4.7

```
Zusammenhangskomponenten( G = (V,E) ) {

    für alle v ∈ V {
        comp[v] = 0;
        makeset(v);
    }

    i = |V|;

    für alle e = (u,v) ∈ E {
        falls findset(u) ≠ findset(v) dann {
            union( findset(u), findset(v) );
            i = i - 1;
        }
    }

    für alle v ∈ V {
        r = findset(v);
        falls comp[r] == 0 dann {
            comp[r] = i;
            i = i - 1;
        }
        comp[v] = comp[r];
    }

}
```

(a) findset: $4*|E| + |V|$
union: $|E|$

(b) $O(2*|V| + |E|) = O(|V| + |E|)$

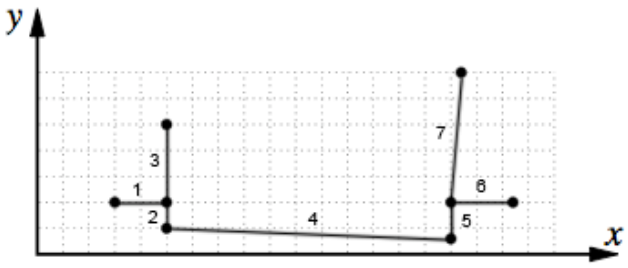
(c) DFS: $O(|V| + |E|)$, die Laufzeiten sind also ident

4.8

(a)

```
Prim-MST-Grad-3( G ) {
    Wähle einen beliebigen Startknoten s ∈ V;
    C = { s };
    T = ∅;
    solange |C| ≠ |V| {
        Ermittle eine Kante e = (u,v) mit u ∈ C und grad(u) < 3, v ∈ V \ C und minimalem Gewicht w_e;
        T = T ∪ { e };
        C = C ∪ { v };
        grad(u) = grad(u) + 1;
        grad(v) = grad(v) + 1;
    }
}
```

(b)



(c)

w1 > 6 w2 + 1 < w1
w2 > 5

4.9

(a) Fehler in der Angabe: Gesucht sind alle Hamiltonschen Pfade, ...

P ... Pfadliste
V ... aktueller Knoten

Pseudocode 1:

```
SucheHamil( P, V ) {
    für alle w ∈ N(v) {
        falls (w in P) continue;
        // -> w neuer Knoten für Pfad P
        falls P.length + 1 == |V| dann {
            gib Pfad P + w aus;
        }
        SucheHamil( P + w, w );
    }
}
```

Pseudocode 2:

```
mark[v] = false;
SucheHamil( P, V ) {
    für alle w ∈ N(v) {
        falls (mark[w] == false) continue;
        // -> w neuer Knoten für Pfad P
        falls P.length + 1 == |V| dann {
            gib Pfad P + w aus;
        }
        mark[w] = true;
        SucheHamil( P + w, w );
        mark[w] = false;
    }
}
```

(b)

O(n!)
Worst Case: vollständig verbundener Graph

4.10

FRP {

für i = 1, ..., n {
// Wert / Gewicht
s(i) = v(i) / w(i);
}

sortiere s(i) absteigend;
w = 0;
i = 1;
solange $w + w(i) < w_{max} \wedge i \leq n$ {
packe a_i ein;
w = w + w(i);
i = i + 1;
}
packe w_{max} - w Einheiten von a_i ein;

}

Laufzeit:

Berechnung der s(i): linear in n
Sortieren: Theta(nlogn)
Befüllen: linear in w_{max}

=> Theta(nlogn)

4.12

M0 = { (∅, 0, 0) }

M1 = { (∅, 0, 0), ({1}, 4, 2) }

M2 = { (∅, 0, 0), ({2}, 2, 1), ({1}, 4, 2), ({1,2}, 6, 3) }

M3 = { (∅, 0, 0), ({3}, 6, 5), ({2}, 2, 1), ({1}, 4, 2), ({1,2}, 6, 3) }

M4 = { (∅, 0, 0), ({3}, 6, 5), ({4}, 4, 3), ({2}, 2, 1), ({2,4}, 6, 4), ({1}, 4, 2), ({1,4}, 8, 5), ({1,2}, 6, 3) }

M5 = { (∅, 0, 0), ({4}, 4, 3), ({3}, 6, 5), ({2}, 2, 1), ({2,4}, 6, 4), ({1}, 4, 2), ({1,4}, 8, 5), ({1,2}, 6, 3) }

optimale Auswahl: ({1,4}, 8, 5)

4.13

d[v] ... Abstand

```
Dijkstra( G, V, t ) {

    for ( all v ∈ G ) {
        d[v] = ∞;
        n[v] = NULL;
    }

    d[t] = 0;
    Q = V;

    while ( Q not empty ) {
        nimm u ∈ Q mit min. d[u];
        for ( all v ∈ N(u) ) {
            if ( d[v] > d[u] + w( u,v ) ) {
                d[v] = d[u] + w( u,v );
                n[v] = u;
            )
        }
    }

    for ( all v ∈ V ) {
        gib aus: V → n[v], d[v];
    }

}
```

4.14

LB ... Lower Bound
UP ... Upper Bound
OP ... Open Problems
Profit ... Punkte / Zeit

sortiere Algorithmen nach Profit: Profit(Ai) ≥ Profit(Ai+1)
i = 1, ..., n

```
Best.LB = -1;
OP.addProblem( ∅, A1, 1, Best );
OP.addProblem( ∅, ¬A1, 1, Best );
while ( OP ≠ leer ) {
    Problem = OP mit höchster LB;
    if( (P.UB < Best.LB) or (P.LB == P.UB) ) continue;
    i = P.num + 1;
    OP.addProblem( P, Ai, i, Best );
    OP.addProblem( P, ¬Ai, i, Best );
}
```

```
OPEN PROBLEMS.add Problem ( P1, P2, i, Best ) {
    füge P1 ∧ P2 zusammen zu Problem P;
    P.num = i;
    berechne LB ∧ UB für P;
    if ( P.LB > Best.LB ) Best = P;
    if (P.UB > Best.LB) {
        füge P in OP ein;
    }
}
```

(4.15)

	Punkte	Zeit	Profit (Punkte/Zeit)
A	7	40	0,175
B	3	10	0,3
C	10	60	$0,1\overline{6}$
D	4	20	0,2
E	13	90	$0,1\overline{4}$

$B \geq D \geq A \geq C \geq E$