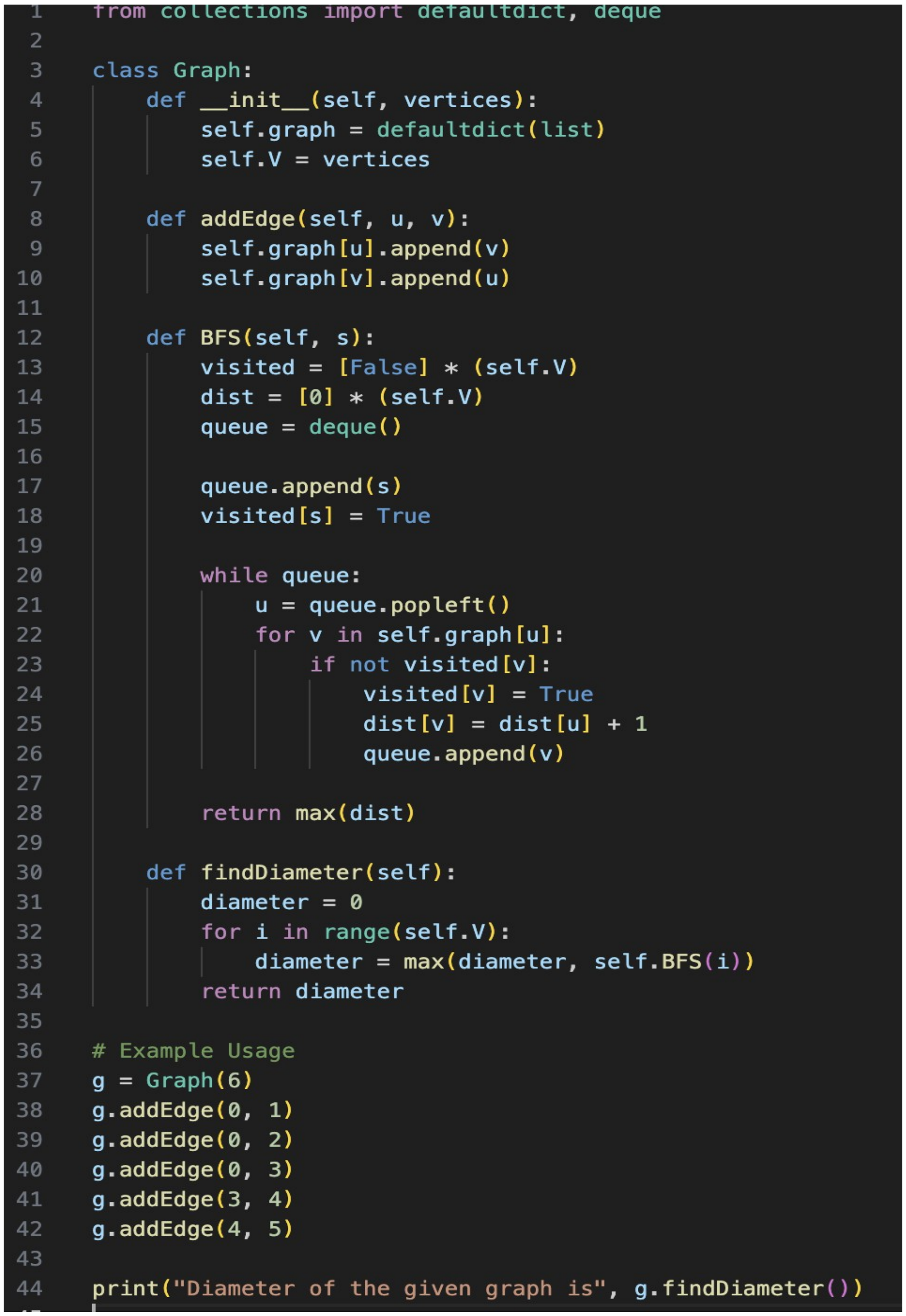


```
(base) yaroslavpopryho@MacBook-Air-6 HW1 % python 3.py
Graph contains cycle
```

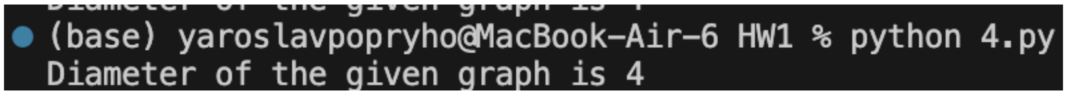
The algorithm uses DFS. Each node is visited exactly once, and each edge is also visited exactly once. Therefore, the running time of this algorithm is $\boxed{O(n+m)}$.

Problem 4: (26 points) We say that the distance between two nodes u, v in a connected graph G = (V, E) is the minimum number of edges in a path joining them; we'll denote this by dist(u, v). We say that the diameter of G is the maximum distance between any pair of nodes. For example, the diameter of the following graph is 3 (d(1, 6) = d(2, 6) = d(3, 6) = 3).

Design an algorithm that runs in time O(n(n + m)) and output the diameter of G.



```python
from collections import defaultdict, deque

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def BFS(self, s):
        visited = [False] * (self.V)
        dist = [0] * (self.V)
        queue = deque()

        queue.append(s)
        visited[s] = True

        while queue:
            u = queue.popleft()
            for v in self.graph[u]:
                if not visited[v]:
                    visited[v] = True
                    dist[v] = dist[u] + 1
                    queue.append(v)

        return max(dist)

    def findDiameter(self):
        diameter = 0
        for i in range(self.V):
            diameter = max(diameter, self.BFS(i))
        return diameter

# Example Usage
g = Graph(6)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(0, 3)
g.addEdge(3, 4)
g.addEdge(4, 5)

print("Diameter of the given graph is", g.findDiameter())
```

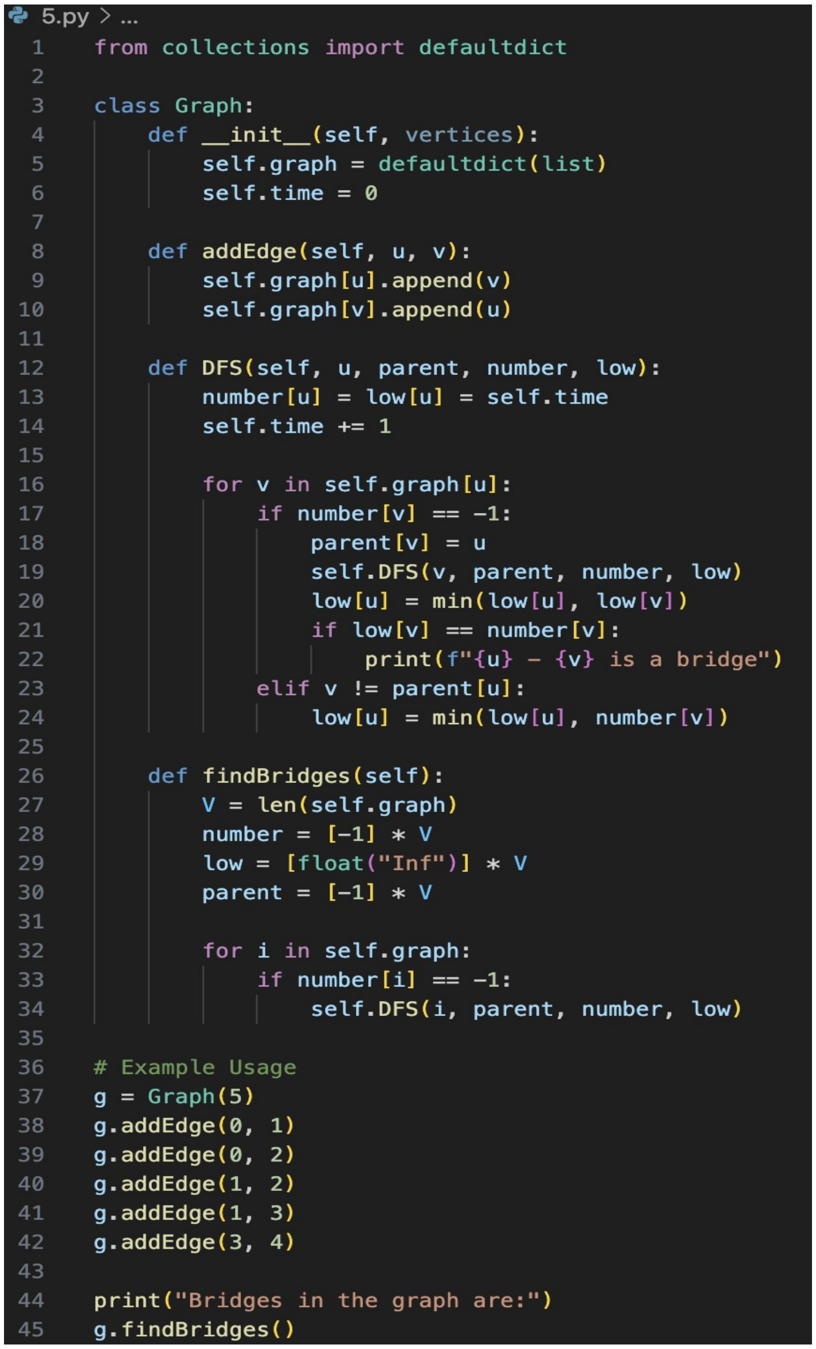


```
(base) yaroslavpopryho@MacBook-Air-6 HW1 % python 4.py
Diameter of the given graph is 4
```

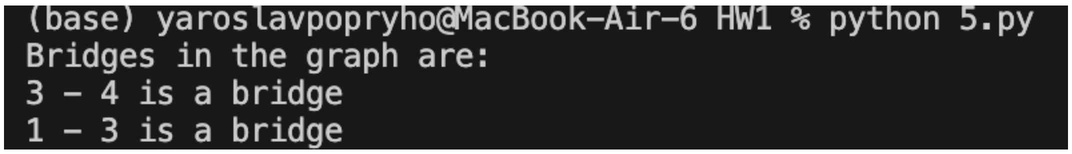
For each node in the graph, we perform a BFS which takes O(n+m) time, where n is the number of nodes and m is the number of edges.

Since we perform BFS for each of the n nodes, the total time complexity of the algorithm is O(n(n+m)).

Problem 5$*$: (25 points for graduate students, 5 extra points for undergraduate students) Given a connected graph G with n vertices. We say an edge of G is a bridge if the graph becomes a disconnected graph after removing the edge. Give an algorithm that finds all the bridges. Justify the running time bound of your algorithm.



```python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.time = 0

    def addEdge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def DFS(self, u, parent, number, low):
        number[u] = low[u] = self.time
        self.time += 1

        for v in self.graph[u]:
            if number[v] == -1:
                parent[v] = u
                self.DFS(v, parent, number, low)
                low[u] = min(low[u], low[v])
                if low[v] == number[v]:
                    print(f"{u} - {v} is a bridge")
            elif v != parent[u]:
                low[u] = min(low[u], number[v])

    def findBridges(self):
        V = len(self.graph)
        number = [-1] * V
        low = [float("Inf")] * V
        parent = [-1] * V

        for i in self.graph:
            if number[i] == -1:
                self.DFS(i, parent, number, low)

# Example Usage
g = Graph(5)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(1, 3)
g.addEdge(3, 4)

print("Bridges in the graph are:")
g.findBridges()
```



```
(base) yaroslavpopryho@MacBook-Air-6 HW1 % python 5.py
Bridges in the graph are:
3 - 4 is a bridge
1 - 3 is a bridge
```

The algorithm traverses each vertex once and explores each edge once during the DFS, leading to a time complexity of $\boxed{O(n+m)}$

- **Additional operations performed during the traversal for each vertex and edge, such as updating values and condition checks, are done in constant time, $\boxed{O(1)}$, and do not affect the overall time complexity.**
- **Thus, the overall time complexity of finding all bridges in the graph is $\boxed{O(n+m)}$.**