Task 1:
In this task we created 3 files; Node.py, Edge.py and Graph.py, each consisting of a class. In each class we created functions to get, set, delete.etc. The graph class manages nodes and edges according to the factory design pattern.

Test(in “Tester.py”): To test we created a small graph called “venner”. Run codeline 22-29.

Task 2:
We created the file Printer.py, consisting of a class. The class exports a graph into a text file with a given format. We handled this by separating the nodes and arcs, before printing them into a file named "graph.tsv".

Test(in “Tester.py”): To test we created a small graph called “Grid32”. Run codeline 47-48.

Task 3:
We created the file Parser.py, consisting of a class. The class imports a graph at the given format from a text file. Here we inserted all the content into the list “cleanlines”, before inserting all nodes and edges into the graph.

Test(in “Tester.py”): To test the task we used the file created in task 2 ("graph.tsv"). Run codeline 52-55.

Task 4:
In this task we created a method called ‘degreeOfNodes’ inside the Calculator.py-class that finds the degree of all nodes in a graph and appends this into a dict that holds the node-object as key, and the degree of the node as value.

Test(in “Tester.py”): Run the codeline 60 to get the degree of all the nodes in the graph ‘grid32’.

Task 5:
Here we created a method called ‘plotDegreeOfNodes’ that runs the method ‘degreeOfNodes’ to get all the data, then plots this into a bar-chart figure.

Test(in “Tester.py”): Run codeline 64 to plot the distribution of the nodes to the graph ‘grid32’.

Task 6:
Here we created the method ‘extractComponentsOfNode’. It finds all the neighbours of a given node, and then iterates through all these neighbours and finds all the neighbours of these nodes again. It returns the list containing all the components the node is connected to.

Test(in “Tester.py”): Run codeline 68-70 to see all the connected components of the node ‘n11’ in the graph ‘grid32’.

Task 7:
In calculator.py we created the method ‘extractComponentsOfGraph’, which obtains all nodes from the graph. Uses the ‘extractComponentsOfNode’ on the first obtained node and stores the whole component of nodes into a list. Then it removes the “used” nodes from the list and repeats itself until all nodes in the graph are checked. At the end a list with connected components is returned. The returned list looks like this; [[component1], [component2],...].

Test(in “Tester.py”): To test the task we created a small graph called “Grid70”. Rund codeline 87.

Task 8:
Here we created the method ‘plotSizesOfConnectedComp’ that runs the method ‘extractComponentsOfGraph’ to get the data, then plots these subgraphs and their size into a bar-chart figure.

Test(in “Tester.py”): Run codeline 91 to see the subgraphs and their sizes of ‘graph70’.

Task 9:
Here we created the method ‘distanceToAllNodes’ in Calculator.py. This method uses a auxiliary-function ‘buildGraph’ in Graph.py, that returns the graph in a dictionary, having the nodes name as key (string), and the neighbour nodes as values (string). Then ‘distanceToAllNodes’ iterates through all the nodes in the given graph, and uses the auxiliary-function ‘shortestPathBFS’ that does a breadth-first-search and finds the shortest path between the ‘startNode’ and all the other nodes in the given graph. At last it returns the dict containing all the nodes that the ‘startNode’ could reach as key, and the distance as value.

Test(in “Tester.py”): Run codeline 95 in to see all the nodes the startnode ‘n11’ could reach in ‘graph32’ and the distances between them.

Task 10:
In calculator.py we created the method ‘diameterOfGraph’,which uses the auxiliary functions ‘buildGraph’ and ‘distanceToAllNodes’. For each node the method finds the distance from the given node to all other nodes, picks the largest one and stores this pair of nodes and the distance between them into the dictionary ‘combDict’. When all nodes are checked, the method returns the pair of nodes in ‘combDict’ with the largest distance between them. The returned diameter is correct, but the returned nodepair may not be the only option.

Test(in “Tester.py”): To test the task with ‘grid32’ as input graph run codeline 99.

Task 11:
In Generator.ny we created a method called ‘barbasiAlbert’, which generates graphs according to the Barabási–Albert algorithm. First we generate a graph and insert two connected nodes. For each number from 3 and up to the input ‘numberOfNodes’ we use the auxiliary function ‘getProbability’ to calculate the connection-probability to all nodes in the graph, before storing the node and its connection-probability into the dictionary ‘nodeProb’. Now we generate a new node. Then we use an auxiliary function called ‘getR’ to get a random number between 0 and 1. For each node in ‘nodeProb’ we call the ‘getR’ function, and if the generated number is below the connection-probability of the node, the new node is inserted into the graph and the two nodes are connected. This way the new node can connect to everything from 0 to all of the nodes in the graph, dependent on the probability-outcome. If the new node wont connect to any of the nodes in the graph, it does not take part of the graph. The method repeats itself for every new node until ‘numberOfNodes’ is reached.

Test(in “Tester.py”): Task 11 is tested in task 12.

Task 12:
From Task 11 we have a random-generated graph with 50 nodes called ‘graph’.
Test(in “Tester.py”):

- To print this graph into a file, run codeline 110-111
- To parse this file that you now have printed, run codeline 113-116. Notice that you must run codeline 110-111 before trying to parse the file, else the file would not exist.
- To test if the generated graph is made of a single connected component, run codeline 120.
- To extract and plot the distribution of the degree of the nodes from the generated graph, run codeline 124-126.
- To calculate the diameter of the generated network, run codeline 130.