Prof. Dr. Dirk Riehle, M.B.A.
Prof. Dr. Michael Philippsen
Department Informatik, FAU

30. Juli 2015

Klausur: Algorithmen und Datenstrukturen

Angaben zur Person (Bitte Etikett aufkleben bzw. in Druckbuchstaben ausfüllen!):

Name, Vorname: ....

Laufende Nr.: ....

Bitte kleben Sie hier das Etikett auf.

Matrikelnummer:

....

Folgende Hinweise bitte lesen und Kenntnisnahme durch Unterschrift bestätigen!

- **Hilfsmittel außer Schreibmaterialien sind *nicht* zugelassen**.
- Lösungen müssen in den dafür vorgesehenen freien Raum geschrieben werden. Sollte der Platz nicht ausreichen, verwenden Sie zunächst (mit kurzem Hinweis) die Zusatzseite am Ende. Weitere Zusatzseiten müssen von der Aufsicht ausgegeben und *eingeheftet* werden.
- Sie können Schmierpapier von der Aufsicht anfordern. ***Das Schmierpapier darf nicht mit abgegeben werden.***
- Können Sie die Prüfung aus gesundheitlichen Gründen nicht fortsetzen, dann müssen Sie Ihre Prüfungsunfähigkeit durch Vorlage eines *erweiterten ärztlichen Attestes* beim Prüfungsamt nachweisen. Melden Sie sich in jedem Fall bei der Aufsicht und lassen Sie sich das *entsprechende Formular* aushändigen.
- ***Überprüfen Sie diese Klausur auf Vollständigkeit (16 Seiten inkl. Deckblatt) und einwandfreies Druckbild!***

Durch meine Unterschrift bestätige ich den Empfang der vollständigen Klausurunterlagen und die Kenntnisnahme der obigen Informationen.

Erlangen, den 30.07.2015 ....
(Unterschrift)

Nicht von der Kandidatin bzw. vom Kandidaten auszufüllen!

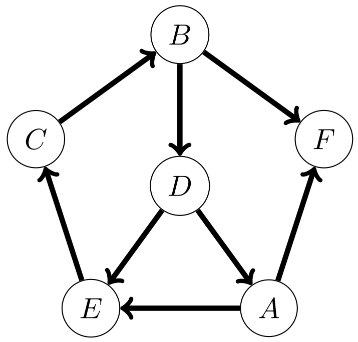
Bewertung (Punkteverteilung unter Vorbehalt):

Aufgabe	1	2	3	4	5	6	7	8	$\sum$
Maximal	16	9	18	11	15	17	16	18	120
Erreicht	6	4	5	4	7				26

Aufgabe 1 (Wissensfragen) (16 Punkte)

Bei den folgenden Teilaufgaben werden richtige Kreuze positiv (+) und falsche **oder fehlende** Kreuze entsprechend negativ (–) gewertet. Jede Teilaufgabe wird mit mindestens 0 Punkten bewertet. Pro Teilaufgabe ist mind. eine Aussage wahr. Kreuzen Sie alle richtigen Aussagen an:

a) Je nach Darstellung eines Graphen können Breiten- (BFS) und Tiefensuche (DFS) unterschiedliche Ergebnisse liefern. Welche sind im folgenden Beispiel möglich?



- ☒ BFS $\rightsquigarrow D, E, A, C, F, B$
- ☐ BFS $\rightsquigarrow D, A, E, C, B, F$
- ☐ DFS $\rightsquigarrow D, E, A, C, F, B$
- ☒ DFS $\rightsquigarrow D, A, E, C, B, F$

2 P

b) Aus der Vorlesung kennen Sie den ADT `Set<E>` mit folgenden **ops**. Welche **axs** passen?

adt *Set*
sorts *Set,E,Boolean*
ops

empty:		$\mapsto$ *Set*	// $empty \triangleq \emptyset$
add:	$Set \times E$	$\mapsto$ *Set*	// $add(S,\ e) \triangleq S \cup \{e\}$
union:	$Set \times Set$	$\mapsto$ *Set*	// $union(A,\ B) \triangleq A \cup B$
intersect:	$Set \times Set$	$\mapsto$ *Set*	// $intersect(A,\ B) \triangleq A \cap B$
diff:	$Set \times Set$	$\mapsto$ *Set*	// $diff(A,\ B) \triangleq A \setminus B$

...

axs

...

- ☒ *diff(S, empty) = S*
- ☒ *union(S, empty) = S*
- ☐ *union(add(A, e), B) = add(B,e)* A fehlt
- ☐ *intersect(A, B) = diff(diff(union(A,B),A),B)* $\triangleq \emptyset$

...

end *Set*

2 P

c) Einen *minimalen Spannbaum* für einen Graphen mit n Knoten und e Kanten kann man mit dem Algorithmus von ...

- ☐ *Floyd* in $\mathcal{O}(\sqrt{n \cdot e})$ Zeit bestimmen.
- X ☐ *Dijkstra* in $\mathcal{O}(n \cdot (e + n \cdot \log(n)))$ Zeit bestimmen.
- ☒ *Kruskal* in $\mathcal{O}(e \cdot log(e))$ Zeit bestimmen.
- ☒ *Prim* in $\mathcal{O}(n \cdot log(n))$ Zeit bestimmen.

0P

d) Betrachtet man in einem *vollständigen* UML-Klassendiagramm (d.h. Vererbungshierarchie bis einschließlich `Object` explizit dargestellt) eines beliebigen Java-Programms alle Klassenkästchen als Knoten sowie alle Linen als Kanten eines Graphen und zeichnet ...

☒ nur `extends`-*Linien* (ungerichtet) ein, erhält man *genau einen* Baum.

☐ nur `implements`-*Linien* (ungerichtet) ein, ist der Graph zusammenhängend. 0P

☐ nur `implements`-*Pfeile* (gerichtet!) ein, kann der Graph Zyklen enthalten.

☐ nur `extends`- *und* `implements`-*Pfeile* (gerichtet!) ein, ergibt sich ein DAG.

e) Was gilt für eine Streutabelle mit m Fächern?

☐ $h(k) = k\%7$ ist eine gute Streufunktion, weil 7 eine Primzahl ist.

☐ $h(k) = (k\%b)\%m$ ist eine gute Streufunktion, wenn b eine Zweierpotenz ist. 0P

☐ Enthält die Tabelle n Elemente, dann ist ihr Belegungsfaktor: $BF = \frac{n}{m}$.

☐ Der Lastfaktor kann bei offenem Hashing auch *größer* als 100% sein.

f) Welche der folgenden Aussagen ist/sind korrekt?

☐ Ein AVL-Baum mit 666 Knoten kann links-vollständig sein.

☐ Eine Max-Halde muss nicht unbedingt balanciert sein. 0P

☐ Die worst-case-Laufzeit für die Suche in einem Binärbaum ist $\mathcal{O}(log_2(n))$.

☐ Ein AVL-Baum mit > 2 paarweise verschiedenen Knoten kann die Max-Halden-Eigenschaft erfüllen.

g) Wonach entscheiden der Übersetzer bzw. das Laufzeitsystem in Java, welches gleichnamige Attribut verwendet bzw. welche Methode aufgerufen werden soll?

☐ Verdecktes Instanzattribut: nach dem dynamischem Typ an der Zugriffsstelle

☒ Überladene Instanzmethode: nach den Datentypen der Parameter 2P

☒ Überschriebene Instanzmethode: nach dem dynamischen Typ an der Zugriffsstelle

☐ Verdecktes Klassenattribut: nach dem dynamischen Typ an der Zugriffsstelle

h) Gegeben sind ein kreuzungsfreies Polygon $\mathcal{P}$ aus n Strecken sowie zwei verschiedene Punkte $\mathcal{A}$ und $\mathcal{B}$, wobei $\mathcal{A}$ sicher außerhalb des Polygons liegt. Beim Punkt-in-Polygon-Problem prüft man, wie oft die Verbindung zwischen $\mathcal{A}$ und $\mathcal{B}$ das Polygon schneidet. Die *kleinsten oberen* Schranken für die Laufzeit $\mathcal{T}_n$ und den zusätzlichen Speicherbedarf $\mathcal{S}_n$ dieser Prüfung sind:

☒ $\mathcal{T}_n \in \mathcal{O}(n)$

☐ $\mathcal{T}_n \in \mathcal{O}(\sqrt{n})$ 0P

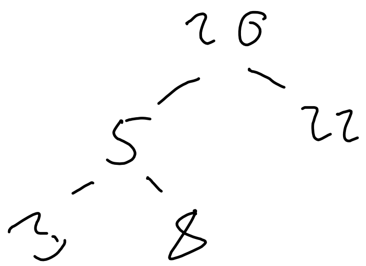
☐ $\mathcal{S}_n \in \mathcal{O}(log(n))$

☐ $\mathcal{S}_n \in \mathcal{O}(1)$

Aufgabe 2 (Bäume) (9 Punkte)

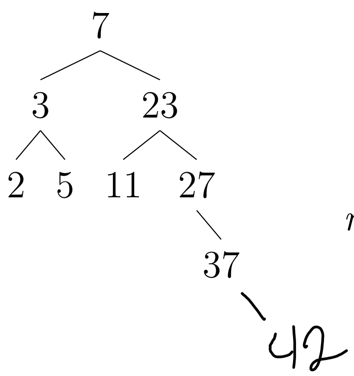
a) Fügen Sie die Zahlen 20, 22, 3, 8, 5 in der gegebenen Reihenfolge in einen *binären Suchbaum* mit aufsteigender Sortierung ein. Stellen Sie nur das Endergebnis dar:

Binärer Suchbaum:

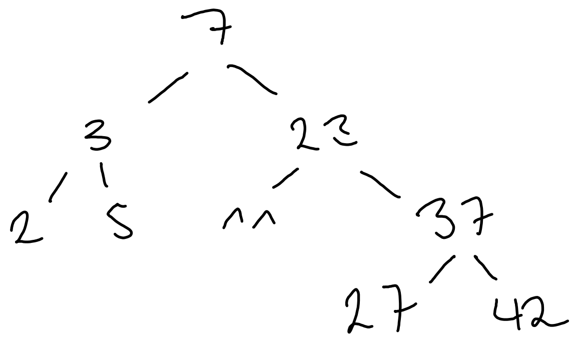


b) Fügen Sie die 42 in jeden der folgenden *AVL-Bäume* ein (samt Kante direkt in den links angegebenen Baum einzeichnen). Führen Sie anschließend bei Bedarf die erforderliche(n) Rotation(en) aus und stellen Sie nur dann das Ergebnis rechts von $\Longrightarrow$ dar:

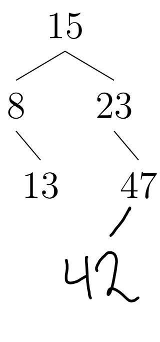
i) AVL-Baum $\mathcal{A}$:



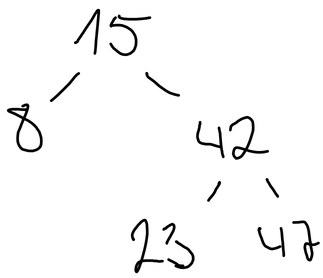
? $\Longrightarrow$ *rotieren*



ii) AVL-Baum $\mathcal{B}$:



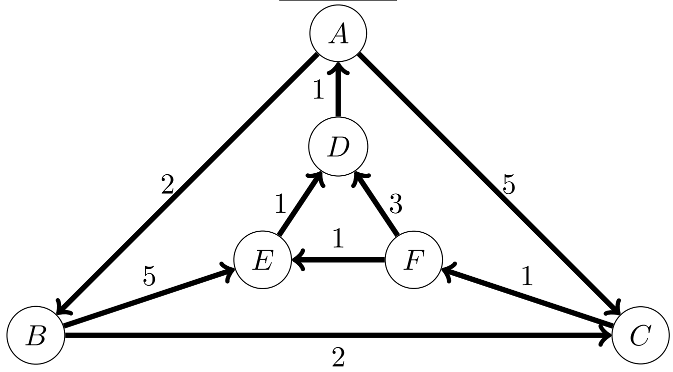
? $\Longrightarrow$ *rotieren*



Aufgabe 3 (Graphen) (18 Punkte)

a) Ermitteln Sie mit dem Algorithmus von *Dijkstra* den kürzesten Weg vom Knoten A zu allen erreichbaren Knoten in $\mathcal{G}$. Verwenden Sie zur Lösung die unten stehende Tabelle und markieren Sie in jeder Zeile den jeweils als nächstes zu betrachtenden Knoten.

Graph $\mathcal{G}$:



A	B	C	D	E	F
$\boxed{0}$	∞	∞	∞	∞	∞
0	2	5	∞	∞	∞
0	2	4	∞	7	∞
0	2	4	∞	6	7
0	2	4	8	6	7
0	2	4	7	6	7

Endergebnis

0	2	4	7	6	7

b) Geben Sie den kürzesten Weg und dessen Weglänge vom Knoten A zum Knoten D an:

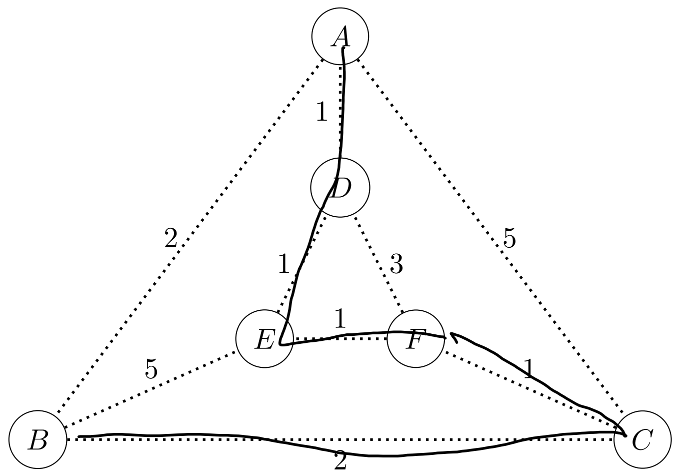
$A \longrightarrow$ B → C → F → E $\longrightarrow D$

Kürzeste Weglänge ($A \rightsquigarrow D$): 7

c) Betrachten Sie den Graphen $\mathcal{G}$ nun als *ungerichtet* und geben Sie die Kanten von $\mathcal{G}$ in der Reihenfolge an, in welcher der Algorithmus von *Prim* sie in den Spannbaum aufnimmt, falls er beim Knoten B startet.

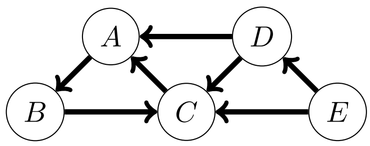
(B, C)
(C, F)
(F, E)
(E, D)
(D, A)
(____, ____)
(____, ____)

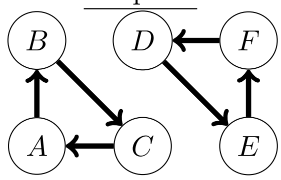
Schmiergraph zu Ihrer Unterstützung:
*(wird **nicht** bewertet)*

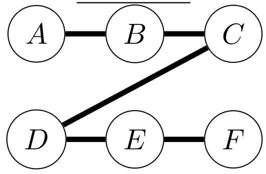


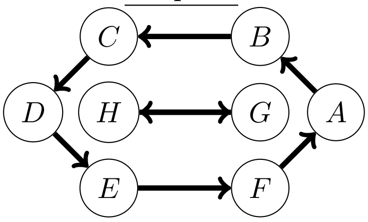
1P

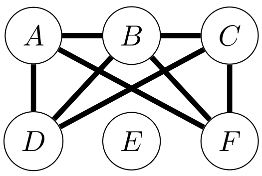
d) Klassifizieren Sie die folgenden Graphen entsprechend *in genau eine* Kategorie:

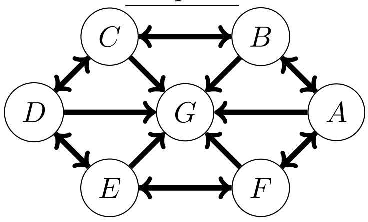
Graph 1:


Graph 2:


Graph 3:


Graph 4:


Graph 5:


Graph 6:


3P

Stark zusammenhängend:	1, 6	kein
Schwach zusammenhängend:	2	1, 6
Zusammenhängend:	3	✓
Nicht zusammenhängend:	4, 5	2, 4, 5

Aufgabe 4 (Rekursion) (11 Punkte)

Die *Potenzmenge* $\mathcal{P}(s)$ einer Menge s ist die Menge aller Teilmengen von s, wobei sowohl die leere Menge $\emptyset$ als auch s selbst zu den Teilmengen von s gehören. Beispiel:

$$\mathcal{P}(\{A, u, D\}) = \{\emptyset, \{A\}, \{u\}, \{D\}, \{A, u\}, \{A, D\}, \{u, D\}, \{A, u, D\}\}$$

Die Methode `potenzmenge` soll $\mathcal{P}(s)$ bestimmen. Die Eingabe wird hier vereinfacht durch ein Array dargestellt, wobei alle Elemente in `s` garantiert paarweise verschieden sind. Ergänzen Sie die Methode `helfer`, die die Potenzmenge von `s` ab dem Index `idx` berechnen soll.

```
static <T> List<List<T>> potenzmenge(T[] s) {
    // bestimme Potenzmenge ab Index 0
    return helfer(s, 0);
}

static <T> List<List<T>> helfer(T[] s, int idx) {
    // Rueckgabe pms ist Potenzmenge von s ab Index idx
    List<List<T>> pms = new ArrayList<List<T>>();
```

```
    if ( idx >= s.length())
```

```
        // Basisfall
```

```
            return pms;
```

```
    } else {
        // aktuelles Kopfelement bestimmen
```

```
        T kopf = s[idx];
```

```
        // Potenzmenge der Restliste bestimmen
```

```
        List<List<T>> potRest = helfer(s, idx+1);
```

```
        // Ergebnisse zusammenfuehren
        for (List<T> ohneKopf : potRest) {
            List<T> mitKopf = new ArrayList<>(ohneKopf); // *noch* ohne Kopf
```

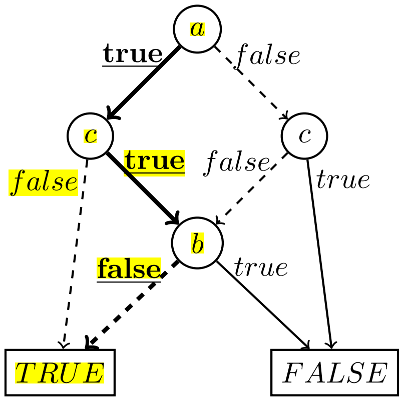
```
            mitKopf.add(kopf);
            pms.add(mitKopf);
            pms.add(ohneKopf);
```

```
        }
    }

    return pms;
}
```

Aufgabe 5 (Gerichtete azyklische Graphen) (15 Punkte)

Ein *Binary Decision Diagram (BDD)* ist ein DAG (gerichteter azyklischer Graph) mit einer Wurzel zur Darstellung Boolescher Funktionen. BDDs haben ein oder zwei Blätter (*TRUE* und/oder *FALSE*). Abgesehen von den Blättern hat jeder Knoten des BDDs einen Variablennamen und genau zwei ausgehende Kanten, die mit *true* oder *false* beschriftet sind.



Das Beispiel rechts zeigt den BDD für die Funktion $f(a, b, c)$:

`(a && !b && c) || (a && !c) || (!a && !b && !c)`

Um z.B. $f(true, false, true)$ auszuwerten, folgt man dem Pfad von der Wurzel zu einem Blatt, welcher die jeweilige Variablenbelegung $a = true$, $b = false$, $c = true$ darstellt (im Bild hervorgehoben).

Für die Darstellung von BDDs sei folgende Klasse vorgegeben:

```
public class BDD {
    static final BDD TRUE = new BDD(); // true-leaf
    static final BDD FALSE = new BDD(); // false-leaf
    BDD trueC, falseC; // true child respectively false child
    String v; // variable name (null, iff in a leaf)
```

Hinweis: `HashSet<E>` hat u.a. die Methoden `add(E)`, `contains(Object)` und `remove(Object)`.

a) Ergänzen Sie die *rekursive* Methode `eval(tv)`, die den BDD für die Variablenbelegung `tv` auswertet. Dabei sei die Variable mit Namen `this.v` genau dann `true`, wenn sie im HashSet `tv` vorkommt. Sie dürfen annehmen, dass der aktuelle Knoten die Wurzel eines gültigen BDDs ist. *Hinweis:* `this` kann auch `TRUE` oder `FALSE` sein.

```
boolean eval(HashSet<String> tv) {
if (this == BDD.true) return true;
if (this == BDD.false) return false;
if (tv.contains(v))
    return trueC.eval(tv);
else
    return falseC.eval(tv);
}
```

b) Ergänzen Sie die <u>*rekursive*</u> Methode `isDAG(p)` so, dass `isDAG()` genau dann `true` zurückgibt, wenn `this` die Wurzel eines azyklischen Graphen ist. Anstatt wie in Teilaufgabe a), verwenden Sie dieses Mal ein `HashSet p` für die Verwaltung bestimmter besuchter Knoten. Ihr Code muss **nicht** effizient sein.

```
boolean isDAG() {
    return isDAG(new HashSet<BDD>());
}

boolean isDAG(HashSet<BDD> p) {
```

```
//wurde der Knoten bereits besucht
if (p.contains(this)) return false;
p.Add(this);

if (trueC != null && !trueC.isDAG(p))
    return false;
if (falseC != null && !falseC.isDAG(p))
    return false;
p.Remove(this);
return true;
```

```
}
```

Aufgabe 6 (Dynamische Programmierung) (17 Punkte)

In der Zahlenfolge f_n für $n \geq 1$ kommt jede Zahl $i \geq 1$ geordnet jeweils $(2 \times i)$-fach vor:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	20	21	...
f_n	1	1	2	2	2	2	3	3	3	3	3	3	4	4	4	4	...	4	5	...
	2×1		2×2				2×3						2×4							...

Der folgende Code-Ausschnitt zeigt die Methode `f(n)`, die das n-te Glied der Folge ermittelt:

```
public class DynProg {
    int f(int n) {
        return n < 3 ? 1 : f(n - 2 * f(n - f(n - 1))) + 1;
    }
```

a) Um welche Art der Rekursion handelt es sich bei `f(n)`?

`f(n):` verschachtelte Rekursion

b) Mittels *Dynamischer Programmierung* sollen alle f_n für $n \leq max$ höchstens einmal berechnet werden. Erweitern Sie die Klasse und den Konstruktor um die notwendigen Datenstrukturen:

```
int[] dp;
```

```
DynProg(int max) {
    dp = new int[max];
}
```

c) Ergänzen Sie nun die Methode `fDP(n)`, die für $n \leq max$ *Dynamische Programmierung* nutzt und für $n > max$ weiterhin korrekte Ergebnisse liefert:

```
int fDP(int n) { // Dynamische Programmierung
    int fn = 1; // Basisfall n < 3 trivial
    // fn schon einmal berechnet?
    if (dp[n-1] != 0 && n <= dp.length)
        fn = dp[n-1];          siehe f(n) oben
    } else if (n >= 3) { // fn muss noch berechnet werden
        fn = fDP(n - 2* fDP(n - fDP(n-1))) +1;
        if (n <= dp.length) // n kleiner max?
            dp[n-1] = fn; // speichern
    }
    return fn;
}
```

d) Schreiben Sie eine <u>iterative</u> Methode `fIter(n)` zur Berechnung von f_n.
Tipp: Erhöhen Sie `k` von 1 ausgehend in mindestens einer Schleife, bis `k == n` erreicht ist. Vor und nach jeder Iteration soll `fk` den Wert f_k gemäß Tabelle in der Aufgabenstellung enthalten.

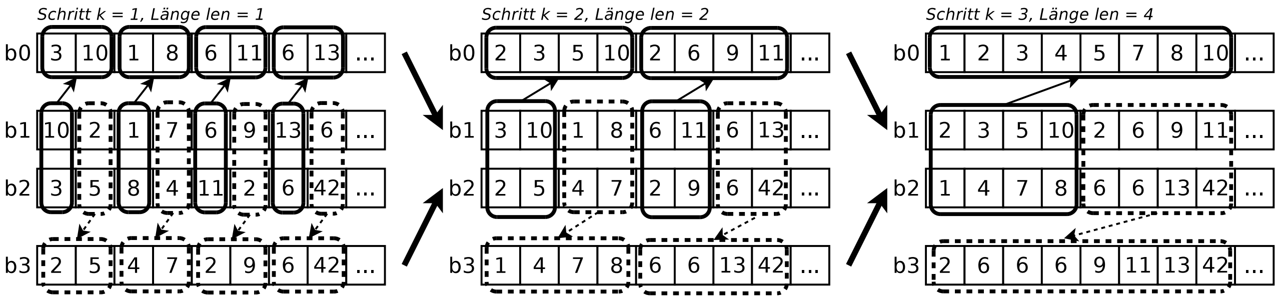
```
int fIter(int n) { // Iterativ
    int k = 1; // Zaehler
    int fk = 1;
```

```
int x = 2* fk;
while(k <= n) {        // x=2; bis k>x    ist fk = 1
                       //      ab k==3  ist fk = 2
    k++;               // x=2+2*fk=6; ab 7  ist fk=3...
    if ( k > x ) {
        fk++;
        x += 2* fk;
    }
}
```

```
}
```

Aufgabe 7 (Sortieren) (16 Punkte)

MergeSort kann als externes Sortierverfahren für Dateien verwendet werden, die nur sequentiell verarbeitet werden können. Das Verfahren aus der Vorlesung wird hier wie folgt leicht angepasst:
▶ Die sequentiellen Dateien werden durch vier `LinkedList<Integer>` simuliert.
▶ Sortiert wird auf vier „Hilfsbändern" b_0, b_1, b_2 und b_3.
▶ In der Vorbereitung wird die Eingabe je zur Hälfte auf b_1 und b_2 aufgeteilt.
▶ In jedem `merge`-Schritt $k = 1, 2, 3, \ldots$ werden Blöcke zu jeweils bis zu `len` $= 2^{k-1}$ Elementen von den Bändern b_1 und b_2 *aufsteigend* sortiert ineinander gemischt und abwechselnd auf die Zielbänder b_0 bzw. b_3 geschrieben. Nach jedem Schritt k werden jeweils die Bänder b_0/b_1 bzw. b_2/b_3 vertauscht:



▶ Falls das Zielband b_3 nach einem `merge`-Schritt leer bleibt, ist kein weiterer Schritt notwendig.
▶ Ihnen stehen die folgenden Methoden der Klasse `LinkedList` zur Verfügung:

- `isEmpty`: Returns `true` if this collection contains no elements.
- `getFirst`: Returns the first element in this list.
- `removeFirst`: Removes and returns the first element from this list.
- `addLast`: Appends the specified element to the end of this list.

▶ Sie dürfen *keine* Methoden mit wahlfreiem Zugriff verwenden, z.B. *nicht* `get(int index)`.

Ergänzen Sie die folgende rekursive Methode `merge`, die jeweils einen Schritt des Sortiervorgangs durchführt:

```
static void mergeSortExtern(LinkedList<Integer> b) {
    assert b != null;
    assert b.size() > 0;

    // Hilfsbaender vorbereiten:
    LinkedList<Integer> b0 = new LinkedList<>();
    LinkedList<Integer> b1 = new LinkedList<>(b.subList(0, b.size() / 2));
    LinkedList<Integer> b2 = new LinkedList<>(b.subList(b.size() / 2, b.size()));
    LinkedList<Integer> b3 = new LinkedList<>();

    merge(1, b0, b1, b2, b3);

    // Ergebnis auf Eingabeband zurueck kopieren:
    b.clear();
    b.addAll(b0);
    b.addAll(b1);
}
```

```
static void merge(int len, LinkedList<Integer> b0, LinkedList<Integer> b1,
                           LinkedList<Integer> b2, LinkedList<Integer> b3) {
    LinkedList<Integer> zielband = b0; // abwechselnd b0 und b3
    do {
        int z1 = 0, z2 = 0; // Zaehler fuer die bereits von b1/b2 kopierten ints
                            // (jeweils bis zu len)
        // fuelle zielband nach Bedarf und Moeglichkeit aus b1 *UND* b2 auf
        while (

                                                                        ) {
            if (







        }
        // fuelle zielband nach Bedarf und Moeglichkeit nur aus b1 auf
        while (





        // fuelle zielband nach Bedarf und Moeglichkeit nur aus b2 auf
        while (...) { ... } // analog zur vorangehenden Box

        // Schalte zielband um
        zielband = zielband == b0 ? b3 : b0;

        // solange weitere zu mischende Elemente vorhanden:

    } while (



    // naechster Schritt, falls notwendig
    if (                                                                    ) {




    }
}
```

Aufgabe 8 (Backtracking) (18 Punkte)

Tief im Busch stehen m Missionare ✝ und k Kannibalen ☠ vor einem Fluss und wollen diesen überqueren. Sie haben nur ein kleines Boot, das *mindestens eine und höchstens zwei* Personen befördern kann. Es gibt jedoch ein Problem: Wenn zu irgendeinem Zeitpunkt auf einer Uferseite mehr Kannibalen als Missionare sind, werden die Missionare aufgefressen.

Helfen Sie den Missionaren mittels Backtracking! Jeder Zustand wird durch ein `int[]`-Feld $\{m_L, k_L, b, m_R, k_R\}$ dargestellt: Im Anfangszustand $\{m, k, 0, 0, 0\}$ stehen alle Personen und das Boot am Linken Ufer ($b = 0$); im Zielzustand $\{0, 0, 1, m, k\}$ befinden sich alle Personen und das Boot am Rechten Ufer ($b = 1$). Bereits untersuchte Zustände werden im Feld `mem` verwaltet:

```
boolean[][][][][] mem; // Zustandsraum (true <=> wenn besucht)
```

Ein `int[]`-Feld `fahrt` $= \{\Delta m \geq 0, \Delta k \geq 0\}$ ist eine Bootsfahrt, bei dem Δm Missionare und Δk Kannibalen vom linken zum rechten Ufer übersetzen ($b = 0$). Bei einer Bootsfahrt vom rechten zum linken Ufer ($b = 1$) enthält das Feld zwei Zahlen ≤ 0. Das Feld `denkbar` enthält alle zulässigen Fahrten, die jeweils an einem der beiden Ufer `b` starten:

```
int[][][] denkbar = {
      { { 2, 0}, { 1, 0}, { 1,  1}, {0,  1}, {0,  2} }, // b = 0
      { {-2, 0}, {-1, 0}, {-1, -1}, {0, -1}, {0, -2} }  // b = 1
    };
```

Lösungsbeispiel für $(m, k) = (2, 2)$:

Zustand linkes Ufer	*Start-Ufer und Daten der Überfahrt*	*Zustand rechtes Ufer*	*Zustandsfolge in der Ergebnisliste*
✝✝ ☠☠ 🛶			[2, 2, 0, 0, 0]
	b = 0, `fahrt`={`1,1`}={✝, ☠}		
✝ ☠		🛶 ✝ ☠	[1, 1, 1, 1, 1]
	b = 1, `fahrt`={`-1,0`}={✝, -}		
✝✝ ☠ 🛶		☠	[2, 1, 0, 0, 1]
	b = 0, `fahrt`={`2,0`}={✝✝, -}		
☠		🛶 ✝✝ ☠	[0, 1, 1, 2, 1]
	b = 1, `fahrt`={`-1,0`}={✝, -}		
✝ ☠ 🛶		✝ ☠	[1, 1, 0, 1, 1]
	b = 0, `fahrt`={`1,1`}={✝, ☠}		
		🛶 ✝✝ ☠☠	[0, 0, 1, 2, 2]

Die Methode `loese` soll eine Zustandsfolge zurückgeben, die vom Anfangs- zum Zielzustand führen. Falls es keine Lösung gibt, muss `loese` `null` zurückgeben, sonst genügt eine beliebige Lösung. Ergänzen Sie die Methode `helfer` entsprechend:

```
LinkedList<int[]> loese(int m, int k) {
    mem = new boolean[m + 1][k + 1][2][m + 1][k + 1];
    return helfer(m, k, 0, 0, 0);
}
```

```
LinkedList<int[]> helfer(int mL, int kL, int b, int mR, int kR) {
    LinkedList<int[]> erg = new LinkedList<>();

    if (

        // Abbruch wegen negativer Personenzahl
        return null;

    } else if (

        // Abbruch weil Zustand schon besucht
        return null;

    } else if (

        // Abbruch zu viele Kannibalen an einem der Ufer
        return null;

    } else if (

        // Zielzustand erreicht

        erg.add(new int

        return erg;
    } else {
        // zulaessiger Zwischenzustand



        for (int[] fahrt : denkbar[b]) {
            // probiere jede denkbare fahrt, die am Ufer b beginnt



        }
    }

    return

}
```

Zusatzseite