1) Bijgeleverd bevindt zich de file met code met commentaar.

a.

De complexiteit van dit algoritme is $O(n^2)$. Het gaat hier om een dubbel geneste for-loop. Beide loops lopen van nul tot n, waarbij n het aantal nodes is.

b.

De complexiteit van het recursieve algoritme is $O(n!)$. Bij de eerste aanroep kunnen n nieuwe aanroepen gestart worden. Elke aanroep kan vervolgens weer $n-1$ aanroepen starten, enzovoort. In het worst-case geval resulteert dit dus in een algoritme van orde $O(n!)$.

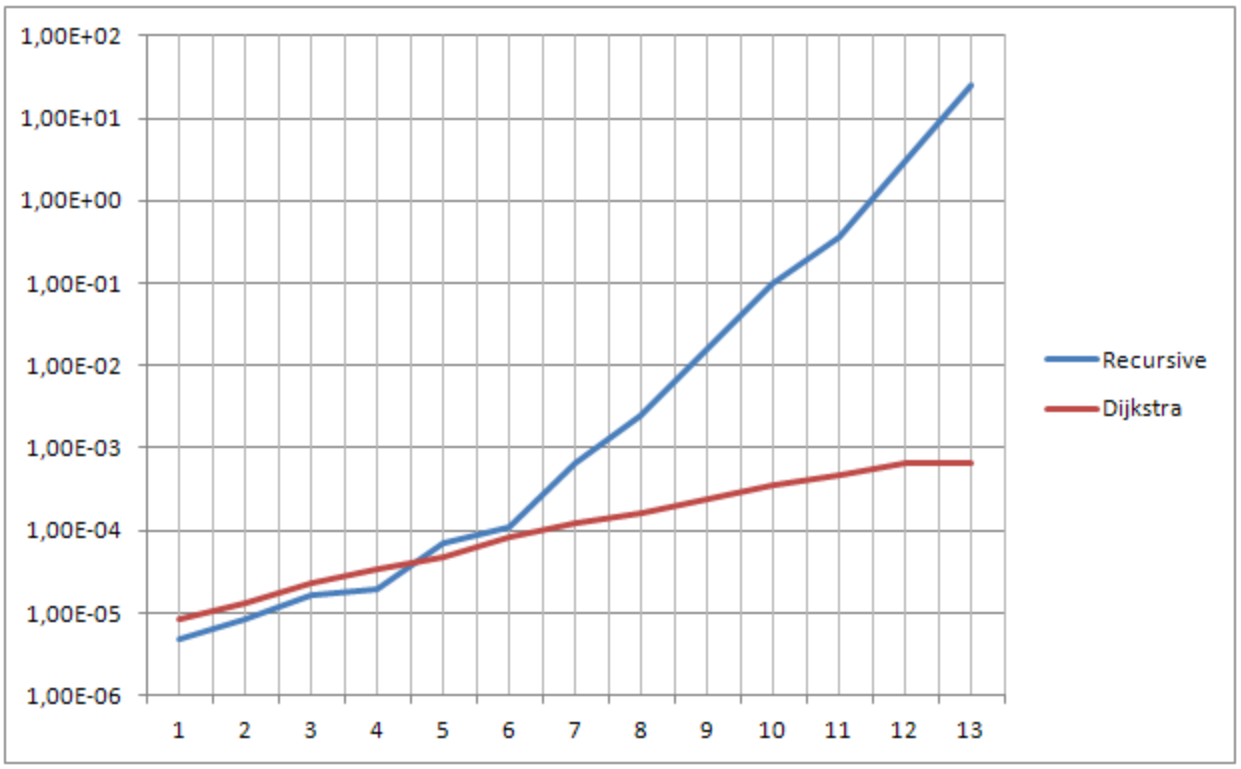
c.

Dijkstra's algoritme heeft n^2+e stappen nodig om volledig afgelopen te worden. hierbij staat n voor het aantal knopen en e voor het aantal edges tussen alle knopen. De complexiteit is dus $O(n^2)$. Immers is $O(n^2+e) \Leftrightarrow O(n^2)$

Resultaten:

n	Recursive	Dijkstra
2	4,71E-06	8,63E-06
3	8,33E-06	1,35E-05
4	1,60E-05	2,31E-05
5	1,97E-05	3,32E-05
6	7,06E-05	4,77E-05
7	1,10E-04	8,45E-05
8	6,68E-04	1,20E-04
9	2,56E-03	1,66E-04
10	1,59E-02	2,37E-04
11	1,01E-01	3,57E-04
12	3,59E-01	4,71E-04
13	3,04E+00	6,47E-04
14	2,54E+01	6,63E-04

René Aparicio Saez 10214054
Tom Peerdeman 10266186
Freddy de Greef 10287302



De verkregen resultaten zijn een gemiddelde van vijf runs. De hierbij gegenereerde grafiek is weergegeven met een logaritmische schaal.

René Aparicio Saez 10214054
Tom Peerdeman 10266186
Freddy de Greef 10287302

2) Zoek het kortste pad van s naar t door middel van een kortse pad vindt algoritme. Wil je ervoor zorgen dat de lengte van het kortste pad toeneemt moet er een kant worden weggenomen op het huidige gevonden kortste pad. Hiervoor moet een edge op het huidige kortste pas worden verwijderd. Bereken de nieuwe paden voor elke edge die weggehaald kan worden in het kortste pad. De edge die de hoogste waarde voor het nieuwe pad van s naar t oplevert moet worden verwijderd.

```
tryEdgeDelete(graph G, node s, node t){
        int length, newLength;
        int spath = [];
        int newSpath = [];
        lenght, spath = shortestPathDijkstra(G,s,t);
        G_buf = G;
        Edge toRemove;
        bool removedAndFound = false;
        for each EDGE in path{
                remove EDGE from G_buf;

                newLength, newSpath = shortestPathDijkstra(G_buf,s,t);
                removedAndFound = true;
                if(newLength > length){
                        length = newLength;
                        toRemove = EDGE;
                        spath = newSpath
                }

                G_buf = G;
        }
        if(removedAndFound)
                remove toRemove from G;
}
```

De complexiteit van dit algoritme is $O(n^3)$. Om alle Edges af te lopen in de for-loop zijn we worst-case $n-1$ stappen kwijt. Per Edge moet vervolgens een kortste pad gevonden worden aan de hand van een algoritme. Hierbij gaan we uit van Dijkstra's algoritme. Dit algoritme kost n^2 stappen. Dit geeft ons een totaal aantal stappen van $(n-1)\cdot n^2 = n^3 - n^2$. Dit is van orde $O(n^3)$.

René Aparicio Saez 10214054
Tom Peerdeman 10266186
Freddy de Greef 10287302

3)

a.

Het gegeven algoritme is een greedy algoritme omdat het voldoet aan alle eisen die bij een greedy algoritme horen. Omdat er gebruik wordt gemaakt van Depth First Search, wordt slechts één kant opgelopen in de graaf, en mag dus niet teruggekomen worden op een gemaakte 'loop' keuze. Het algoritme kiest vervolgens van een gevonden cykel de lokaal hoogste kant en verwijderd deze. Als de cykel is verbroken kan verder gezocht worden.

b.

Dit algoritme is van complexiteit $O(n)$. Alle nodes moeten worden afgelopen om te zoeken naar cykels. Wordt een cykel gevonden dan kan hij aan de hand van het gelopen pad, dat is onthouden, bepalen welke edge verwijderd moet worden (namelijk die met de hoogste waarde). Stel het algoritme zou de waardes niet hebben onthouden dan zou de complexiteit $n+n = 2n$ zijn, maar $O(2n) \Leftrightarrow O(n)$, de complexiteit blijft dan ongewijzigd.

René Aparicio Saez 10214054
Tom Peerdeman 10266186
Freddy de Greef 10287302

4)

```
hasIndependentSet(Graph G, k){
    set = hasIndependentSet(G.V, G.E)
    return len(set) >= k
}

hasIndependentSet(Vertices V, Edges E){
    if(len(V) == 1)
        return V
    else
        set1 = hasIndependentSet(V.subset(0, floor(len(V) / 2.0)) ?, E)
        set2 = hasIndependentSet(V.subset(ceil(len(V) / 2.0), len(V)), ?, E)
        set = notconnectedvertices(set1, set2, E)
        return set
}

notconnectedvertices(set1, set2, E){
    set = {0}
    for(v1 in set1):
        bool connected = false
        for(v2 in set2):
            for(edge in E):
                if(edge connects v1, v2)
                    connected = true
        if(not connected)
            set.add(v1)

    for(v2 in set2):
        bool connected = false
        for(v1 in set1):
            for(edge in E):
                if(edge connects v1, v2)
                    connected = true
        if(not connected)
            set.add(v2)

    return set
}
```

René Aparicio Saez 10214054
Tom Peerdeman 10266186
Freddy de Greef 10287302

Dit algoritme heeft een complexiteit van $O(n^3)$. Totaal zijn er voor het divide (splits) deel van het algoritme $(n-1)$ stappen nodig. Per divide worden vervolgens nog stappen uitgevoerd om te kijken welke knopen er niet verbonden zijn. In het worst-case geval kost het algoritme $2 \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot c = 2 \cdot \frac{1}{2}n \cdot \frac{1}{2}n \cdot c = 2 \cdot \frac{1}{4}cn^2 = \frac{1}{2}cn^2$ stappen. Waarbij $\frac{n}{2}$ het aantal nodes in de deelset is en c het aantal edges. Totaal kost het gehele algoritme dus in het worst-case geval $\frac{1}{2}cn^2 \cdot (n-1) = \frac{1}{2}cn^3 - \frac{1}{2}cn^2$ stappen. Dit is van orde $O(n^3)$. Er is uitgegaan van een gerichte graaf.

René Aparicio Saez 10214054
Tom Peerdeman 10266186
Freddy de Greef 10287302