Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 11: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Zusicherungen und Verifikation (Präsenzaufgabe)

a) Geben Sie vor dem Feldzugriff eine Assertion an, welche garantiert, dass der Feldzugriff sicher ausgeführt werden kann. Bestimmen Sie dann eine möglichst präzise Assertion für den Anfang der Prozedur.

Bei einer solchen Aufgabenstellungen ist die geforderte Zusicherung selbst aufzustellen und an der passenden Programmstelle einzutragen. Von dieser ausgehend können wir dann bestimmen, was im Vorfeld gelten muss, um diese Zusicherung halten zu können. Wir arbeiten uns also von der Zusicherung im Programm rückwärts zum Anfang der Prozedur durch.

In dieser Aufgabe geht es noch nicht darum, ausschließlich die Regeln zu verwenden, um einen wirklich wasserdichten Beweis zu erstellen, sondern sinnvoll zu argumentieren. Für **new**-Konstrukte und Feldzugriffe, sowie Methodenaufrufe, fehlen uns auch einfach die Regeln.

Trotzdem kann hier schon gut die Zuweisungsregel geübt werden und man sieht wie man mit Assertions umgeht.

```
int helper(int index, int first, int last) {
    int i;
    int[] feld;

  // (5) einfache Umformungen
  assert first <= index && index <= last;

  // (4) quasi Zuweisungsregel (fuer feld.length) und feld kann hier noch null sein
  assert 0 <= index - first && index - first < last - first + 1;

    feld = new int[last - first + 1];

  // (3) Prozedur kann keine Auswirkung auf diese Bedingung haben
  assert feld != null && 0 <= index - first && index - first < feld.length;

    initialize(feld);

  // (2) Zuweisungsregel
  assert feld != null && 0 <= index - first && index - first < feld.length;

    i = index - first;

  // (1) das Feld darf nicht null sein und der Index muss passen
  assert feld != null && 0 <= i && i < feld.length;

    return feld[i];
}
```

b) Zeigen Sie mit einem Annotationsbeweis, dass die folgende Prozedur ihre Spezifikation erfüllt:

Bei einem Annotationsbeweis ist dies etwas anders. Hier haben wir eine Zusicherung für das Endergebnis (*ensures*) gegeben und wollen beweisen, dass dieses am Ende der Prozedur tatsächlich gilt. Eine gute Möglichkeit dies zu tun ist mit Hilfe von Regeln eine Vorbedingung zu generieren, die uns die Bedingungen an die Parameter der Funktion zeigt. In diesem Fall gehen wir also "von unten" vor.

Hinweis: Einem Annotationsbeweis an sich sieht man natürlich die Reihenfolge in der die Annotationen getätigt wurden nicht mehr an und dies zu wissen ist auch für die Korrektheit des Beweises nicht notwendig. Hierbei geht es lediglich um das Vorgehen einen solchen zu erstellen.

Damit es wirklich ein Beweis ist, dürfen wir nur die aus der Vorlesung bekannten Regeln – und zwar eine nach der anderen – anwenden. Verschiedene Regelanwendungen dürfen nicht vermischt werden! Außerdem ist es essentiell, dass Umformungen in mehreren – jeweils *einfach nachvollziehbaren* – Schritten durchgeführt werden.

Für diese Aufgabe ist es ausreichend die *Zuweisungsregel* zu benutzen und zum Schluss einmal eine (echte) *Abschwächung* vorzunehmen. In den Kommentaren steht jeweils wann und wie eine Assertion entstanden ist.

```
/*@ requires N > 0
  @ modifies \nothing
  @ ensures \result == N*N*N + 4*N*N + 4*N
*/
double exec(double N) {
    double s, f1, f2, f3, result;

  // Damit ist die spezifizierte Vorbedingung (requires) erfuellt.
  // (10) echte Abschwaechung, da "N > 0 => true" gilt und "N > 0 <=/=> true"
  assert N > 0;

  // (7-9) Mathematische Umformungen (praezise: 3x Abschwaechungsregel)
  assert true;
  assert 2*N*N*N + 8*N*N + 8*N == 2*N*N*N + 8*N*N + 8*N;
  assert (2*N*N + 4*N) * (N + 2) == 2 * (N*N*N + 4*N*N + 4*N);

  // (6) Zuweisungsregel
  assert (N * (2*N + 4) * (N + 2)) / 2 == N*N*N + 4*N*N + 4*N;

    f1 = 2*N + 4;

  // (5) Zuweisungsregel
  assert (N * f1 * (N + 2)) / 2 == N*N*N + 4*N*N + 4*N;

    f2 = N + 2;

  // (4) Zuweisungsregel
  assert (N * f1 * f2) / 2 == N*N*N + 4*N*N + 4*N;

    f3 = N * f1 * f2;

  // (3) Zuweisungsregel
  assert f3 / 2 == N*N*N + 4*N*N + 4*N;

    s = f3 / 2;

  // (2) Zuweisungsregel
  assert s == N*N*N + 4*N*N + 4*N;

    result = s;

  // (1) spezifizierte Nachbedingung (ensures)
  assert result == N*N*N + 4*N*N + 4*N;

    return result;
}
```

Aufgabe 2 Zusicherungen und Verifikation (Einreichaufgabe)

a) Fügen Sie in die Implementierung die nötigen Zusicherungen ein, um plausibel zu machen, dass auf `sorted` nur zugegriffen wird, wenn es nicht `null` ist und der Index innerhalb der gültigen Schranken ist.

```
boolean binary(int[] sorted, int first, int last, int key) {
  // (6) assert true;

    if (sorted == null || 0 > first || last >= sorted.length) return false;

  // (3) Direkte Folge aus der Bedingung des if.
  assert sorted != null && 0 <= first && last < sorted.length;

    if (first < last) {
      // (4) Direkte Folge aus (3) und noch einem if.
      assert sorted != null && 0 <= first && last < sorted.length && first < last;

      // (5) Einzig verbleibende Beweisverpflichtung: (4) impliziert (2)
      // (2) Zuweisungsregel (zu beweisen! Erfolg beweist (1) aber automatisch!)
      assert sorted != null && 0 <= first + (last - first) / 2
                           &&      first + (last - first) / 2 < sorted.length;
        int mid = first + (last - first) / 2;

      // (1) Damit genannte Fehler nicht passieren (zu beweisen!).
      assert sorted != null && 0 <= mid && mid < sorted.length;

        int x = sorted[mid];
        if (key < x) { return binary(sorted, first, mid, key); }
        if (key > x) { return binary(sorted, mid + 1, last, key); }
        return true;
    }
    return false;
}
```

1. Wir fangen an mit der einzigen Zusicherung die wir brauchen, um alle angegebenen Exceptions zu vermeiden. Können wir (1) beweisen sind wir fertig.
2. Mit der Zuweisungsregel können wir diese Beweisverpflichtung in eine andere umwandeln. Können wir (2) beweisen ist (1) automatisch bewiesen.
3. Da im ersten **`if`** aus der Prozedur gesprungen wird, können wir die Negation der Bedingung annehmen. (3) ist also bereits gültig und braucht nicht mehr bewiesen zu werden.
4. Für den then Zweig des **`if`** können wir (zumindest am Anfang) dessen Bedingung direkt annehmen. Da sich sonst nichts geändert hat gilt in (4) sowohl (3), also auch die zusätzliche Bedingung.
5. Damit haben wir zwei Zusicherungen (4) und (2) aneinander stehen, wovon die eine (4) schon gilt, die andere (2) zu zeigen ist. Können wir beweisen, dass (4) Bedingung (2) impliziert sind wir fertig.

   Die erste Bedingung von (2), dass `sorted != null` ist, ist trivial, da sie in (4) direkt so steht.

   Aus `first < last` folgt direkt, dass `last - first > 0` ist und natürlich auch `(last - first) / 2 > 0`. Nehmen wir noch `0 <= first` dazu, ist also `0 <= first + (last - first) / 2` bewiesen.

   Die letzte Bedingung von (2) lässt sich auch einfach zeigen, da wir wissen, dass `first < last`. Denn addieren wir auf `first` die Hälfte vom Abstand von `first` zu `last`, während `last` größer als `first` ist, landen wir maximal auf `last`, also `first + (last - first) / 2 <= last`. Da ebenfalls gilt `last < sorted.length` erhalten wir insgesamt `first + (last - first) / 2 <= sorted.length`.
6. Um die rekursiven Aufrufe vernachlässigen zu können, müssen wir uns nur bewusst machen, dass wir nichts als Vorbedingung voraussetzen.

Was die Frage der Korrektheit angeht: Die kann immer nur im Zusammenhang mit einer *Spezifikation* beantwortet werden und ist an sich damit sinn- und bedeutungslos! Ja, die beiden genannten Fehler werden dann nicht mehr auftreten. *Aber*: Nein, die Funktion muss nicht unbedingt die binäre Suche korrekt implementieren.

b) Betrachten Sie folgende Implementierung der Funktion n^3. Führen Sie einen Annotationsbeweis und zeigen Sie damit, dass die Funktion ihre Spezifikation erfüllt.

```
// Damit ist die spezifizierte Vorbedingung (requires) erfuellt.
// (16+17) Abschwaechungsregel
assert N > 0;
assert true && N >= 1;

// (15) Zuweisungsregel
assert 1 == 1*1*1 && 1 <= N;

  i = 1;

// (14) Zuweisungsregel
assert 1 == i*i*i && i <= N;

  s = 1;

// (5) Schleifenregel: INV
assert s == i*i*i && i <= N;

  while (i < N) {
    // (5) Schleifenregel: B && INV
    assert i < N && s == i*i*i; // && i <= N, wird aber von i < N impliziert

    // (10-13) Abschwaechungsregeln (um (5) eins drueber zu beweisen)
    assert s == i*i*i && i < N;
    assert s + 3*i*i + 3*i + 1 == i*i*i + 3*i*i + 3*i + 1 && i < N;
    assert s + 3*i*i + 3*i + 1 == i*i*i + 2*i*i + i + i*i + 2*i + 1 && i < N;
    assert s + 3 * (i*i + i) + 1 == (i*i + 2*i + 1)*(i+1) && i + 1 <= N;

    // (9) Zuweisungsregel
    assert s + 3 * (i * (i+1)) + 1 == (i+1)*(i+1)*(i+1) && (i+1) <= N;

        t = i;

    // (8) Zuweisungsregel
    assert s + 3 * (t * (i+1)) + 1 == (i+1)*(i+1)*(i+1) && (i+1) <= N;

        i = i + 1;

    // (7) Zuweisungsregel
    assert s + 3 * (t * i) + 1 == i*i*i && i <= N;

        t = t * i;

    // (6) Zuweisungsregel
    assert s + 3 * t + 1 == i*i*i && i <= N;

        s = s + 3 * t + 1;

    // (5) Schleifenregel: INV
    assert s == i*i*i && i <= N;
  }

// (5) Schleifenregel: not B && INV
assert i >= N && s == i*i*i && i <= N;

// (3+4) Abschwaechungsregel (um (5) eins drueber zu beweisen)
assert i >= N && s == i*i*i && i <= N;
assert i == N && s == i*i*i;

// (2) Zuweisungsregel
assert s == N*N*N;

  result = s;

// (1) spezifizierte Nachbedingung (ensures)
assert result == N*N*N;
```

Bei dieser Aufgabe ist die einzig neue Schwierigkeit die Schleife. Um diese korrekt zu annotieren, benötigen wir die *Schleifenregel*. Die Idee dieser Regel ist einen Zusammenhang zwischen den in der Schleife veränderten Werten zu finden, der vor und über die Ausführung der Schleife unverändert bleibt. Die *Invariante* muss einerseits korrekt für die Schleife sein, andererseits aber auch stark genug, um die Bedingung, die nach der Schleife gelten soll, zu implizieren.

Es gibt im Allgemeinen keine Möglichkeit, die "richtige" – oder überhaupt eine passende – Invariante zu bestimmen, jedoch hilft es sich folgende Dinge anzuschauen:

- Die Anweisungen in der Schleife, da diese letztendlich bestimmen was die Schleife berechnet und den Zusammenhang zwischen in ihr veränderten Variablen enthalten.
- Die geforderte Bedingung nach der Schleife, die "von unten" abgeleitet wurde. Diese muss schließlich von der Invariante zusammen mit der negierten Schleifenbedingung impliziert werden. Sie eignet sich meist nicht direkt als Invariante, gibt aber einen Hinweis auf deren Form.

Um die Schleifenregel dann für eine Invariante anzuwenden, schreibt man an alle mit (5) markierten Stellen die entsprechenden Formeln. Ob diese Invariante bewiesen werden kann, zeigt sich an der Stelle (13) zu (5), wo mit Hilfe der Abschwächungsregel als letzter Schritt gezeigt wurde, dass die Invariante vom Rumpf der Schleife erhalten wird. Ob die Invariante ausreichend stark ist, zeigt sich am Übergang von (4) zu (5), da auch hier höchstens abgeschwächt werden kann um mit den Zusicherungen im Rest der Prozedur zu verknüpfen.

Aufgabe 3 Listen und Komplexität (Einreichaufgabe)

Hinweis: Diese Aufgabe bezieht sich auf die Listen von Aufgabe 2 aus Übungsblatt 9.

a) Betrachten Sie eine Sequenz von n Aufrufen der Prozedur `append` mit vorheriger Erzeugung (`create`) der leeren Liste. Leiten Sie die Aufwandsfunktion $A(n)$ für diese Sequenz von Aufrufen her und bestimmen Sie dann deren Komplexitätsklasse.

Die Kosten von `append` werden in Abhängigkeit von der Länge m der Liste, an die angehängt wird, angegeben, die Kosten von `create` sind konstant.

$$
\begin{aligned}
A(n) &= A_{create} + \sum_{i=0}^{n-1} A_{append}(i) \\
A_{create} &= 6 \\
A_{append}(m) &= 16 + m + \sum_{i=0}^{m-1} 9 = 10m + 16 \\
\rightsquigarrow A(n) &= 6 + \sum_{i=0}^{n-1} (10i + 16) \\
&= 6 + 16n + 10 \sum_{i=0}^{n-1} i = 6 + 16n + 10\frac{(n-1)n}{2} = 5n^2 + 11n + 6 \\
&\leq 16n^2 + 6 \\
\rightsquigarrow A &\in O(n^2) \quad \text{mit } c = 16 \text{ und } d = 6 \quad \text{(vgl. Folie 762)}
\end{aligned}
$$

b) Implementieren Sie diese Optimierung und passen Sie die restlichen Prozeduren entsprechend an. Was sind die Vor- und Nachteile dieser Implementierung und in welchen Fällen sind diese besonders relevant?

```
static class List {
  double[] contents;
  int size;
}

static List create() {
  // [10] +3 Zuweisung, +2 Zugriff, +3 new (Groesse 2), +2 new (Laenge 1)
  List l = new List();
  l.contents = new double[1];
  l.size = 0;

  return l;
}

static void append(List l, double elem) {
  // [4] +1 Vergleich, +3 Zugriff
  if (l.size == l.contents.length) {
    // [2m+4] +1 Zuweisung, +1 Zugriff, +1 Mult, +(2m+1) new (Laenge 2m)
    double[] c = new double[2 * l.size];

    // [1] Initialisierung: +1 Zuweisung
    for (int i = 0; i < l.size; i++) {
      // [2] Bedingung: +1 Vergleich, +1 Zugriff
      // [6] Body + Update: +2 Zuweisung, +2 Feldzugriff, +1 Zugriff, +1 Plus
      c[i] = l.contents[i];
    }

    // [2] +1 Zuweisung, +1 Zugriff
    l.contents = c;
  }

  // [7] +2 Zuweisung, +4 Zugriffe, +1 Feldzugriff, +1 Plus
  l.contents[l.size] = elem;
  l.size += 1; // Koennte man auch als +2 Zugriffe zaehlen.
}

static int size(List l) {
  return l.size;
}

static double get(List l, int index) {
  return l.contents[index];
}

static void remove(List l, int index) {
  if (index < 0 || index >= l.size) return;

  for (int i = 0; i < l.size - 1; i++) {
    if (i >= index)
      l.contents[i] = l.contents[i+1];
  }

  l.size -= 1;
}
```

c) Führen Sie erneut eine Analyse der Zeitkomplexität durch, wie Sie es in a) für die erste Implementierung getan haben. Gehen Sie hier allerdings vereinfachend davon aus, dass es ein k mit $n = 2^k + 1$ gibt.

Die Aufwandsfunktion $A_{appThen}(m)$ ist auch hier abhängig von der Länge der Liste, die erweitert wird. Da der Fall nur eintritt, wenn das interne Array voll belegt ist, ist die Größe des Arrays gleich der Anzahl m der Elemente in der Liste.

$$
\begin{aligned}
A(n) &= A_{create} + \sum_{i=0}^{n-1} A_{append}(i) \\
A_{create} &= 10 \\
A_{appElse} &= 11 \\
A_{appThen}(m) &= A_{appElse} + 9 + 2m + \sum_{i=0}^{m-1} 8 \\
&= 10m + 20 \\
A_{append}(m) &= \begin{cases} A_{appThen}(m) & \text{falls } m = 2^k \\ A_{appElse} & \text{sonst} \end{cases}
\end{aligned}
$$

Nun müssen wir überlegen, wie oft die beiden Varianten in $A_{append}(m)$ auftauchen und mit welchem Parameter. Die Variable m läuft über das betrachtete Programm von 0 bis $n - 1$. Bei allen Zweierpotenzen wird letztendlich dann $A_{appThen}(m)$ benutzt, bei allen anderen Fällen der Konstante Aufwand von $A_{appElse}$. Im Allgemeinen wäre die Anzahl der Zweierpotenzen $\lfloor \log_2 (n-1) \rfloor$, durch die Zusatzvoraussetzung sind dies aber einfach k.

$$
\begin{aligned}
\rightsquigarrow A(n) &= A_{create} + \sum_{i=1}^{k} A_{appThen}(2^i) + (n-k) \cdot A_{appElse} \\
&= 10 + \sum_{i=1}^{k} \left(10 \cdot 2^i + 20\right) + (n-k) \cdot 11 \\
&= 10 + \sum_{i=1}^{k} \left(10 \cdot 2^i + 20\right) + 11n - 11k \\
&= 10 + 10 \sum_{i=1}^{k} 2^i + 11n + 9k \\
&= 10 + 10 \cdot \left(2^{k+1} - 2\right) + 11n + 9k \\
&= -10 + 20 \cdot 2^k + 11n + 9k \\
&= -10 + 20 \cdot (n-1) + 11n + 9k \\
&= 31n + 9 \log_2 (n-1) - 30 \\
&< 40n \\
\rightsquigarrow A &\in O(n) \quad \text{mit } c = 40 \text{ und } d = 0
\end{aligned}
$$

Aufgabe 4 Routenplanung (Einreichaufgabe)

a) In der Grafik auf der nächsten Seite sehen Sie die Repräsentation des obigen Stadtplans als Geflecht. Erstellen Sie Verbundtypen gemäß dieser Graphik und Code um den Stadtplan als Geflecht anzulegen.

```
class Knoten {
  Kante[] ausgehend;
  Knoten pred;
  int dist;
  boolean inB;
}

class Kante {
  int gewicht;
  Knoten ende;
}
```

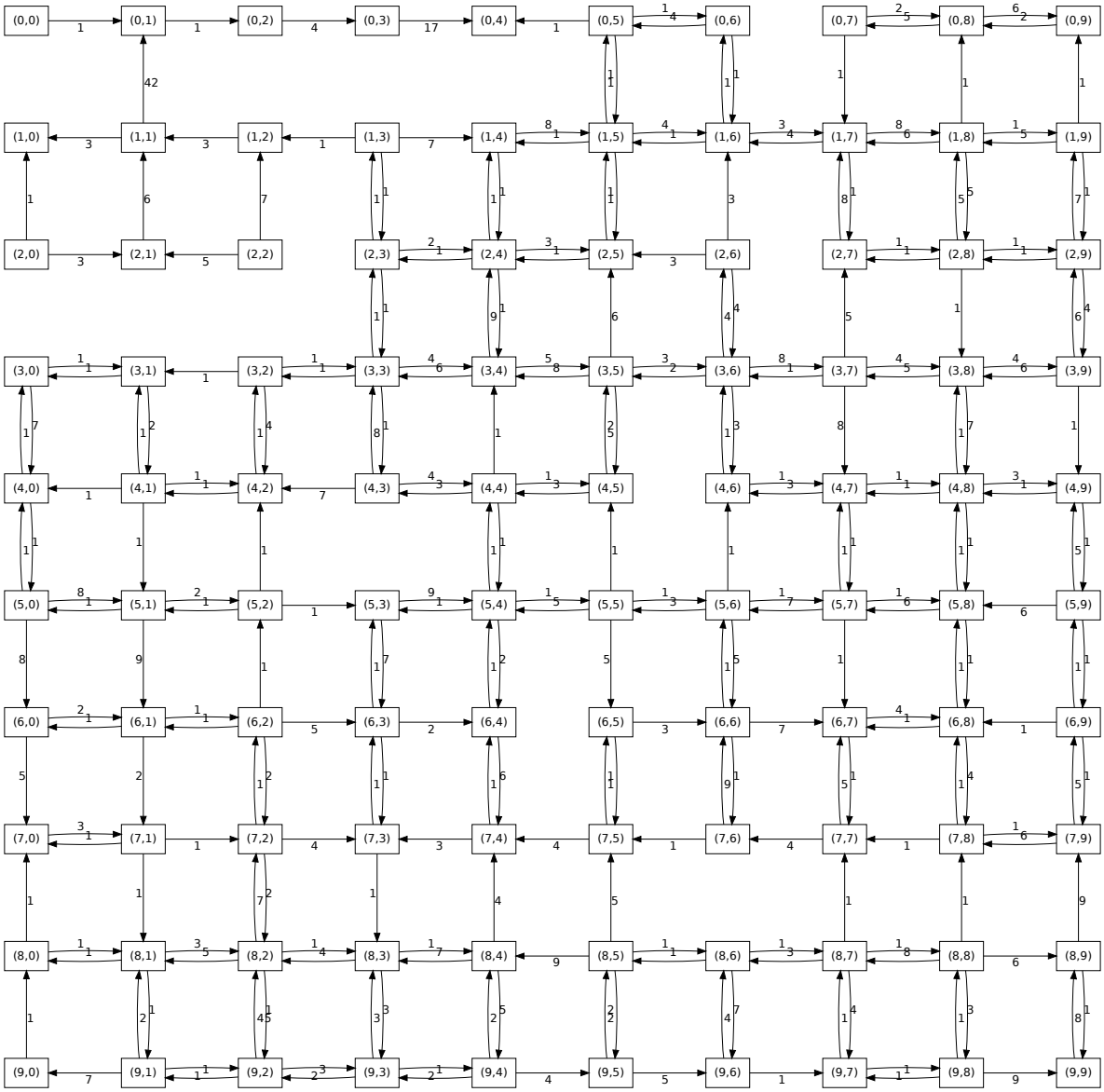
b) Implementieren Sie nun den in der Vorlesung entwickelten Algorithmus zum Finden der kürzesten Route. Testen Sie ihn am obigen Beispiel.

```
static Knoten[] sucheRoute(Knoten[] knoten, Knoten start, Knoten ziel) {
  initialisiere(knoten);

  // Anfang initialisieren
  start.pred = start;
  start.dist = 0;
  start.inB = true;

  R.Rand rand = R.erzeugeRand();

  ergaenzeRand(start, rand);

  while (!R.leer(rand)) {
    Knoten min = R.naechsterKnoten(rand);

    R.entfernen(rand, min);
    min.inB = true;
    ergaenzeRand(min, rand);
  }

  // Hier kann einfach wieder die eigene Listenimplementierung genommen werden.
  List<Knoten> pfad = new ArrayList<Knoten>();
  Knoten current = ziel;

  while (current != start) {
    pfad.add(0, current);
    current = current.pred;
  }
  pfad.add(0, current);

  return pfad.toArray(new Knoten[pfad.size()]);
}

static void ergaenzeRand(Knoten v, R.Rand rand) {
  for (int i = 0; i < v.ausgehend.length; i++) {
    Knoten w = v.ausgehend[i].ende;
    if (!w.inB && v.dist + v.ausgehend[i].gewicht < w.dist) {
      w.pred = v;
      w.dist = v.dist + v.ausgehend[i].gewicht;
      R.einfuegen(rand, w);
    }
  }
}
```

```
static void initialisiere(Knoten[] knoten) {
  for (int i = 0; i < knoten.length; i++) {
    knoten[i].pred = null;
    knoten[i].dist = Integer.MAX_VALUE;
    knoten[i].inB = false;
  }
}
```

c) In dem heruntergeladenen Code ist auch ein Stadtplan enthalten. Testen Sie Ihre Routenplanung mit diesem Anhand der Start- und Zielknoten-Paare, die in der Methode `main` stehen. Entfernen Sie dazu die entsprechenden Kommentarzeichen. Eine Lösung ist jeweils darunter angegeben. Den gesamten Stadtplan finden Sie in der Datei Stadtplan.pdf.

Zur schnellen Referenz, wenn er gebraucht wird:



Aufgabe 5 Speicherverwaltung in Java (Präsenzaufgabe)

Kreuzen Sie an, ob folgende Aussagen wahr oder falsch sind.

wahr	falsch	
☐	☒	Speicher für Verbunde und Felder wird auf dem Stack allokiert.
☐	☒	Lokale Variablen werden auf dem Heap angelegt.
☒	☐	Der Stackspeicher wird vom Programmierer nicht selbst verwaltet. *Zur Verwaltung des Laufzeitkellers wird vom Compiler Code erzeugt, der diese Aufgabe übernimmt. Der Programmierer hat nichts damit zu tun.*
☐	☒	Der Programmierer muss den Heap selbst verwalten. *Java ist eine Sprache mit automatischer Speicherbereinigung, nicht mehr benötigter Speicher wird automatisch freigegeben.*
☒	☐	Ein nicht erreichbares Objekt kann von der Speicherbereinigung aus dem Speicher entfernt werden.
☐	☒	Ein nicht erreichbares Objekt kann später im Programm wieder erreichbar sein.
☒	☐	Mit `new` wird Speicher auf dem Heap allokiert.
☒	☐	Das Nichtfreigeben von Speicher ist in Sprachen ohne automatische Speicherbereinigung eine häufige Problemquelle.
☒	☐	Ein `new`-Ausdruck kann nicht immer erfolgreich ausgewertet werden. *Wenn nicht mehr genügend Haldenspeicher zur Verfügung steht, kann natürlich kein neues Objekt mehr angelegt werden und das Programm wird einen "Out of Memory"-Fehler erzeugen.*