1) Speicherdarstellung von Graphen **richtig** **falsch**

1.1) Die Darstellung eines Graphen als Adjazenzliste eignet sich besonders für Graphen mit wenigen Kanten, während die Darstellung als Adjazenzmatrix für Graphen mit vielen Kanten günstiger ist.

1.2) Bei gerichteten Graphen ist die Adjazenzmatrix symmetrisch, man kann sie also intern als Dreiecksmatrix abspeichern.

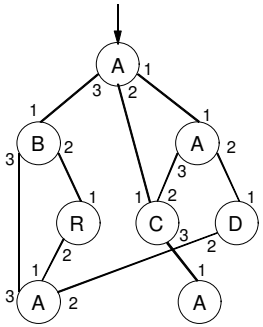
1.3) Normalerweise enthält die Adjazenzmatrix nur Boolesche Einträge (Kante vorhanden oder nicht). Bei gewichteten Graphen enthält sie allerdings die Kantengewichte.

1.4) Das Traversieren eines Graphen über Adjazenzlisten ($O(n)$) geht schneller als das Traversieren eines Graphen über die Adjazenzmatrix ($O(n^2)$).

2) Breadth-First-Search

Gegeben ist folgender Graph. Geben Sie an, in welcher Reihenfolge die Knoten des Graphen besucht werden, wenn Sie den Graph mittels BFS beginnend bei A durchwandern, wobei die Söhne aufsteigend nach ihren Nummern besucht werden. Die Verbindungen zwischen den einzelnen Knoten sind bidirektional. Wenn ein Knoten mehrmals besucht würde, führen Sie auch diese Besuche (z.B. in Klammer) an, ohne allerdings in endlose Schleifen zu geraten.

```
PROCEDURE BFS (node: Node);
   VAR i: INTEGER; son: Node;
BEGIN
   InitQueue;
   node.mark := mark;
   REPEAT
      (* ... visit node ... *)
      i := 0; son := node.son[i];
      WHILE son # NIL DO
         IF son.mark # mark THEN
            Enqueue(son); son.mark := mark;
         END ;
         INC(i); son := node.son[i]
      END ;
      n := Dequeue()   (* returns NIL if empty *)
   UNTIL node = NIL
END BFS;
```



3) Depth-First-Search

Geben Sie an, in welcher Reihenfolge die Knoten des Graphen aus Aufgabe 2 besucht werden, wenn Sie den Graph mittels DFS beginnend bei A durchwandern, wobei die Söhne aufsteigend nach ihren Nummern besucht werden. Die Verbindungen zwischen den einzelnen Knoten sind bidirektional. Wenn ein Knoten mehrmals besucht würde, führen Sie auch diese Besuche (z.B. in Klammer) an, ohne allerdings in endlose Schleifen zu geraten.

```
PROCEDURE DFS (node: Node);
   VAR i: INTEGER; son: Node;
BEGIN
   node.mark := mark;
   (* ... visit node ... *)
   i := 0; son := node.son[i];
   WHILE son # NIL DO
      IF son.mark # mark THEN DFS(son) END ;
      INC(i); son := node.son[i]
   END
END DFS;
```