## Motivation of choices

In this exercise, initially, we assume that all the vertex have the system. In this way, we don't have to create the constraint that for every pair of friends at least one of them has the software, but we just have to check that the constraint is maintained when we remove the software on a computer.

We think that with this reasoning we can achieve a good result in a simple, fast and error-free way.

The vertices are crossed randomly. This choice was made to minimize the computational complexity and the working time of the algorithm, but also because other types of visit of the graphs in some cases gave better results, in others worse (on average the same result), but with worse performance.

To find our locally optimal, the current vertex and all its friends were considered. Among these vertices we subsequently chose to remove the software on the vertex with the minimum full_coverage_degree (note that the explanation of the meaning of this parameter is given in the README.txt file present in the folder of exercise 5).

## Feature of the greedy programming

**Feasible**: Our choice is always feasible and always respects the constraints of the problem because only the vertices that have the system and have the full_coverage parameter set to True are taken into consideration. This parameter set to True indicates that all friends of that vertex have the system, therefore removing the system from the vertex will certainly respect the constraints.

**Locally optimal**: Our locally optimal is taken according to the full_coverage_degree parameter. By making the decision based on this parameter instead of the degree we are able to minimize the number of vertices that will not be able to remove the system. In fact, the algorithm has been tested on the 100 graphs generated randomly using both the degree and the full_coverage_degree. With the use of this last parameter on average there were 0.7 systems per graph less.

**Non revocable**: Our choice is not revocable because when you remove the system from a vertex, all his friends will have the full_coverage parameter set to false, consequently his friends will never be considered for the removal of the system and there will be no way to revisit that vertex. At the same time, if the system is removed on a friend of the current vertex, the current vertex will have the full_coverage parameter set to false and therefore it will not be possible to remove the system on that vertex.

**Global optimal**: Our algorithm cannot always obtain an optimal solution. This is demonstrated by the fact that by using BFS to visit the graph in our algorithm in some cases it manages to give a better (and in others worse) result. This highlights that the result also depends on which vertices are selected before or after. But suppose that our solution does not stray far from the optimal one because in small graphs, such as the one used in the test.py file in the folder of exercise 5, it manages to find the optimal solution

## Computational Complexity

**graphs_generator(directed):**

Because both the number of vertices and the number of graphs to be generated are 100, we consider n = 100 for the calculation of the complexity.
In the first for-loop we only create the n graphs. Therefore the time complexity of this loop is O(n).
In the second for-loop we insert n vertex in n graphs. The time complexity of the insert_vertex(v) is O(1), therefore this loop has complixity $O(n^2)$.
The third for-loop is made by three for-loop innested, one for the n graphs and two for the n vertices. Because the time complexity of the get_edge(i, j) and of the insert_edge(i, j, None) is O(1), the total complexity of this for-loop is $O(n^3)$.
The total complexity of this algorithm is:
$O(n + n^2 + n^3) = O(n^3)$

**Systems_decision_maker(graph):**

In the first for-loop we use the method set_system(boolean), set_full_coverage(boolean), set_full_coverage_degree(int) and graph.degree(v). All this methods have time complexity

O(1), therefore the time complixity of this loop is O(n).
In the second for-loop we cross all the vertices (time O(n)) and, after, we use the system() and full_coverage() method (time O(1)) and the incident_edges(v) (time O(deg(v)).
Subsequently we have a for-loop of the incident edges, therefore this for-loop have time complexity O(deg(v)).
After this loop, we have an another loop on the incident edges in which there is a further loop always on the incident edges, therefore this for-loop have time complexity $O(deg^2(v))$.
Therefore the all second for-loop have time complexity:

$O(n*(deg(v) + deg^2(v)) = O(n*deg^2(v)) = O((n + 2m)*deg(v)) = O(n + 4m) \risingdotseq O(n + m)$

At the end the all function have a complexity of
O(n) + O(n + m) = O(n + m)

**FakeNewsDetector**:

In this file is present the main.
At the start, we use the graph_generator for create the 100 graphs (time complexity $O(n^3)$).
After, for each graph, we call the system_decision_maker and we check how many systems and errors there are in each graph. This operation has time complexity:

O(n*[(n+m)+n]) = O(n*(n + m))

The whole main has time complexity:

$O(n^3) + O(n*(n + m)) = O(n^3)$