Carp Nicoleta
Gr. 911

```
def topological_sort(g: GraphDirected):
    """
    1. We look through all the in bound vertices of each vertex and keep a count of them in a vector,
       if the counter for any vertex is 0 it is added to the queue

    2. While the queue still has vertices we take the first one and append it to the topologically sorted graph
    3. Then we look through its out_bound vertices and for each decrement the counter
       if any of them reach 0 they are added to the queue

    :param g: a directed graph
    :return: None - if nr of vertices in top. sort. is smaller than  nr of vertices in graph
             else - the graph vertices sorted topologically
    """
    sort = []
    queue = []
    count = {}

    for x in g.in_bound.keys():
        count[x] = len(g.in_bound[x])
        if count[x] == 0:
            queue.append(x)

    while len(queue) != 0:
        x = queue.pop(0)
        sort.append(x)
        for y in g.out_bound[x]:
            count[y] -= 1
            if count[y] == 0:
                queue.append(y)
    if len(sort) < len(g.out_bound.keys()):
        return None
    return sort

def highest_cost_path(x, y, sort, g : GraphDirected):
    """
    1. Initializes the distances dictionary
    2. Go through the topol. sorted vertices till we reach x and start calculating the distances from there
    3. Look at every out bound vertex of each vertex in the sort vector starting with x and determining the
       longest distance for it so far.
       In case we discover a longer path we change it as well as the predecessor of that specific vertex
    4. When we reach the end vertex we reconstruct the path and return the necessary values

    :param x: starting vertex
    :param y: end vertex
    :param sort: the topologically sorted vertices
    :param g: the graph
    :return: the highest cost path between two vertices and the distance
    """
    dist = {}
    pred = {}
    for v in g.in_bound.keys():
        dist[v] = 0
```

Carp Nicoleta
Gr. 911

```
i = 0
while sort[i] != x:
    if sort[i] == y:
        return sort, 0
    i += 1
while i < len(sort):
    if sort[i] == y:
        break
    for v in g.out_bound[sort[i]]:
        if dist[v] < (dist[sort[i]] + g.edges[sort[i], v]):
            dist[v] = dist[sort[i]] + g.edges[sort[i], v]
            pred[v] = sort[i]
    i += 1

path = [y]
while y != x:
    y = pred[y]
    path.append(y)
path.reverse()
return path, dist[path[len(path) - 1]]
```