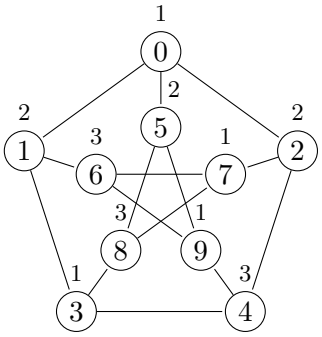
## Aufgabe 6 (Graphen einfärben) (20 Punkte)

Gegeben sei ein *ungerichteter* Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ mit Knotenmenge $\mathcal{V}$ und Kantenmenge $\mathcal{E}$ repräsentiert durch die Adjazenzmatrix `g`. Gesucht ist eine Einfärbung `cs` aller Knoten mit bis zu $m$ Farben, so dass benachbarte Knoten stets unterschiedlich gefärbt sind.
Im nebenstehenden Beispiel werden die Knoten 0 bis 9 mit den $m = 3$ numerisch codierten Farben 1, 2, 3 exemplarisch so gefärbt:

| *Knoten:* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *Farbe:* | 1 | 2 | 2 | 1 | 3 | 2 | 3 | 1 | 3 | 1 |

Das Ergebnis ist somit: `int[] cs = {1,2,2,1,3,2,3,1,3,1}`




**a)** Implementieren Sie folgende Hilfsmethode so, dass sie für den Graphen `g` und eine partielle Färbung `cs` prüft, ob der Knoten `v` mit der Farbe `c` gefärbt werden kann.

```
// check if current coloring cs allows color c for vertex v
boolean isSafe(boolean g[][], int cs[], int v, int c) {
```

```
for (int n=0; n<g.length; n++) {
    if (g[v][n] && cs[n] == c) {
        return false;
    }
}
return true;
```

```
}
```

**b)** Ergänzen Sie folgende Hilfsmethode. Sie bekommt den Graphen `g` und eine Färbung `cs` für die Knoten 0 bis $v-1$ und soll `cs` für alle anderen Knoten ab $v$ vervollständigen. Falls keine weitere Färbung möglich ist, müssen alle Knoten wieder ihre alte Farbe bekommen.

```
boolean helper(boolean g[][], int cs[], int m, int v) {
    // base case: all vertices are assigned a color 1..m => return true
```

```
if (v >= g.length) {
    return true;
}
```

```
    // try different colors for vertex v
```

```
for (int c=1; c<=m; c++) {
    if (isSafe(g, cs, v, c)) {
        cs[v] = c;
        if (helper(g, cs, m, v+1)) {
            return true;
        }
    }
}
cs[v] = 0;
return false;
```

```
}
```

**c)** Die Einfärbung startet in der Methode `color`. Falls mindestens eine Lösung existiert, dann soll `color` sie im Feld `cs` speichern und dieses zurückgeben, andernfalls muss die Methode `null` zurückgeben.

```
int[] color(boolean g[][], int m) {
    int[] cs = new int[g.length];
    if (helper(g, cs, m, 0)) {
        return cs;
    }
    return null;
}
```

## Aufgabe 7 (Vererbung und Polymorphie) (14 Punkte)

Gegeben sei folgendes Java-Programm:

```
1  interface Alpha {
2      String omega(int c);
3  }
4
5  abstract class Beta {
6      public final String omega(int i) {
7          return "Beta";
8      }
9
10     static long iota(long veryLong) {
11         return 4711;
12     }
13 }
```

```
15 class Gamma extends Beta implements Alpha {
16     public final String omega(char c) {
17         return "Gamma";
18     }
19
20     static short iota(short veryShort) {
21         return 666;
22     }
23 }
24
25 public class Polymorphie {
26     public static void main(String[] args) {
27         Alpha ag = new Gamma();
28         System.out.println(ag.omega('c'));
29         Beta bg = new Gamma();
30         System.out.println(bg.iota('c'));
31     }
32 }
```

**a)** Welches ist der *statische* Typ der in Zeile 27 deklarierten Variablen `ag`?

Alpha

**b)** Welche Ausgabe erzeugt die Anweisung in Zeile 28?

Beta

**c)** Wie ist der *dynamische* Typ der Variablen `bg` unmittelbar nach der Zuweisung in Zeile 29?

Gamma

**d)** Welche Ausgabe erzeugt die Anweisung in Zeile 30 bzw. welche Exception wird ausgelöst?

4711

**e)** Welcher Variablenart (*Instanz-/Klassen-/methodenlokale Variable*) ist `out` in Zeile 28?

Klassenvariable

**f)** Welche der vier Eigenschaften ▶ *statisch*, ▶ *abstrakt*, ▶ *überladen*, ▶ *öffentlich* trifft bzw. treffen auf die Methode `println` in Zeile 30 zu?

überladen, öffentlich

**g)** In welcher der vier Beziehungen ▶ *implementiert*, ▶ *überschreibt*, ▶ *überlädt*, ▶ *verdeckt* stehen die Methoden `omega` aus Zeile 6 und Zeile 16 zueinander?

| `omega` aus Zeile | *Beziehung* | `omega` aus Zeile |
|---|---|---|
| 16 | überlädt | 6 |

## Aufgabe 2 (Dynamische Programmierung) (20 Punkte)

Gegeben sei eine nichtleere Reihung ganzer Zahlen der Länge $n$. Gesucht ist eine Teilfolge maximaler Länge (LIS: Longest Increasing Subsequence), sodass die Teilfolge streng monoton steigend ist. Beispiel:

| Reihung: | 10 | 22 | 9 | 33 | 21 | 50 | 41 | 60 | 80 |
|---|---|---|---|---|---|---|---|---|---|
| Teil der $LIS$: | ✓ | ✓ | | ✓ | | ✓ | | ✓ | ✓ |

Sei $[a_0, a_1, \ldots a_{n-1}]$ die Reihung und sei $L_i$ die Länge derjenigen LIS, die mit $a_i$ endet.

**a)** $L_i$ kann/soll rekursiv wie folgt berechnet werden:

$$L_i = \begin{cases} 1 + \max(L_k) & \text{für } 0 \leq k < i \text{ und } a_k < a_i \\ 1 & \text{falls es kein solches } k \text{ gibt} \end{cases}$$

Die Länge der LIS der Eingabereihung ist dann das Maximum aller $L_i$ für $0 \leq i < n$. Ergänzen Sie die beiden Methoden entsprechend.

```
int naiveLIS(long[] a) { // length of LIS(a)
    int max = 1, curr;
```

```
    for (int i = 0; i < a.length; i++) {
       curr = naiveL(a, i);
       max = max >= curr ? max : curr;
```

```
    }
    return max;
}
int naiveL(long[] a, int i) {
    int max = 1, curr;
```

```
    for (int k = 0; k < i; k++) {
      if (a[k] < a[i]) {
        curr = 1 + naiveL(a,k);
        max = max >= curr ? max : curr;
      }
```

```
    }
    return max;
}
```

**b)** Verwenden Sie nun *Memoization* zur Berechnung der LIS-Länge. Dabei dürfen Sie jedes $L_i$ höchstens einmal rekursiv berechnen und bei erneutem Bedarf wiederverwenden.

```
int memLIS(long[] a) {
    int max = 1;
```

```
    int[] mem = new int[a.length];
    int curr;
```

```
    for (int i = 0; i < a.length; i++) {
```

```
      curr = memL(a, i, mem);
      max = max >= curr ? max : curr;
```

```
    }
    return max;
} // NOTE: the helper method memL must be implemented on the next page! =>
```

```
int memL(long[] a, int i, int[] mem) {
```

```
  if (mem[i] != 0) {
    return mem[i];
  }
  int max = 1, curr;
  for (int k = 0; k < i; k++) {
    if (a[k] < a[i]) {
      curr = 1 + memL(a, k, mem);
      max = max >= curr ? max : curr;
    }
  }
  mem[i] = max;
  return max;
```

```
}
```

**c)** Die rekursive Implementierung mittels *Memoization* lässt sich in eine *iterative* überführen, die die $L_i$ für aufsteigende $i$ berechnet. Ergänzen Sie die folgende Methode entsprechend:

```
int dpLIS(long[] a) {
    int max = 1;
    // initialize all li (each entry of a is its own LIS):
```

```
    int[] li = new int[a.length];
    for (int i = 0; i < li.length; i++) {
      li[i] = 1;
    }
```

```
    // fill li iteratively:
```

```
    for (int i = 0; i < a.length; i++) {
      for (int k = 0; k < i; k++) {
        if (a[k] < a[i]) {
          int curr = 1 + li[k];
          li[i] = li[i] > curr ? li[i] : curr;
        }
      }
      max = max >= li[i] ? max : li[i];
    }
```

```
    return max;
}
```

## Aufgabe 3 (Sortierverfahren) (22 Punkte)

Gegeben sei folgende Klasse:

```
class ListSorter<T extends Comparable<? super T>> {
    private List<T> data; // NOTE: assume O(1) runtime for all methods of List!
    public ListSorter(List<T> data) { //... }
    private void swap(int a, int b) { //... } swaps data elements at positions a and b
```

**a)** Die Einträge von `data` zwischen den Indizes $s+1$ und $e$ (inkl.) erfüllen die **Min**-Heap-Eigenschaft. Nur das Element am Index `s` könnte gegen die Heap-Eigenschaft verstoßen. Ergänzen Sie die **rekursive** Methode `reheap`, die die **Min**-Heap-Eigenschaft in Laufzeit $\mathcal{O}(log(e-s))$ herstellt, indem sie `data[s]` in der bei `e` endenden Halde „versickert".

```
    private void reheap(int s, int e) { // restores MIN-heap in interval [s,e]
        int left = 2 * s + 1, right = left + 1, choice;
        if (right <= e) {
            T leftChild = data.get(left);
            T rightChild = data.get(right);
            choice = leftChild.compareTo(rightChild) < 0 ?
                                                    left : right;
        } else if (left <= e) {
            choice = left;
        }
        if (choice != 0) {
            T child = data.get(choice);
            T curr = data.get(s);
            if (curr.compareTo(child) > 0) {
                swap(choice, s); reheap(choice, e);
            }
        }
    }
}
```

**b)** Implementieren Sie nun eine Haldensortierung, die `data` unter Verwendung von `reheap` **ABsteigend** sortiert und dazu die initiale Halde möglichst *effizient* herstellt:

```
    public void sortDescending() { // sorts data using MinHeapSort
        int n = data.size();
        // A: establish min-heap property (upper half always satisfies heap-property!):
        for (int s = n/2; s >= 0; s--) {
            reheap(s, n-1);
        }
        // B: repeatedly swap min towards end of list:
        for (int e = n-1; e > 0; e--) {
            swap(0, e);
            reheap(0, e-1);
        }
    }
}
```

**c)** Das Feld `data` soll nun **AUFsteigend** sortiert werden. Ergänzen Sie dazu die iterative Methode `partition`, die das Feld `data` zwischen den Indizes `s` und `e` (jeweils einschließlich) für das QuickSort-Verfahren aufbereitet.

```
private int partition(int s, int e) { // prepares interval [s,e] for QuickSort
    T p = data.get(e); // Pivot
    int a = s - 1, b = s;
    // establish partitions:
    // - elements in [s,a] are <= p
    // - elements in (a,b) are > p
```

```
    while (b < e) {
        T elem = data.get(b);
        if (elem.compareTo(p) > 0) {
            b++;
        } else {
            a++;
            swap(a,b);
            b++;
        }
```

```
    }
    // move pivot between partitions and return its final position:
```

```
    swap(a+1, p);
    return a+1;
```

```
}
```

**d)** Implementieren Sie nun das eigentliche Sortieren durch Zerlegen:

```
public void sortAscending() { // sorts data using QuickSort
```

```
    quickSortRec(0, data.size()-1);
```

```
}
```

```
protected void quickSortRec(int s, int e) {
```

```
    if (s < e) {
        int p = partition(s, e);
        quickSortRec(s, p-1);
        quickSortRec(p+1, e);
    }
```

```
}
```

## Aufgabe 6 (Rücksetzverfahren) (22 Punkte)

Gegeben sei folgende Aufzählung:

```
enum MOp { // MathOp
    ADD, SUB, MUL, DIV // represent operators +, -, *, /
}
```

Für einen Zielwert $goal > 0$ und eine Zahl $z > 0$ ist eine *minimal* lange Folge `list` von Operatoren `ADD`, `SUB`, `MUL`, `DIV` mit Endergebnis `result` = $goal$ bei folgender Auswertungsvorschrift gesucht:

```
result := z;
foreach op in list
    result := result op z;
```

Dabei sind zwei Randbedingungen zu beachten:

- Zwischenergebnisse müssen alle positiv sein und
- die Division muss stets ganzzahlig aufgehen.

Beispiel:

▶ $goal = 63, z = 3$
`list = [ADD, MUL, ADD, MUL]`, Auswertung: $\underbrace{((3+3)*3+3)*3}_{\text{Korrekt geklammerter Ausdruck in Infix-Notation}} = 63$

**a)** Ergänzen Sie folgende Hilfsmethode, die diejenige von `null` verschiedene Liste mit den wenigsten Elementen zurückgeben soll. Haben beide gleich viele Elemente (oder sind beide `null`), dann soll eine beliebige der beiden zurückgegeben werden.

```
public class SmallestExpression {
    LinkedList<MOp> better(LinkedList<MOp> a, LinkedList<MOp> b) {
```

```
if (a == null) {
  return b;
} else if (b == null) {
  return a;
} else if (a.size() < b.size()) {
  return a;
} else {
  return b;
}
```

```
    }
```

**b)** Ergänzen Sie die *rekursive* Methode `helper`, sodass `shortest` die *kürzeste* Liste zurückgibt, deren Auswertung *goal* ergibt, siehe obiges Beispiel. Gibt es keine solche Liste mit maximal $d$ Operatoren, soll `helper` den Wert `null` zurückgeben. Begrenzen Sie die Rekursionstiefe geeignet und beachten Sie die Randbedingungen. Hinweis: $r$ stellt jeweils das Ergebnis nach Auswertung der bisher in $s$ gesammelten Operatoren dar.

```
LinkedList<MOp> shortest(long goal, int z, int d) {
    return helper(goal, z, new LinkedList<MOp>(), z, d);
}

LinkedList<MOp> helper(long goal, int z, LinkedList<MOp> s, long r, int d) {
    LinkedList<MOp> best = null;
    if (r == goal)
        // found a solution => return a copy of s:
        return new LinkedList<MOp>(s);
    } else if (s.size() < d) {
        // explore each of the four operators if possible:
        s.add(MOp.ADD);
        best = helper(goal, z, s, r+z, d-1);
        s.removeLast();
        if (r-z > 0) {
            s.add(MOp.SUB);
            best = better(best, helper(goal, z, s, r-z, d-1));
            s.removeLast();
        }
        s.add(MOp.MUL);
        best = better(best, helper(goal, z, s, r*z, d-1));
        s.removeLast();
        if (r % z == 0) {
            s.add(MOp.DIV);
            best = better(best, helper(goal, z, s, r/z, d-1));
            s.removeLast();
        }
    }
    return best;
}
```

**c)** Konstruieren Sie für die gegebene Operatorenliste `list` eine Zeichenkette, die den korrekt geklammerten Ausdruck in Infix-Notation enthält, der die Auswertung für ein gegebenes $z$ beschreibt. Dabei soll die Anzahl der Klammerpaare unter Beachtung der Operatorpräzedenz („Punkt-vor-Strich“) *minimal* sein!

```
String toMathString(int z, LinkedList<MOp> list) {
    String r = Integer.toString(z);
    MOp last = null;
    for (MOp mop : list) {
        // add paranthesis if and only if necessary:
```

```
if ((last == MOp.ADD || last == MOp.SUB) &&
    (mop == MOp.MUL || mop == MOp.DIV)) {
    r = '(' + r + ')';
}
```

```
        // append next operation:
```

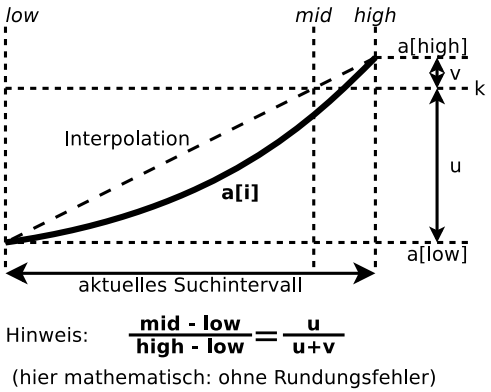
```
if (mop == MOp.ADD) {
    r = r + '+';
} else if (mop == MOp.SUB) {
    r = r + '-';
} else if (mop == MOp.MUL) {
    r = r + '*';
} else if (mop == MOp.DIV) {
    r = r + '/';
}
r = r + Integer.toString(z);
last = mop;
```

```
    }
    return r;
}
```

## Aufgabe 2 (Binäre Suche) (21 Punkte)

Bei klassischer binärer Suche wird das aufsteigend sortierte Suchintervall in jedem Schritt halbiert, was eine Laufzeit von $\mathcal{O}(\log n)$ verursacht.

**a)** Sucht man jedoch einen numerischen Schlüssel, dann kann man die Steigung der Werte im Suchabschnitt nutzen, um das Intervall asymmetrisch aufzuteilen und so die Position des Schlüssels, sofern vorhanden, durch lineare Interpolation ggf. schneller zu finden. Verwenden Sie diese Idee bei Ihrer Ergänzung der folgenden Methode, die den Index von `k` in `a` zurückgibt, oder $-1$, falls `k` nicht in `a` vorkommt. Achten Sie darauf, nicht durch 0 zu dividieren.




```
int interpolationSearch(long[] a, long k) { // iterativ
    int low = 0, mid, high = a.length - 1;
    if (high < 0)
        return -1;
    while (a[low] <= k && k <= a[high]) {
```

```
        if (a[low] == a[high]) {
            return low;
```

```
        } else {
```

```
            mid = (high - low) *
                  (k - a[low]) / (a[high] - a[low])
                  + low;
```

```
        }
```

```
        if (a[mid] > k) {
            high = mid - 1;
        } else if (a[mid] == k) {
            return mid;
        } else {
            low = mid + 1;
        }
    }
```

```
    return -1;
}
```

**b)** Bei klassischer binärer Suche wird ein Array fester Länge durchsucht. Um in einem aufsteigend sortierten Strom `s`, also in einer potentiell unendlichen Liste von Werten, mit binärer Suche nach einem Schlüssel `k` zu suchen, betrachtet man nacheinander Segmente des Stroms fester Länge `seg`.
Ergänzen Sie dazu die Hilfsmethode `segmentSearch`, die nur das erste Segment eines gegebenen Stroms mit Hilfe von `interpolationSearch` aus Aufgabenteil a) durchsucht, den Strom `s` als Nebeneffekt um dieses Segment verkürzt und als Ergebnis die Position des Schlüssels `k` innerhalb dieses Segments zurückgibt, bzw. $-1$, falls der Schlüssel nicht im Segment vorkommt.
Nehmen Sie dazu an, dass der Datentyp `Strom` die Methoden `long[] take(long n)` und `void drop(long n)` bereitstellt, die die ersten $n$ Elemente zurückgeben bzw. aus dem Strom entfernen (hat der Strom weniger als $n$ Elemente, werden alle verbliebenen zurückgegeben bzw. entfernt).

```
int segmentSearch(Strom s, long k, long seg) {
    long[] a = s.take(seg);
    s.drop(seg);
    int idx = interpolationSearch(a, k);
    return idx;
}
```

**c)** Die exponentielle Suche durchsucht den Strom aus Aufgabenteil b) nun in Segmenten, deren Längen sich schrittweise verdoppeln. Ergänzen Sie die Methode so, dass sie den Index von `k` in `s` zurückgibt, oder $-1$, falls `k` nicht in `s` vorkommt.
Der Datentyp `Strom` stellt auch die Methoden `isEmpty()` (genau dann `true`, wenn der `Strom` leer ist) bzw. `first()` (gibt das erste Element des `Stroms` zurück, ohne diesen zu verändern) bereit.

```
long exponentialSearch(Strom s, long k) {
    long dropped = 0;
    long seg = 2; // length of current segment
    while (!s.isEmpty() && s.first() <= k) {
        long relative = segmentSearch(s, k, seg);
        //found in segment:
        if (relative != -1) {
            return relative + dropped;
        }
        //not found in segment:
        dropped += seg;
        seg *= 2;
    }
    return -1;
}
```

## Aufgabe 4 (Backtracking: Binoxxo) (23 Punkte)

Binoxxo ist eine binäre Variante von Sudoku, bei der die leeren Kästchen mit `O` und `X` zu füllen sind. Dabei dürfen in einer korrekten Lösung des Rätsels höchstens zwei aufeinanderfolgende `X` oder `O` in einer Reihe oder Spalte erscheinen. Ferner müssen in jeder Reihe und jeder Spalte `O` und `X` gleich oft vorkommen. Beispiel:

*Vorgabe:*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| O | O | | | | | O | |
| | | | | X | O | | |
| | O | X | | X | | O | |
| | | O | | | | | X |
| | O | | X | | O | | |
| | | | | | O | O | |
| | | X | | | | | |
| X | O | O | X | | X | | |

*Mögliche Lösung:*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| O | O | X | X | O | X | O | X |
| X | X | O | O | X | O | X | O |
| X | O | X | O | X | O | O | X |
| O | X | O | X | O | X | O | X |
| X | O | O | X | X | O | X | O |
| O | X | X | O | X | O | O | X |
| O | X | X | O | O | X | X | O |
| X | O | O | X | O | X | X | O |

Sie dürfen im Folgenden ungeprüft annehmen, dass das Eingabe-Array quadratisch mit gerader Kantenlänge ist und noch zu füllende Felder mit einem Leerzeichen vorbelegt sind.

**a)** Ergänzen Sie die folgende Methode. Sie soll die vollständig befüllte Spalte mit Index `col` dahingehend prüfen, ob sie gleich viele `O` und `X` enthält und ob höchstens zwei `O` bzw. `X` aufeinanderfolgen:

```
boolean checkCol(char[][] in, int col) { // in[row][col]
    int os = 0, xs = 0, b = 0; // consecutive os resp. xs, balance
```

```
for (int r = 0; r < in.length; r++) {
  if (in[r][col] == 'O') {
    os += 1;
    xs = 0;
    b += 1;
  } else {
    xs += 1;
    os = 0;
    b -= 1;
  }
  if (os > 2 || xs > 2) {
    return false;
  }
}
if (b != 0) { return false; }
return true;
```

```
}
```

**HINWEIS:**
Nehmen Sie an, dass es eine analoge Methode `checkRow(char[][] in, int row)` gibt, welche die entsprechende Prüfung für die Zeile mit Index `row` durchführt.

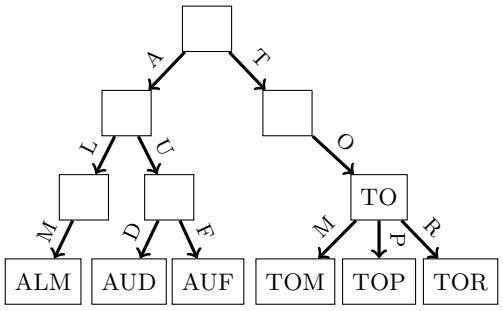
**b)** Ergänzen Sie die Hilfsmethode so, dass `solve` eine korrekte Lösung des Rätsels im Eingabe-Array ermittelt und dann `true` zurückgibt. Falls das Rätsel keine Lösung hat, dann soll das Eingabe-Array unverändert bleiben und `false` zurückgegeben werden. Um das Verfahren zu beschleunigen, brechen Sie die Rekursion ab, sobald eine von Ihrem Verfahren aktuell bearbeitete Zeile keine Lösung mehr erlaubt.

```
boolean solve(char[][] in) {
    return helper(in, 0, 0);
}
boolean helper(char[][] in, int r, int c) {
    if (r >= in.length) {
        // Basisfall
        // all completed: num. of O vs. X in each column => success or fail?
        for (int col = 0; col < in.length; col++) {
            if (!checkCol(in, col)) {
                return false;
            }
        }
        return true;
    } else if (c >= in.length) {
        // row completed => num. of O vs. X in current row => abort or cont.?
        if (!checkRow(in, r)) {
            return false;
        }
        return helper(in, r+1, 0);
    } else if (in[r][c] != ' ') {
        // skip pre-defined field
        return helper(in, r, c+1);
    } else {
        // try an O
        in[r][c] = 'O';
        if (helper(in, r, c+1)) {
            return true;
        }
        in[r][c] = 'X';
        if (helper(in, r, c+1)) {
            return true;
        }
        in[r][c] = ' ';
    }
    return false;
}
```

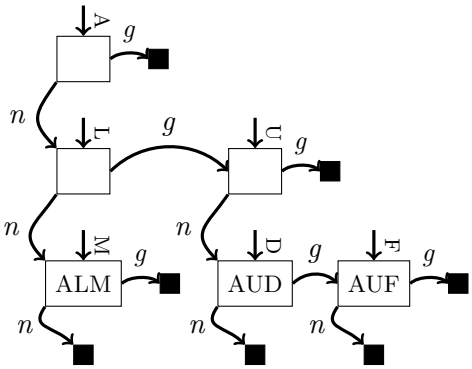
## Aufgabe 5 (ADT: Trie) (22 Punkte)

Ein *Trie* ist eine Variante eines Suchbaums. Jeder Kante wird hier implizit ein Zeichen zugeordnet. Ein Pfad durch den Baum stellt daher eine Zeichenkette dar. Um einen String einzufügen, wird sowohl dieser String als auch der *Trie* (vom Wurzelknoten aus) durchlaufen. In jedem Schritt wird die Ausgangskante des aktuellen Knotens verfolgt, die dem nächsten Zeichen im String entspricht. Falls es eine solche Kante noch nicht gibt, wird sie samt Zielknoten neu angelegt. Sind alle Zeichen abgearbeitet, wird die Zeichenkette im erreichten Knoten vermerkt. Mehrfaches Einfügen des gleichen Strings wird ignoriert. Beachten Sie, dass Zeichenketten nicht nur in Blättern des *Tries* vermerkt sind.

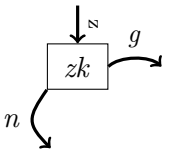
*Beispiel: Trie* nach Einfügen der Wörter TOR, AUD, ALM, TOM, AUF, TO und TOP:




ADT-Darstellung des linken Teilbaums: ($g$ = Geschwister, $n$ = Nachfolger)




Der Primärkonstruktor $Knoten(z, zk, g, n)$ erzeugt einen $Trie$-Knoten folgender Struktur:




$z$: ein einzelnes Zeichen
$zk$: Zeichenkette (ggf. *Leer*)
$g$: erstes Geschwister
$n$: erster Nachfolger

Gegeben seien die folgenden abstrakten Datentypen:

---

**adt** *Liste // generische Liste*
**sorts** *Liste, T // T ist der Datentyp der Elemente in der Liste<T>*
**ops**

| | | | |
|---|---|---|---|
| *Leer:* | | $\mapsto$ *Liste* | *// Primärkons.: erzeugt leere Liste* |
| *Kette:* | *T* × *Liste* | $\mapsto$ *Liste* | *// Primärkons.: Kette(k,r) → Liste mit Kopf k und Rest r* |
| *verketten:* | *Liste* × *Liste* | $\mapsto$ *Liste* | *// siehe Teilaufgabe a)* |

**axs**
...
**end** *Liste*

---

**adt** *Trie*
**sorts** *Trie, Zeichen, Liste // betrachten Sie Zeichen als weiteren ADT ohne* ***ops***
**ops**

| | | | |
|---|---|---|---|
| *Neu:* | | $\mapsto$ *Trie* | *// leerer Trie (■)* |
| *Knoten:* | *Zeichen* × *Liste<Zeichen>* × *Trie* × *Trie* | $\mapsto$ *Trie* | *// siehe Skizze* |
| *alle:* | *Trie* | $\mapsto$ *Liste<Liste<Zeichen>>* | *// siehe Teilaufgabe b)* |
| *einfuegen:* | *Trie* × *Liste<Zeichen>* | $\mapsto$ *Trie* | *// siehe Teilaufgabe c)* |
| *helfer:* | *Trie* × *Liste<Zeichen>* × *Liste<Zeichen>* | $\mapsto$ *Trie* | *// Helfer für einfuegen* |
| *loeschen:* | *Trie* × *Liste<Zeichen>* | $\mapsto$ *Trie* | *// siehe Teilaufgabe d)* |

**axs**
...
**end** *Trie*

---

**a)** Ergänzen Sie die Axiome der Operation *verketten*, die zwei Listen unter Bewahrung der Elementreihenfolge konkateniert:

$verketten(Leer, l2) =$ l2

$verketten(Kette(t, l1), l2) =$ Kette(t, verketten(l1, l2))

**b)** Ergänzen Sie die Axiome der Operation *alle* so, dass *alle* sämtliche in den Knoten des *Tries* gespeicherten Werte als *Liste* zurückgibt:

$alle(Neu) =$ Leer

$alle(Knoten(z, Leer, g, n)) =$ verketten(alle(g), alle(n)) // $zk = Leer$

$alle(Knoten(z, zk, g, n)) =$ Kette(zk, verketten(alle(g), alle(n)) // $zk \neq Leer$

**c)** Ergänzen Sie die Axiome der Operation *helfer* so, dass *einfuegen* eine neue Zeichenkette (Liste von Zeichen *zkNeu*) korrekt in den *Trie* einfügt:

$einfuegen(t, zkNeu) = helfer(t, zkNeu, zkNeu)$

$helfer(Neu, Kette(z2, Leer), zkNeu) =$ Knoten(z2, zkNeu, Neu, Neu)

$helfer(Neu, Kette(z2, zk2), zkNeu) =$

$=$ Knoten(z2, Leer, Neu, helfer(Neu, zk2, zkNeu)) // $zk2 \neq Leer$

$helfer(Knoten(z1, zk1, g, n), Kette(z1, Leer), zkNeu) =$

$=$ Knoten(z1, zkNeu, g, n)

$helfer(Knoten(z1, zk1, g, n), Kette(z1, zk2), zkNeu) =$

$=$ Knoten(z1, zk1, g, helfer(n, zk2, zkNeu)) // $zk2 \neq Leer$

$helfer(Knoten(z1, zk1, g, n), Kette(z2, zk2), zkNeu) =$

$=$ Knoten(z1, zk1, helfer(g, Kette(z2, zk2, zkNeu), n) // $z1 \neq z2$

**d)** Ergänzen Sie die Axiome der Operation *loesche*, die den String $zk2$ aus dem *Trie* löscht und dabei den Baum verkleinert, wenn der Zielknoten ein Blatt ist:

$loeschen(Neu, zk2) = Neu$

$loeschen(Knoten(z1, zk1, Neu, Neu), Kette(z1, Leer)) =$ Neu

$loeschen(Knoten(z1, zk1, g, n), Kette(z1, Leer)) =$

$=$ Knoten(z1, Leer, g, n)

$loeschen(Knoten(z1, zk1, g, n), Kette(z1, zk2)) =$

$=$ Knoten(z1, zk1, g, loeschen(n, zk2)) // $zk2 \neq Leer$

$loeschen(Knoten(z1, zk1, g, n), Kette(z2, zk2)) =$

$=$ Knoten(z1, zk1, loeschen(g, Kette(z2, zk2)), n) // $z1 \neq z2$

## Aufgabe 6 (Graphen) (18 Punkte)

Für einen gerichteten zusammenhängenden Graphen $g$ (`g[x][y] == true` $\Leftrightarrow$ es gibt eine Kante $x \to y$) soll die Anzahl der Zyklen der Länge $n > 0$ bestimmt werden. Die letzte der $n$ Kanten eines Zyklus endet am ersten der $n$ Knoten (Startknoten) des Zyklus. Außer dem Startknoten kommt kein weiterer Knoten mehrfach in einem Zyklus vor.

**a)** Implementieren Sie in der folgenden Hilfsmethode eine rekursive Tiefensuche, die für einen Knoten `start` alle von diesem ausgehenden einfachen Pfade der Länge $n-1$ durchsucht. Wenn am Ende eines solchen einfachen Pfades eine Verlängerung um eine weitere Kante wieder `start` erreicht, zählen Sie diesen Zyklus in `count`. Markieren Sie den Besuch eines Knotens im Feld `m`, um sicherzustellen, dass jeder Knoten (außer dem Startknoten) nur höchstens einmal im Zyklus vorkommt.

```
long count;

void dfs(boolean[][] g, boolean[] m, int n, int start, int curr) {
    m[curr] = true; // mark
    if (n == 1) {
        // end of path reached
        if (g[curr][start]) {
            count++;
        }
    } else {
        // somewhere along the path
        for (int next = 0; next < g[curr].length; next++) {
            if (g[curr][next] && !m[next]) {
                dfs(g, m, n-1, start, next);
            }
        }
    }
    m[curr] = false;
}
```

**b)** Ergänzen Sie nun die Methode `countCycles`, die unter Verwendung von `dfs` alle in $g$ enthaltenen Zyklen der Länge $n$ zählt und die Gesamtanzahl als Ergebnis liefert.
Hinweis: Ein Zyklus mit $n$ Knoten ist dabei mehrfach zu zählen, denn jeder der $n$ Knoten kann als Startknoten betrachtet werden.

```
long countCycles(boolean[][] g, int n) {
    count = 0;
    boolean[] m = new boolean[g.length];
    for (int s=0; s<g.length; s++) {
        dfs(g, m, n, s, s);
    }
    return count;
}
```

## Aufgabe 2 (Binäre Suche) (20 Punkte)

Gegeben ein aufsteigend sortiertes Array `a` der Länge $n$, in dem Werte auch mehrfach vorkommen können. Ziel ist es, die Anzahl der Vorkommen eines beliebigen Wertes `k` in `a` mit einer logarithmischen Laufzeit von $\mathcal{O}(\log n)$ zu bestimmen.

**a)** Ergänzen Sie die *rekursive* Methode `first`: Sie verwendet *Binäre Suche*, um die Position des *ersten* Vorkommens von `k` in `a` innerhalb des geschlossenen Intervalls `[low,high]` zu bestimmen und gibt `-1` zurück, falls `k` nicht in `a` vorkommt.

```
int first(long[] a, long k, int low, int high) {
    int mid;
```

```
if (high < low) {
    return -1;
}
mid = (high + low) / 2;
if (a[mid] > k || (a[mid] == k && mid-1 >= 0
                    && a[mid-1] == k)) {
    return first(a, k, low, mid-1);
} else if (a[mid] == k) {
    return k;
} else {
    return first(a, k, mid+1, high);
}
```

```
}
```

**b)** Ergänzen Sie die *rekursive* Methode `last`: Sie verwendet *Binäre Suche*, um die Position des *letzten* Vorkommens von `k` in `a` innerhalb des geschlossenen Intervalls `[low,high]` zu bestimmen und gibt `-1` zurück, falls `k` nicht in `a` vorkommt.

```
int last(long[] a, long k, int low, int high) {
    int mid, n = a.length;
```

```
if (high < low) {
    return -1;
}
mid = (high + low) / 2;
if (a[mid] < k || (a[mid] == k &&
                    mid+1 < a.length && a[mid+1]==k)){
    return first(a, k, mid+1, high);
} else if (a[mid] == k) {
    return k;
} else {
    return first(a, k, low, mid-1);
}
```

```
}
```

**c)** Ergänzen Sie die Methode `count`: Sie verwendet die bisherigen Hilfsmethoden, um die Anzahl der Vorkommen von `k` in `a` zu bestimmen.

```
int count(long[] a, long k) {
    int n = a.length;
```

```
int first = first(a, k, 0, n-1);
if (first == -1) {
  return 0;
}
int last = last(a, k, 0, n-1);
return last - first + 1;
```

```
}
```

## Aufgabe 3 (Dynamische Programmierung: Maximaler Profit) (20 Punkte)

Beim sog. Daytrading handelt man Aktien kurzfristig, indem man sie innerhalb eines Tages kauft und später wieder verkauft, eventuell mehrfach, um aus den Kursschwankungen Profit zu machen. In dieser Aufgabe sollen Sie mit dynamischer Programmierung in Laufzeit $\mathcal{O}(n)$ ermitteln, wie viel Profit mit einer einzigen Aktie an einem Tag hätte maximal gemacht werden können, wenn die Kursentwicklung vorab bekannt gewesen wäre. Die Randbedingung ist, dass die Aktie maximal zweimal gekauft und wieder verkauft werden darf.
Beispiel:
- Eingabe Kurse: `ks` $= [\underset{K}{10}, \underset{V}{22}, \underset{K}{5}, 75, 65, \underset{V}{80}]$ $\quad n =$ `ks.length`, `ks[i]` $\geq 0$
- Lösung:
- Ausgabe Maximaler Profit: 87 = (22-10) + (80-5)

**a)** Ergänzen Sie die iterative Methode `maxProfitOneNaiv`, die alle Kombinationen aus *einem* Verkaufskurs und *einem* früheren Einkaufskurs betrachtet und den maximalen Profit innerhalb des geschlossenen Intervalls $[a, z]$ in `ks` ohne weitere Optimierungen in $\Theta(n^2)$ Laufzeit bestimmt. Falls kein Profit möglich ist (weil der maximale Profit negativ oder das Intervall leer bzw. einelementig ist), geben Sie 0 zurück.

```
int maxProfitOneNaiv(int[] ks, int a, int z) { // Suchintervall [a, z] in ks
    int maxProf = 0; // falls kein Profit moeglich ist
```

```
for (int k = a; k <= z; k++) {
    for (int v = k+1; v <= z; v++) {
        maxProf = Math.max(maxProf,
                           ks[v] - ks[k]);
    }
}
```

```
    return maxProf;
}
```

**b)** Man kann die Laufzeit der Profitsuche auf $\mathcal{O}(n)$ reduzieren, indem man jeweils nur die Differenz zwischen einem untersuchten Kurs und dem kleinsten früher gesehenen Kurs bestimmt. Ergänzen Sie die iterative Methode `maxProfitOne`, die den maximalen Profit (kann auch 0 sein) innerhalb des geschlossenen Intervalls $[a, z]$ in `ks` (kann auch leer oder einelementig sein) in *linearer* Zeit bestimmt:

```
int maxProfitOne(int[] ks, int a, int z) { // Suchintervall [a, z] in ks
    int maxProf = 0, minPrice = Integer.MAX_VALUE;
```

```
for (int v = a+1; v <= z; v++) {
    minPrice = Math.min(minPrice, ks[v-1]);
    maxProf = Math.max(maxProf, ks[v] - minPrice);
}
```

```
    return maxProf;
}
```

**c)** Ergänzen Sie die folgende iterative Methode `maxProfitTwoNaiv`, die den maximalen Profit bei bis zu zwei nicht-überlappende Transaktionen (kaufen und später verkaufen) mit den Kursen aus `ks` mittels `maxProfitOne` in $\mathcal{O}(n^2)$ Laufzeit bestimmt:

```
int maxProfitTwoNaiv(int[] ks) {
    int maxProf = 0;
    for (int z = 0; z < ks.length; z++) {
        maxProf = Math.max(maxProf,
                           maxProfitOne(ks, 0, z) +
                           maxProfitOne(ks, z+1,
                                        ks.length-1));
    }
    return maxProf;
}
```

**d)** Mit dynamischer Programmierung soll der quadratische Aufwand von `maxProfitTwoNaiv` auf lineare Laufzeit $\mathcal{O}(n)$ reduziert werden. Dazu traversieren Sie `ks` je einmal aus jeder Richtung und befüllen das Feld `profit` wie folgt:

- zuerst von hinten nach vorne, um den maximalen Profit mit höchstens einer Transaktion im Intervall $[i, n-1]$ zu bestimmen und in `profit[i]` abzuspeichern;
- danach von vorne nach hinten, um den maximalen Profit mit bis zu zwei Transaktionen zu bestimmen und in `profit[i]` abzuspeichern (bei zwei Transaktionen findet die frühere im Intervall $[0, i]$ statt).

Ergänzen Sie die iterative Methode `maxProfitOpt` entsprechend.
**Achtung:** Es ist nicht erforderlich, hierzu `maxProfitOne` zu verwenden.

```
int maxProfitTwoOpt(int[] ks) {
    int n = ks.length, minPrice, maxPrice;
    if (n < 1) {
        return 0;
    }
    int[] profit = new int[n];
    // traverse right to left:
    maxPrice = 0;
    for (int i = n-2; i >= 0; i--) {
      maxPrice = Math.max(maxPrice, ks[i+1]);
      profit[i] = Math.max(profit[i+1],
                           maxPrice - ks[i]);
    }
    // traverse left to right:
    minPrice = Integer.MAX_VALUE;
    for (int i = 1; i < n-1; i++) {
       minPrice = Math.min(minPrice, ks[i]);
       profit[i] = Math.max(profit[i-1],
                            ks[i] - minPrice + profit[i+1]);
    }
    return profit[n-1];
}
```

## Aufgabe 6 (Breitensuche) (21 Punkte)

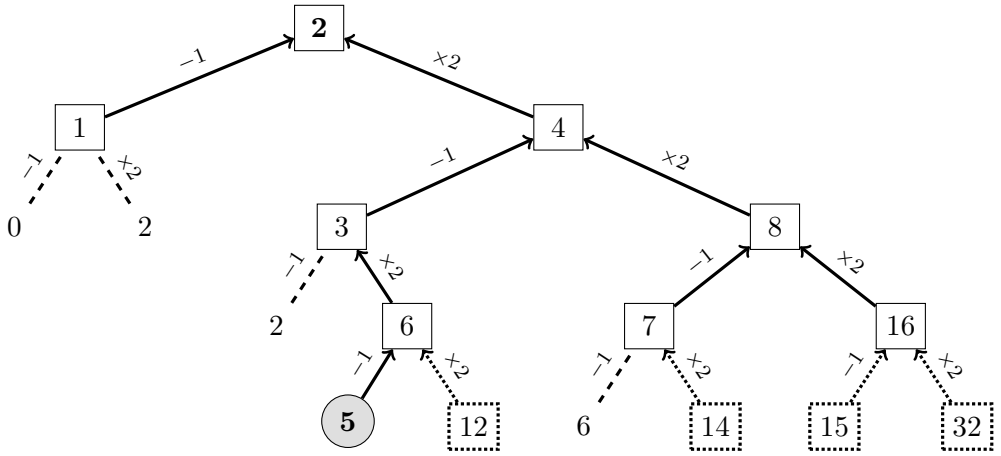
Gegeben seien folgende Aufzählung bzw. Klasse:

```
enum MOp { // MathOp
    SUB, MUL // represent operators -, *
}

class Step {
    final Step previous; // null if first
    final MOp op; // last MathOp applied (null if first)
    final long result; // current result after op

    Step(Step previous, MOp op, long result) {
        this.previous = previous;
        this.op = op;
        this.result = result;
    }
}
```

Ausgehend von einem Startwert $start \geq 1$ soll ein Zielwert $goal \geq 1$ mit einer *minimalen* Anzahl Operationen $SUB$ und $MUL$ mit gegebenem $subtrahend \geq 1$ bzw. $factor \geq 2$ erreicht werden. Die Operationen werden ohne Berücksichtigung der Operatorpräzedenz jeweils sofort auf Zwischenergebnisse angewandt.
Finden Sie die *minimale* Anzahl Schritte mittels *Breitensuche*, die konzeptionell einen „Baum" aufspannt (siehe Beispiel). Wird ein Zwischenergebnis erreicht, das $\leq 0$ ist oder bereits zuvor gefunden wurde, dann endet die Breitensuche an diesem Zwischenknoten (gestrichelte Linie im Beispiel wird nicht mehr verfolgt). Sie endet auch, wenn ein Knotenwert das 1000-fache des Zielwerts *goal* übersteigt. Sobald erstmals der Zielwert *goal* erreicht wird, endet die Suche. Je nach Ihrer Implementierung werden die weiteren Knoten (gepunktet im Beispiel) der Ebene mit der Lösung ggf. nicht mehr erzeugt.
▶ Beispiel: Von $start = \mathbf{2}$ nach $goal = \mathbf{5}$ mittels $subtrahend = 1$ und $factor = 2$:




Jeder Knoten im Lösungsbaum wird durch ein Objekt der Klasse `Step` repräsentiert. Folgt man der Verkettung entlang `previous` vom Lösungsblatt zur Wurzel, erhält man die gewünschte minimale Folge an Operationen in umgekehrter Reihenfolge.

$$-1, \times 2, -1, \times 2 \rightsquigarrow \Big(\Big(\big((\mathbf{2} \times 2) - 1\big) \times 2\Big) - 1\Big) = \mathbf{5} \rightsquigarrow [\texttt{MUL}, \texttt{SUB}, \texttt{MUL}, \texttt{SUB}]$$

Ihre Lösung soll eine minimale Folge an Operationen in der anzuwendenden Reihenfolge liefern bzw. `null`, falls keine Lösung unter den gegebenen Einschränkungen existiert.

Ergänzen Sie die folgende *iterative* Methode entsprechend:

```
List<MOp> findShortest(long start, long goal, int subtrahend, int factor) {
    Set<Long> visited = new HashSet<>();
    Queue<Step> queue = new LinkedList<>();
    // prepare data structures:
```

```
visited.add(start);
queue.add(new Step(null, null, start));
```

```
    // perform BFS until final step found or no solution possible:
    boolean found = false;
    Step s = null; // last step
    long sub, mul; // intermediate result after next corresponding step
```

```
while (!queue.isEmpty()) {
    s = queue.removeFirst();
    if (s.result == goal) {
        found = true;
        break;
    }
    sub = s.result - subtrahend;
    if (sub > 0 && !visited.contains(sub)) {
        queue.add(new Step(s, MOp.SUB, sub));
        visited.add(sub);
    }
    mul = s.result * factor;
    if (mul <= 1000 * goal && !visited.contains(mul)) {
        queue.add(new Step(s, MOp.MUL, mul));
        visited.add(mul);
    }
```

```
    } // end while
    // generate solution if it exists:
    List<MOp> result = null;
```

```
if (found) {
    result = new LinkedList<>();
    while (s.op != null) {
        result.addFirst(s.op);
        s = s.previous;
    }
```

```
    } // end if
    return result;
}
```