***The functions from the class Undirected Graph***

class UndirectedGraph:  
 def \_\_init\_\_(self, number\_of\_vertices, number\_of\_edges):  
 self.\_\_number\_of\_vertices = number\_of\_vertices  
 self.\_\_number\_of\_edges = number\_of\_edges  
 self.\_\_neighbors = {}  
 self.\_\_costs = {}  
 for index in range(number\_of\_vertices):  
 self.\_\_neighbors[index] = []  
  
 @property  
 def number\_of\_vertices(self):  
 return self.\_\_number\_of\_vertices  
  
 @property  
 def number\_of\_edges(self):  
 return self.\_\_number\_of\_edges  
  
 @property  
 def neighbors(self):  
 return self.\_\_neighbors  
  
 @property  
 def costs(self):  
 return self.\_\_costs  
  
 def set\_number\_of\_vertices(self, number\_of\_vertices):  
 self.\_\_number\_of\_vertices = number\_of\_vertices  
  
 def set\_number\_of\_edges(self, number\_of\_edges):  
 self.\_\_number\_of\_edges = number\_of\_edges  
  
 def set\_incoming\_edges(self, neighbors):  
 self.\_\_neighbors = neighbors  
  
 def set\_costs(self, costs):  
 self.\_\_costs = costs  
  
 def parse\_vertices(self):  
 *"""  
 Time Complexity - Theta(nr\_of\_vertices)  
 :return: an iterator in the list of vertices  
 """* vertices = list(self.\_\_neighbors.keys())  
 for v in vertices:  
 yield v

def parse\_neighbors(self, vertex):  
 *"""  
 Time Complexity - Theta(nr\_of\_incoming\_edges)  
 :param vertex: the vertex for which we parse the neighbors  
 :return: an iterator in the dict of neighbors  
 """* for y in self.\_\_neighbors[vertex]:  
 yield y  
  
 def parse\_cost(self):  
 *"""  
 Time Complexity - Theta(nr\_of\_edges)  
 :return: an iterator in the list containing the costs of the edges  
 """* keys = list(self.\_\_costs.keys())  
 for key in keys:  
 yield key  
  
 def add\_vertex(self, vertex):  
 *"""  
 Time Complexity - Theta(1)  
 :param vertex: the vertex that we want to add  
 :return: True if a new vertex was added, False otherwise  
 """* if vertex in self.\_\_neighbors.keys():  
 return False  
 self.\_\_neighbors[vertex] = []  
 self.\_\_number\_of\_vertices += 1  
 return True  
  
 def remove\_vertex(self, vertex):  
 *"""  
 Time Complexity - Theta(nr\_of\_incoming\_edges + nr\_of\_outgoing\_edges)  
 :param vertex: the vertex that we want to delete  
 :return: True if the vertex was deleted, False otherwise  
 """* if vertex not in self.\_\_neighbors.keys():  
 return False  
  
 for v in list(self.\_\_neighbors.keys()):  
 if vertex in self.\_\_neighbors[v]:  
 self.\_\_neighbors[v].remove(vertex)  
  
 self.\_\_neighbors.pop(vertex, None)  
  
 for key in list(self.\_\_costs.keys()):  
 if key[0] == vertex or key[1] == vertex:  
 self.\_\_costs.pop(key)  
 self.\_\_number\_of\_edges -= 1  
  
 self.\_\_number\_of\_vertices -= 1  
 return True  
  
 def add\_edge(self, x, y, cost):  
 *"""  
 Time Complexity - Theta(1)  
 - the time complexity of accessing an element in a dict is Theta(1)  
 - the time complexity of appending/adding a new element in a list in Theta(1)  
 :param x: starting vertex of an edge  
 :param y: final vertex of an edge  
 :param cost: the cost of the edge  
 :return: True if the edge was added, False otherwise  
 """* if x == y or self.find\_if\_edge(x, y):  
 return False  
 self.\_\_neighbors[x].append(y)  
 self.\_\_neighbors[y].append(x)  
 self.\_\_costs[(x, y)] = cost  
 self.\_\_costs[(y, x)] = cost  
 self.\_\_number\_of\_edges += 1  
 return True  
  
 def remove\_edge(self, x, y):  
 *"""  
 Time Complexity - Theta(1)  
 :param x: starting vertex of an edge  
 :param y: final vertex of an edge  
 :return: True if the edge was deleted, False otherwise  
 """* if x == y or not self.find\_if\_edge(x, y):  
 return False  
 self.\_\_neighbors[x].remove(y)  
 self.\_\_neighbors[y].remove(x)  
 self.\_\_costs.pop((x, y))  
 self.\_\_costs.pop((y, x))  
 self.\_\_number\_of\_edges -= 1  
 return True  
  
 def degree(self, vertex):  
 *"""  
 Time Complexity - Theta(1)  
 :param vertex: the vertex for which we compute the number of neighbors  
 :return: the number of neighbors for that vertex, if it is an existent vertex in the graph, -1 otherwise  
 """* if vertex not in self.\_\_neighbors.keys():  
 return -1  
 return len(self.\_\_neighbors[vertex])  
  
 def find\_if\_edge(self, x, y):  
 *"""  
 Time Complexity - Theta(1)  
 :param x: starting vertex of an edge  
 :param y: final vertex of an edge  
 :return: True if the edge is in the graph, False otherwise  
 """* return y in self.\_\_neighbors[x]  
  
 def change\_cost(self, x, y, new\_cost):  
 *"""  
 Time Complexity - Theta(1)  
 :param x: starting vertex of an edge  
 :param y: final vertex of an edge  
 :param new\_cost: the new cost of the given edge  
 :return: True if the cost of the edge was changed, False otherwise  
 """* if (x, y) not in self.\_\_costs.keys():  
 return False  
 self.\_\_costs[(x, y)] = new\_cost  
 return True  
  
 def copy\_graph(self):  
 return deepcopy(self)  
  
 def dfs\_preorder(self, node, mst\_edges, visited\_dfs, result):  
 *"""  
 Time Complexity: O(n)  
 A function that performs depth-first search (DFS) in a given Minimum Spanning Tree (MST), from a given  
 starting node and stores the preorder traversal of the tree.  
 :param node: the current node from which the DFS starts  
 :param mst\_edges: a dictionary representing the edges of Minimum Spanning Tree (MST)  
 :param visited\_dfs: a list of visited nodes in the MST  
 :param result: a list that collects the node in the DFS traversal order  
 :return: None  
 """* visited\_dfs.add(node)  
 result.append(node)  
 for neighbor in mst\_edges[node]:  
 if neighbor not in visited\_dfs:  
 self.dfs\_preorder(neighbor, mst\_edges, visited\_dfs, result)  
  
 def hamiltonian\_cycle\_of\_no\_more\_than\_twice\_the\_minimum\_cost(self):  
 *"""  
 Time complexity: O(m + n \* log n)  
 A function that returns a list representing a Hamiltonian cycle of no more than twice the minimum cost,  
 using a 2-approximation algorithm (MST (with Prim's Algorithm) + DFS + shortcutting).  
 :return:a list of vertices representing a Hamiltonian cycle of no more than twice the minimum cost  
 """* # Building MST using Prim's Algorithm. Time Complexity: O(m + n \* log n)  
 start = 0  
 visited = set()  
 mst\_edges = {v: [] for v in self.parse\_vertices()}  
 min\_heap = [(0, start, -1)] # (cost, current\_vertex, parent)  
  
 total\_vertices = self.\_\_number\_of\_vertices  
 added = 0  
  
 while min\_heap and added < total\_vertices:  
 cost, u, parent = heapq.heappop(min\_heap)  
 if u in visited:  
 continue  
 visited.add(u)  
 added += 1  
 if parent != -1:  
 mst\_edges[parent].append(u)  
 mst\_edges[u].append(parent)  
  
 for v in self.parse\_neighbors(u):  
 if v not in visited:  
 heapq.heappush(min\_heap, (self.\_\_costs[(u, v)], v, u))  
  
 # Performing DFS to get preorder traversal. Time Complexity: O(n)  
 preorder = []  
 self.dfs\_preorder(start, mst\_edges, set(), preorder)  
  
 # Shortcutting the tour. Time Complexity: O(n)  
 seen = set()  
 tour = []  
 for node in preorder:  
 if node not in seen:  
 seen.add(node)  
 tour.append(node)  
 tour.append(start) # close the cycle  
  
 return tour

***The functions from the UI***

def hamiltonian\_cycle\_of\_no\_more\_than\_twice\_the\_minimum\_cost\_ui(self):  
 *"""  
 A function that finds a Hamiltonian cycle of no more than twice the minimum cost and prints it in an undirected graph.  
 :return: None, but it prints the Hamiltonian cycle  
 """* self.read\_undirected\_graph\_from\_file\_ui()  
 undirected\_graph = self.\_\_graphs[self.\_\_current]  
 hamiltonian\_cycle = undirected\_graph.hamiltonian\_cycle\_of\_no\_more\_than\_twice\_the\_minimum\_cost()  
 print(f"A Hamiltonian cycle of cost no more than twice the minimum cost is:\n{hamiltonian\_cycle}")