II

1. The for loop will loop n-2 times, the complexity therefor is:

$$O(n)$$

2. ~~Algorithm:~~

~~temp ← first~~

~~while (temp.next != null)~~
~~display temp.key~~
~~temp ← temp.next~~

Algorithm:

```
temp ← first ;
display temp.key ;
while (temp.next != null)
        temp ← temp.next ;
        display temp.key ;
```

3. Algorithm:

~~first elem ←~~
firstElement ← first ;
secondElement ← first.next ;
element ← new (simple-list);
element.next ← first.next ;
first ← element ;
element.key ← 1000 ;

4. Algorithm

```
temp ← first

while (temp.next != null)
        temp ← temp.next

element = new (simple_list);
element.key = 1000 ;
element.next = null ;
temp.next = element;
```

The while loop will loop n times for n elements, the complexity therefore is:

$$O(n)$$

5 We can add a pointer to first pointing to the end of the list. In that way, adding to the end of the list is $O(1)$

2.

1 & 2 a. Function isEmpty()
return no_elements == 0 ;

Complexiteit: $O(1)$

Function push(element)
~~stack[0] ← elem~~
stack(no_elements) ← element ;
no_elements ++ ;

Complexiteit: $O(1)$

Function pop()
no_elements -- ;
return stack[no_elements] ;

Complexiteit: $O(1)$

Function top()
return stack[0] ;

Complexiteit: $O(1)$

Function display()
for (int i = 0 ; i < no_elements ; i++)
display (stack[i]) ;

Complexiteit: $O(n)$

3.

$$\begin{bmatrix} AAA \\ AAA \\ BBB \\ CCC \\ CCC \\ AAA \\ BBB \end{bmatrix}$$

4. Function empty()

~~for (int i = 0; i < no_elements; i++)~~

stack

no_elements = 0

3.

1&2

```
Function isEmpty()
    ~~if stack ==~~
    return stack == null
```

Complexiteit: $O(1)$

```
Function push(element)
    temp ← element
    stack ← temp
    temp.next ← null
    temp.previous ← stack
```

```
Function pop()
    temp ← stack
    stack ← stack.previous
    stack.next ← null
    return temp
```

~~Function top()~~ Complexiteit beide: $O(1)$

```
Function top()
    ~~while (~~ temp ← stack ;
    while (~~stack~~ temp.previous != null)
        temp ← ~~to~~ temp.previous ;
    return temp;
```

Complexiteit: $O(n)$

¶

Function display()

```
temp ← top()
while (temp.next != null)
    display temp
    temp.~~next~~ ← temp.next
```

Complexiteit: $O(n)$

c.

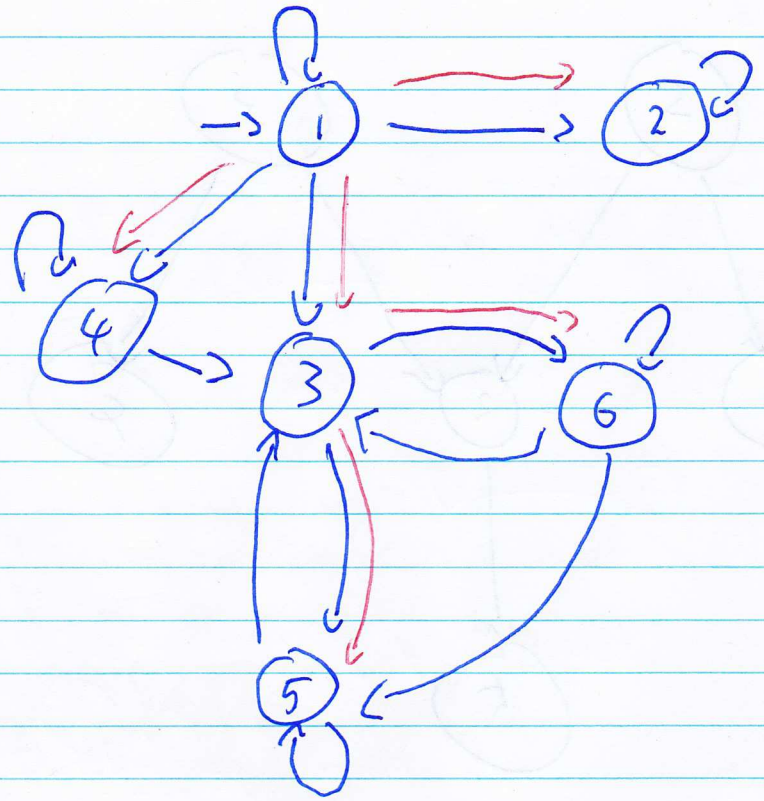
```
[ AAA
  AAA
  BBB
  CCC
  CCC
  AAA
  BBB ]
```

d. Function empty()

~~temp ← stack~~ ~~temp ← top()~~

stack ← null

4.



1.
Q: 1
D: 2, 3, 4
F: 1

2.
Q: 2, 3, 4
D:
F 1, 2

3.
Q: 3, 4
D: 6, 5
F: 1, 2, 3

4.
Q: 4, 6, 5
D:
F: 1, 2, 3, 4

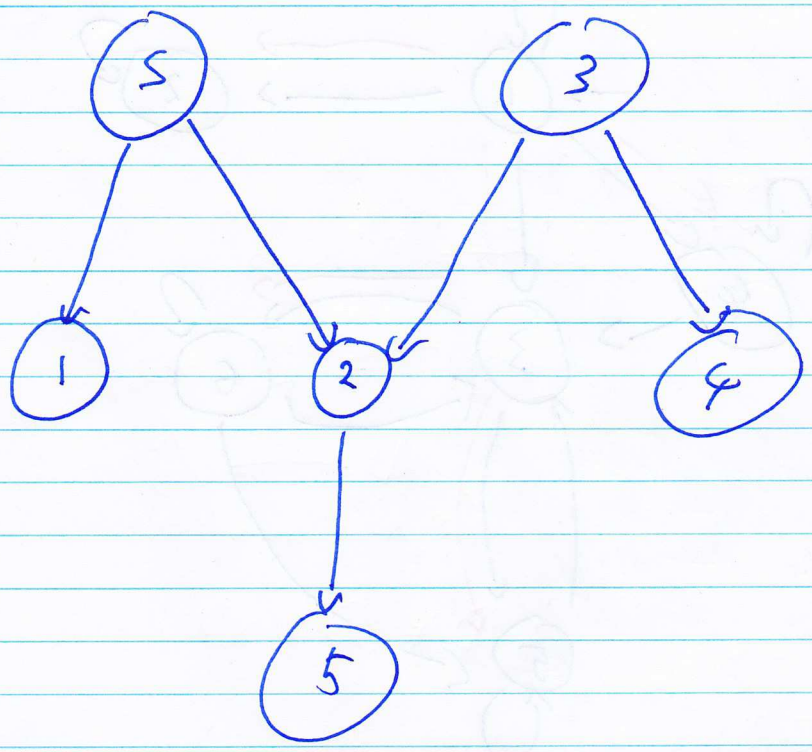
5.
Q: 6, 5
D:
F: 1, 2, 3, 4, 6

6.
Q: 5
D:
F: 1, 2, 3, 4, 6, 5

7:
Q:
D:
F: 1, 2, 3, 4, 6, 5

Notatie:
Q (queue) : eerste element is wat die mee bezig is
D (discovered) : ontdekt in deze beurt
F (finished) : al onderzocht.

5.



6. ~~Algorithm~~

```
Q ← ∅
enqueue (Q, s)
u ← null;
while (Q ≠ ∅)
    let u ← dequeue (Q)
    for each v ∈ Adj (u)
        enqueue (Q, v)
        previous [v] ← u
    if (u == s)
        empty (Q)
```

Algorithm

```
Q ← ∅
previous [s] ← null;
u ← null;
enqueue (Q, s);
~~while (~~
while (Q ≠ ∅)
    let u ← dequeue (Q);

    for each v ∈ Adj [u]
        enqueue (Q, v);
        previous [v] ← u

    if (u = s)
        empty (Q);

return u;
```

u heeft dan als previous een andere knoop, die dan ook weer een andere knoop heeft, tot s.

7. Algorithm

```
for each vertex s in V[G]
    for each vertex u in V[G]
        color[u] ← white

    color[s] ← grey

    Q ← ∅
    enqueue(Q, s)

    while Q ≠ ∅
        let u dequeue(Q)
        for each v ∈ Adj[u]
            if previous[v] ≠ u
                if color[v] = white
                    color[v] ← grey
                    previous[v] ← u
                    enqueue(Q, v)
                else if color[v] == black or grey
                    return true

        color[u] ← black

return false
```

Alle knopen worden via BFS bewegwegt als het niet de vorige knoop was. Als we een grijze of zwarte tegenkomen weten we dat er ergens een cirkel is.