be-OI 2023 **Finale - KADETT** Samstag 18. März 2023	**Füllt diesen Rahmen bitte in GROSSBUCHSTABEN und LESERLICH aus** VORNAME : .... NAME : .... SCHULE :	0 **Reserviert**

Belgische Informatik-Olympiade 2023 (Dauer: maximal 2 Stunden)

Allgemeine Hinweise (bitte sorgfältig lesen bevor du die Fragen beantwortest)

1. Überprüfe, ob du die richtigen Fragen erhalten hast. (s. Kopfzeile oben):
   - Für Schüler bis zum zweiten Jahr der Sekundarschule: Kategorie **Kadett**.
   - Für Schüler im dritten oder vierten Jahr der Sekundarschule: Kategorie **Junior**.
   - Für Schüler der Sekundarstufe 5 und höher: Kategorie **Senior**.
2. Gib deinen Namen, Vornamen und deine Schule **nur auf dieser Seite** an.
3. **Die Antworten** sind auf den dafür vorgesehenen Antwortbogen einzutragen. Schreibe **gut lesbar** mit einem **blauen oder schwarzen Stift oder Kugelschreiber**.
4. Verwende einen Bleistift und einen Radiergummi, wenn du am Entwurf in den Frageblättern arbeitest.
5. Du darfst nur Schreibmaterial dabei haben; Taschenrechner, Mobiltelefone, ... sind **verboten**.
6. Du kannst jederzeit weitere Kladdeblätter bei einer Aufsichtsperson fragen.
7. Wenn du fertig bist, gibst du die erste Seite (mit deinem Namen) und die letzten Seiten (mit den Antworten) ab, du kannst die anderen Seiten behalten.
8. Alle Codeauszüge aus der Anweisung sind in **Pseudo-Code**. Auf den folgenden Seiten findest du eine **Beschreibung** des Pseudo-Code, den wir verwenden.
9. Wenn du mit einem Code antworten musst, dann benutze den **Pseudo-Code** oder eine **aktuelle Programmiersprache** (Java, C, C++, Pascal, Python,....). Syntaxfehler werden bei der Auswertung nicht berücksichtigt.

Viel Erfolg!

Die belgische Olympiade der Informatik ist möglich dank der Unterstützung unserer Mitglieder:

©2023 Belgische Informatik-Olympiade (beOI) ASBL
Dieses Werk wird unter den Bedingungen der Creative Commons Attribution 2.0 Belgium License zur Verfügung gestellt.

Pseudocode Checkliste

Die Daten werden in Variablen gespeichert. Wir ändern den Wert einer Variablen mit $\leftarrow$. In einer Variablen können wir ganze Zahlen, reelle Zahlen oder Arrays (Tabellen) speichern (siehe weiter unten), sowie boolesche (logische) Werte: wahr/richtig (**`true`**) oder falsch/fehlerhaft (**`false`**). Es ist möglich, arithmetische Operationen auf Variablen durchzuführen. Zusätzlich zu den vier traditionellen Operatoren ($+$, $-$, $\times$ und $/$), kann man auch den Operator $\%$ verwenden. Wenn a und b ganze Zahlen sind, dann stellt a/b den Quotienten und $a\%b$ den Rest der ganzen Division dar.
Zum Beispiel, wenn $a = 14$ und $b = 3$, dann $a/b = 4$ und $a\%b = 2$.
Hier ist erstes Beispiel für einen Code, in dem die Variable *alter* den Wert 17 erhält.

```
geburtsjahr ← 2006
alter ← 2023 − geburtsjahr
```

Um Code nur auszuführen, wenn eine bestimmte Bedingung erfüllt ist, verwendet man den Befehl **`if`** und möglicherweise den Befehl **`else`**, um einen anderen Code auszuführen, wenn die Bedingung falsch ist. Das nächste Beispiel prüft, ob eine Person volljährig ist und speichert den Eintrittspreis für diese Person in der Variable *preis*. Beobachte die Kommentare im Code.

```
if (alter ≥ 18)
{
    preis ← 8 // Das ist ein Kommentar.
}
else
{
    preis ← 6 // billiger!
}
```

Manchmal, wenn eine Bedingung falsch ist, muss eine andere überprüft werden. Dazu können wir **`else if`** verwenden, was so ist, als würde man ein anderes **`if`** innerhalb des **`else`** des ersten **`if`** ausführen. Im folgenden Beispiel gibt es 3 Alterskategorien, die 3 unterschiedlichen Preisen für das Kinoticket entsprechen.

```
if (alter ≥ 18)
{
    preis ← 8 // Preis fuer eine erwachsene Person.
}
else if (alter ≥ 6)
{
    preis ← 6 // Preis fuer ein Kind ab 6 Jahren.
}
else
{
    preis ← 0 // Kostenlos fuer ein Kind unter 6 Jahren.
}
```

Um mehrere Elemente mit einer einzigen Variablen zu manipulieren, verwenden wir ein Array. Die einzelnen Elemente eines Arrays werden durch einen Index gekennzeichnet (in eckigen Klammern nach dem Namen des Arrays). Das erste Element eines Arrays *tab*[ ] hat den Index 0 und wird mit *tab*[0] bezeichnet. Der zweite hat den Index 1 und der letzte hat den Index $n - 1$, wenn das Array n Elemente enthält. Wenn beispielsweise das Array *tab*[ ] die 3 Zahlen 5, 9 und 12 (in dieser Reihenfolge) enthält, dann *tab*[0]= 5, *tab*[1]= 9, *tab*[2]= 12. Das Array hat die Länge 3, aber der höchste Index ist 2.

Um Code zu wiederholen, z.B. um durch die Elemente eines Arrays zu laufen, kann man eine **for**-Schleife verwenden. Die Schreibweise **for** $(i \leftarrow a$ **to** b **step** $k)$ stellt eine Schleife dar, die so lange wiederholt wird, wie $i \leq b$, in der i mit dem Wert a beginnt und wird am Ende jedes Schrittes um den Wert k erhöht. Das folgende Beispiel berechnet die Summe der Elemente in dem Array $tab[\,]$. Angenommen, das Array ist n lang. Die Summe befindet sich am Ende der Ausführung des Algorithmus in der Variable sum.

```
summe ← 0
for (i ← 0 to n−1 step 1)
{
    summe ← summe + tab[i]
}
```

Sie können eine Schleife auch mit dem Befehl **while** schreiben, der den Code wiederholt, solange seine Bedingung wahr ist. Im folgenden Beispiel wird eine positive ganze Zahl n durch 2 geteilt, dann durch 3, dann durch 4 ..., bis sie nur noch aus einer Ziffer besteht (d.h. bis $n < 10$).

```
d ← 2
while (n ≥ 10)
{
    n ← n/d
    d ← d + 1
}
```

Häufig befinden sich die Algorithmen in einer Struktur und werden durch Erklärungen ergänzt. Nach `Eingabe` wird jedes Argument (Variabel) definiert, die als Eingaben für den Algorithmus angegeben werden. Nach `Ausgabe` wird der Zustand bestimmter Variablen am Ende der Algorithmusausführung und möglicherweise der zurückgegebenen Werte definiert. Ein Wert kann mit dem Befehl **return** zurückgegeben werden. Wenn dieser Befehl ausgeführt wird, stoppt der Algorithmus und der angegebene Wert wird zurückgegeben.

Hier ist ein Beispiel für die Berechnung der Summe der Elemente eines Arrays.

```
Eingabe: tab[ ], ein Array mit n Zahlen.
         n, die Anzahl der Elemente des Arrays.
Ausgabe: sum, die Summe aller im Array enthaltenen Zahlen.

summe ← 0
for (i ← 0 to n−1 step 1)
{
    summe ← summe + tab[i]
}
return summe
```

Hinweis: In diesem letzten Beispiel wird die Variable i nur als Zähler für die Schleife **for** verwendet. Es gibt also keine Erklärung darüber in `Eingabe` oder `Ausgabe`, und ihr Wert wird nicht zurückgegeben.

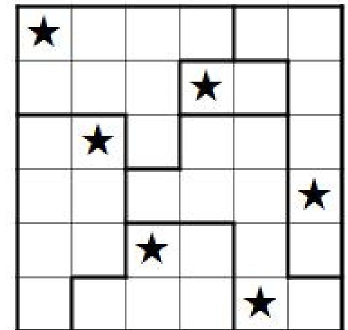
Frage 1 – Star Battle

Star Battle ist ein Logikspiel, bei dem es darum geht, **n** Sterne in einem Raster aus **n** Zeilen und **n** Spalten, das in **n** Zonen aufgeteilt ist, die durch dicke Mauern abgetrennt sind (dargestellt durch dicke Striche), zu verteilen.

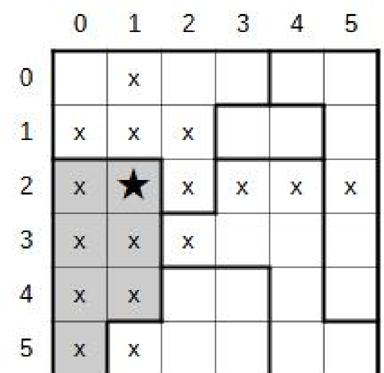
Dabei sind folgende Regeln zu beachten.

- **Regel 1:** Jede Zeile muss genau 1 Stern enthalten.
- **Regel 2:** Jede Spalte muss genau 1 Stern enthalten.
- **Regel 3:** Jede Zone muss genau 1 Stern enthalten.
- **Regel 4:** Die Felder um einen Stern herum dürfen keinen weiteren Stern enthalten.

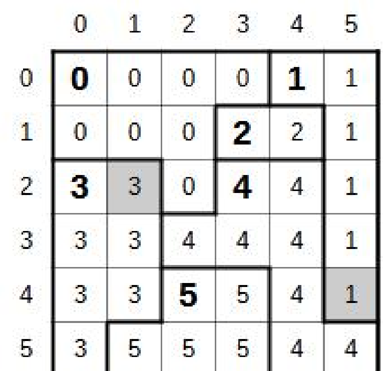
Beispiel: Das Raster rechts ist nach diesen Regeln aufgefüllt worden.



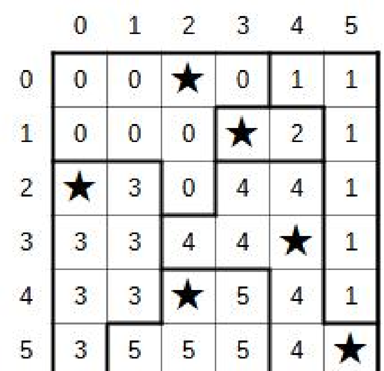
Hier ein Beispiel mit **n=6** wo nur ein Stern plaziert wurde.
Die Nummern der Zeilen (von 0 bis 5) sind auf der linken Seite des Rasters eingetragen.
Die Nummern der Spalten (von 0 bis 5) stehen über dem Raster.
Der Stern wurde am Schnittpunkt von Zeile 2 und Spalte 1 platziert.
Felder, auf denen keine weiteren Sterne mehr platziert werden können, sind mit einem ‘x’ markiert:
• die Felder in Zeile 2 (Regel 1),
• die Felder in Spalte 1 (Regel 2),
• die Felder (grau hinterlegt) der Zone in der sich der Stern befindet (Regel 3),
• die 8 Felder um den Stern (Regel 4).



Wir nummerieren die Felder von **0** bis **n-1**, indem wir uns Feld für Feld “in Leserichtung” von der oberen linken Ecke des Rasters aus bewegen.
0 wird in alle Felder der ersten Zone eingetragen, **1** wird in alle Felder der nächsten Zone eingetragen, mit den weiteren Nummern wird so weiter gemacht, bis **n-1** in den Feldern der letzten Zone eingetragen wurde (Die Zone **5** im Beispiel).
Die ersten Felder jeder Zone “in Leserichtung” sind fett gedruckt.
In den Programmen werden die Zonennummern des Rasters in einem Array **t[ ][ ]** gespeichert.
• **t[2][1]=3**, weil das Feld in der Zeile 2 und der Spalte 1 (in grau) in der Zone 3 ist.
• **t[4][5]=1**, weil das Feld in der Zeile 4 und Spalte 5 (auch in grau) in Zone 1 ist.



Der Spieler schlägt eine Lösung anhand eines Arrays **col[ ]** vor.
Für jede Zeilennummer **i** muss der Spieler die Nummer der Spalte **col[i]** angeben in der er einen Stern für diese Zeile platzieren möchte.
Im rechten Beispiel, **col[ ]=[2,3,0,4,2,5]**
(Kurze Schreibweise für **col[0]=2**, **col[1]=3**, **col[2]=0**, **col[3]=4**, **col[4]=2** und **col[5]=5**).
Das ist ein schlechter Vorschlag, denn nur die Regel 1 wurde respektiert.
• Die Regel 2 wurde nicht respektiert da mehrere Sterne in der Spalte 2 stehen.
• Die Regel 3 wurde nicht respektiert da mehrere Sterne in der Zone 4 stehen.
• Die Regel 4 wurde nicht durch die Sterne der Zeilen 0 und 1 respektiert.



Die Fragen dieser Seite benutzen das Raster des **Star Battle**, das dreimal hier unten angezeigt wird.
(Die Lösung wird im Raster rechts angezeigt.)

Q1(a) /4	**Welche Werte enthalten die Elemente der Tabelle t[ ][ ] für dieses Raster?** `t[1][5]=..., t[2][3]=..., t[3][0]=..., t[4][3]=...`
Lösung: `t[1][5]`=1, `t[2][3]`=2, `t[3][0]`=3, `t[4][3]`=5	

Für die folgenden 4 Fragen musst du die Kästchen der Regeln **ankreuzen**, die **respektiert** sind und die Kästchen der Regeln *leer lassen*, die *nicht respektiert* sind. Du erhälst die Punkte nur, wenn du alle richtigen Kästchen angekreuzt hast.

Q1(b) /2	**Kreuze die Kästchen der Regeln an, die respektiert sind, wenn col[ ]=[1, 5, 0, 3, 1, 4].** ☒ Regel 1 ☐ Regel 2 ☒ Regel 3 ☒ Regel 4

Q1(c) /2	**Kreuze die Kästchen der Regeln an, die respektiert sind, wenn col[ ]=[1, 5, 3, 0, 2, 4].** ☒ Regel 1 ☒ Regel 2 ☐ Regel 3 ☒ Regel 4

Q1(d) /2	**Kreuze die Kästchen der Regeln an, die respektiert sind, wenn col[ ]=[2, 3, 0, 4, 1, 5].** ☒ Regel 1 ☒ Regel 2 ☐ Regel 3 ☐ Regel 4

Q1(e) /2	**Kreuze die Kästchen der Regeln an, die respektiert sind, wenn col[ ]=[2, 5, 3, 0, 1, 4].** ☒ Regel 1 ☒ Regel 2 ☒ Regel 3 ☐ Regel 4

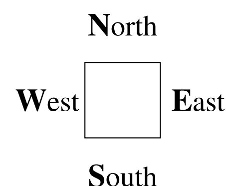
Finde die Lösung!

Q1(f) /2	**Plaziere 6 Sterne in das Raster, die die 4 Regeln respektieren.**
Lösung: Die Lösung wird im Raster oben rechts angezeigt.	

Q1(g) /2	**Was enthält die Tabelle col[ ] für deine Lösung?** `col[]=[..., ..., ..., ..., ..., ...]`
Lösung: col[ ]=[2, 5, 0, 3, 1, 4]	

Du musst das **Programm SB1** hier unten ausfüllen, das dazu dient, das Raster auf dem Bildschirm wie in dem ersten Beispiel auf der ersten Seite der Angabe anzuzeigen.
Das Programm kennt die Grösse **n** des Arrays **t[ ][ ]** und das Array **col[ ]**.

Das Programm benutzt einen Roboter, den man auf dem Bildschirm bewegen kann und der die Linien, Mauern und Sterne zeichnen kann. Das Programm beginnt mit dem Roboter an der richtigen Stelle, um das Feld oben links im Raster zu zeichnen, das der Zelle **t[0][0]** entspricht. Die Befehle `Go()`, `line()`, `wall()` und `star()` stehen zur Verfügung und werden unten erklärt.
`Go()`, `line()` und `wall()` benötigen eine Richtung W, E, N oder S.



- `Go(W,n)`, `Go(E,n)`, `Go(N,n)` und `Go(S,n)` bewegen den Roboter um **n** Felder in die entsprechende Richtung.
  Beispiele: `Go(E,1)` bewegt den Roboter um ein Feld nach rechts, `Go(S,2)` bewegt ihn um 2 Felder nach unten.
- Der Aufruf `wall()` zeichnet eine Mauer auf einen der Ränder des Feldes, auf dem er sich befindet.
  `wall(W)` zeichnet eine vertikale Mauer auf dem linken Rand und `wall(E)` eine vertikale Mauer rechts.
  `wall(N)` zeichnet eine horizontale Mauer auf dem oberen Rand `wall(S)` eine horizontale Mauer unten.
  Das Raster und die verschiedenen Zonen müssen mit Mauern umrandet werden.
- `line()` funktioniert wie `wall()`, aber anstelle einer Mauer wird eine feine Linie gezeichnet.
  Auf allen Rändern aller Zellen müssen eine Mauer oder eine feine Linie gezeichnet werden.
- `star()` zeichnet einen Stern in das Feld, auf dem sich der Roboter befindet.

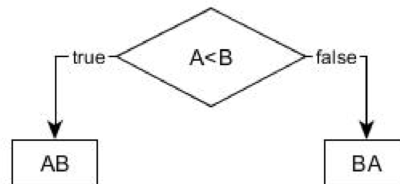
Wenn eine Bedingung mehrere logische Ausdrücke enthält, die mit einen **`or`** verbunden sind, dann ist diese Bedingung **`true`**, wenn mindestens einer der logischen Ausdrücke **`true`** ist.
Wenn der erste Ausdruck des **`or`** **`true`** ist, dann wird der zweite Ausdruck nicht mehr evaluiert (das ist unnötig).

Q1(h) /10	**Fülle die ____ im Programm SB1 aus.** **Punkte zwischen 0 und 10. 1 Punkt wird pro Fehler oder ausgelassene Antwort abgezogen.**
	Lösung: Die Antworten werden hier unten auf grauem Hintergrund angezeigt.

```
Input  : n, t[][], col[]
for (i ← 0 to n−1 step 1) {
   for (j ← 0 to n−1 step 1) {
      if (i=0 or t[i][j] ≠ t[i−1][j]) {wall(N)}
      else { line(N) }

      if (i=n−1) {wall(S)}

      if (j=0 or {t[i][j] ≠ t[i][j−1]) {wall(W)}
      else {line(W)}

      if (j=n−1) {wall(E)}

      if (j=col[i]) {star()}

      Go(E, 1)
   }
   Go(W, n)
   Go(S , 1)
}
```

Frage 2 – Flowchart

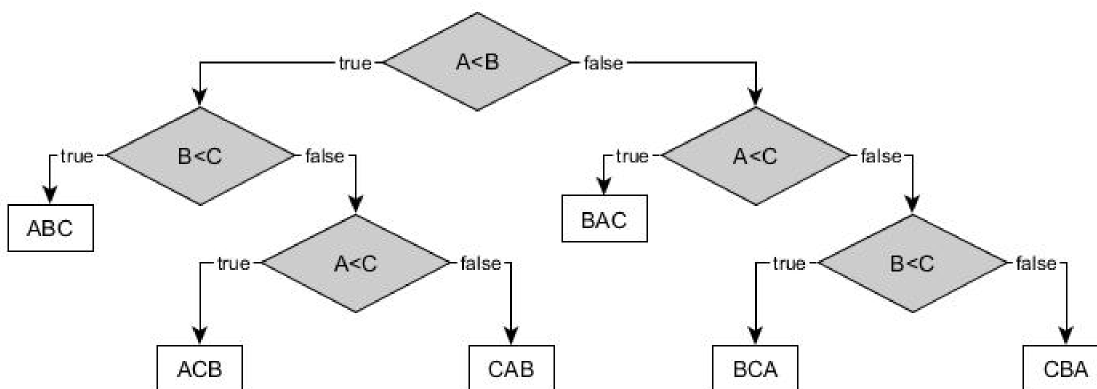
Man kann einen Test durch eine Raute darstellen, in die man die zu prüfende Bedingung schreibt.
Je nach Ergebnis folgt man entweder dem Pfeil **`true`** oder dem Pfeil **`false`**.
Im folgenden Beispiel sind `A` und `B` Variablen, die verglichen werden sollen.
Wenn `A<B`, sollte die Nachricht "**AB**" angezeigt werden, andernfalls sollte "**BA**" angezeigt werden.
Die Operationen zur Anzeige von Nachrichten werden durch die Rechtecke dargestellt.



Wenn beispielsweise `A=7` und `B=4`, folgt der obige Algorithmus dem Pfeil **`false`** des Tests und zeigt "**BA**" an.

Kommen wir zu einem ähnlichen Problem mit drei Variablen `A`, `B` und `C`, die unterschiedliche **Werte** haben.
Du sollst einen Algorithmus grafisch darstellen, der die drei Buchstaben "**A**", "**B**" und "**C**" in der Reihenfolge des kleinsten bis zum größten Wert anzeigt. Wenn beispielsweise `A=11`, `B=16` und `C=7`, sollte der Algorithmus "**CAB**" ausgeben, da `C` den kleinsten Wert und `B` den größten Wert hat.
In jeder Raute musst du einen Test platzieren. Du kannst nur die folgenden sechs Tests verwenden: `A<B`, `B<A`, `A<C`, `C<A`, `B<C` und `C<B`.

Q2(a) /5	Fülle unten stehenden *dots* aus. Für jedes richtig ausgefüllte Feld erhältst du einen Punkt.
Lösung: Die Lösungen sind grau unterlegt.	



Nehmen wir nun an, dass die Variablen identische **Werte** haben können.

Q2(b) /3	Welche Meldung wird angezeigt, wenn alle drei Variablen gleich sind?
Lösung: CBA	

Q2(c) /3	Welche Meldung wird nie angezeigt, wenn 2 Variablen gleich sind?
Lösung: ABC	

Frage 3 – Listenkreuzung

Es sollen alle Elemente gefunden werden, die zwei Listen mit n Elementen $L1[\,]$ und $L2[\,]$ gemeinsam haben.
Es ist bekannt, dass die Elemente von $L1[\,]$ alle unterschiedlich sind, ebenso die Elemente von $L2[\,]$.
Es soll nun die Liste $Li[\,]$ aller Elemente erstellt werden, die sich sowohl in $L1[\,]$ als auch in $L2[\,]$ befinden.
Zum Beispiel, wenn $n = 8$, $L1[\,] = [1, 3, 6, 10, 15, 21, 28, 36]$ und $L2[\,] = [3, 7, 15, 19, 23, 29, 36, 41]$ dann $Li[\,] = [3, 15, 36]$.
Wie üblich nummerieren wir die Elemente ab 0. **Erste Idee: das Programm P1.**
Es verwendet zwei ineinander verschachtelte **for**-Schleifen.
Die erste Schleife wählt bei jedem Durchlauf ein neues Element $x1$ aus $L1[\,]$ aus.
Die zweite Schleife innerhalb der ersten Schleife vergleicht nacheinander $x1$ mit allen Elementen in $L2[\,]$.
Wenn ein Gleichstand gefunden wird, fügt man $x1$ in die Liste $Li[\,]$ ein.
P1 verwendet `Li[]←[]`, um eine leere $Li[\,]$-Liste zu erstellen.
Er verwendet die Methode `append()`, um **ein Element am Ende einer Liste hinzufügen**.
Wenn $Li[\,]$ zum Beispiel leer ist und wir `Li[].append(3)` ausführen, dann ist $Li[\,] = [3]$.
Wenn wir dann `Li[].append(15)` und anschließend `Li[].append(36)` ausführen, dann gilt $Li[\,] = [3, 15, 36]$.

Beachte in **P1** die Zeilen (1) und (2), die mit einer Nummer in einem grauen Rechteck hervorgehoben sind.

```
Input : L1[], L2[], n       Output : Li[]
Li[] ← []
for i1 ← 0 to n-1 step 1 {
    x1 ← L1[i1]         // (1)
    for i2 ← 0 to n-1 step 1 {
        if (L2[i2]=x1)      // (2)
            {Li[].append(x1)}
    }
}
```

Q3(a) /1	**Wenn $n = 100$, wie oft wird Zeile (1) ausgeführt?**
Lösung: 100	

Q3(b) /1	**Wenn $n = 100$, wie oft wird Zeile (2) ausgeführt?**
Lösung: 10000	

Q3(c) /1	**Allgemein gesagt: Wie oft wird für ein gegebenes n die Zeile (1) ausgeführt?**
Lösung: n	

Q3(d) /1	**Allgemein gesagt: Wie oft wird für ein gegebenes n die Zeile (2) ausgeführt?**
Lösung: n^2	

In den folgenden Fragen wird **P1** mit unterschiedlich langen Listen getestet.
Wir nehmen an, dass die Gesamtausführungszeit nur davon abhängt, wie oft **Zeile (2)** ausgeführt wird. (das ist eine Vereinfachung, aber genau genug für uns).

Q3(e) /1	**Wenn die Ausführungszeit für ein bestimmtes n 3 Sekunden beträgt, wie lange, in $Sekunden$, dauert es dann für 5-mal so lange Listen?**
Lösung: $3 \cdot 5 \cdot 5 = 75$ Sekunden	

Q3(f) /1	**Wenn die Ausführungszeit für ein bestimmtes n 6 Sekunden beträgt, wie lange, in $Minuten$ ausgedrückt, dauert es dann für 10-mal so lange Listen?**
Lösung: $6 \cdot 10 \cdot 10 = 600$ Sekunden = 10 Minuten	

Q3(g) /1	**Wenn die Ausführungszeit für ein bestimmtes n 4 Sekunden beträgt, wie lange, in $Stunden$ ausgedrückt, dauert es dann für 60-mal so lange Listen?**
Lösung: $4 \cdot 60 \cdot 60$ Sekunden = $4 \cdot 60$ Minuten = 4 Stunden	

Zweite Idee: das Programm P2.
P1 funktioniert mit jeder Liste, egal ob sortiert oder unsortiert, aber es ist langsam.
P2 ist schneller, funktioniert aber nur, wenn die Listen in aufsteigender Reihenfolge sortiert sind.

Hier ist das Listing von **P2** ohne Erklärungen.
Beachte die Zeile (3), die mit einer Nummer in einem grauen Rechteck hervorgehoben ist.

```
Input : L1[], L2[], n        Output : Li[]
Li[] ← []
i1 ← 0
i2 ← 0
while (i1 < n and i2 < n) {
    x1 ← L1[i1]          // (3)
    x2 ← L2[i2]
    if (x1 < x2) {i1 ← i1 + 1}
    else {
        if (x1 = x2) {Li[].append(x1)}
        i2 ← i2 + 1
    }
}
```

Q3(h) /1	**Wenn $n = 10$, wie oft wird Zeile (3) MINimal ausgeführt?**
Lösung: 10	

Q3(i) /1	**Wenn $n = 10$, wie oft wird Zeile (3) MAXimal ausgeführt?**
Lösung: 19	

Q3(j) /1	**Wenn $n = 10$ und $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$, dann gib eine Liste $L2[\,]$ von verschiedenen ganzen Zahlen > 0 an, so dass Zeile (3) MINimal oft ausgeführt wird.**
Lösung: Jede Liste, deren erstes Element größer als 19 ist, z. B. `L2[] = [20,21,22,23,24,25,26,27,28,29]`.	

Q3(k) /1	**Wenn $n = 10$ und $L1[\,] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$, gib eine Liste $L2[\,]$ verschiedener ganzer Zahlen > 0 an, so dass Zeile (3) MAXimal oft ausgeführt wird.**
Lösung: Es gibt viele Lösungen: `L2[]=[2,4,6,8,10,12,14,16,18,n]` mit $n \geq 20$ oder `L2[]=L1[]=[1,3,5,7,9,11,13,15,17,19]`	

Q3(l) /1	**Wie oft wird für ein gegebenes n die Zeile (3) MINimal ausgeführt?**
Lösung: n	

Q3(m) /1	**Wie oft wird für ein gegebenes n die Zeile (3) MAXimal ausgeführt?**
Lösung: $2n - 1$	

In den folgenden Fragen wird **P2** mit unterschiedlich langen Listen getestet.
Wir nehmen an, dass die Gesamtausführungszeit nur davon abhängt, wie oft **Zeile (3)** ausgeführt wird. (das ist eine Vereinfachung, aber genau genug für uns).

Q3(n) /1	**Wenn die MAXimal-Ausführungszeit** 3 **Sekunden für ein bestimmtes** n **beträgt,** **wie lange wird dann die MAXimal-Ausführungszeit in** $Sekunden$ **für** 5**-mal so lange Listen sein?**
Lösung: $3 \cdot 5 = 15$ Sekunden	

Q3(o) /1	**Wenn die MAXimal-Ausführungszeit** 6 **Sekunden für ein bestimmtes** n **beträgt,** **wie lange wird die MAXimal-Ausführungszeit in** $Minuten$ **für** 10**-mal so lange Listen sein?**
Lösung: $6 \cdot 10 = 60$ Sekunden = 1 Minute	

Q3(p) /1	**Wenn die MAXimal-Ausführungszeit** 4 **Sekunden für ein bestimmtes** n **beträgt,** **wie lange wird die MAXimal-Ausführungszeit in** $Minuten$ **für** 60**-mal so lange Listen sein?**
Lösung: $4 \cdot 60$ Sekunden = 4 Minuten	

Frage 4 – Blocks

$L[\,]$ ist eine Liste mit n Zahlen $L[0], \ldots, L[n-1]$.
Die **Länge** einer Liste ist die Anzahl der Elemente, die sie enthält.
Ein **Block** von $L[\,]$ ist eine Unterliste, die so lang wie möglich ist, von aufeinanderfolgenden und untereinander gleichen Elementen von $L[\,]$.
Beispiel: die Blöcke von $L[\,] = [4, 2, 2, 2, 1, 1, 2]$ sind $[4]$, $[2, 2, 2]$, $[1, 1]$ und $[2]$, mit den Längen 1, 3, 2 und 1.

Die **zugeordnete Liste** $L[\,]$ ist die Liste der Längen der Blöcke von $L[\,]$.
Fortsetzung des Beispiels: Die Liste, die mit $[4, 2, 2, 2, 1, 1, 2]$ verknüpft ist, ist $[1, 3, 2, 1]$.

Q4(a) /1	**Welche Liste ist mit $[1, 2, 3, 4, 5]$ verknüpft ?**
Lösung: $[1, 1, 1, 1, 1]$	

Q4(b) /1	**Welche Liste ist mit $[3, 3, 3, 3, 3]$ verknüpft ?**
Lösung: $[5]$	

Q4(c) /2	**Was ist die Länge von $L[\,]$ wenn seine zugehörige Liste $[3, 2, 2, 3]$ ist ?**
Lösung: 10	

Q4(d) /2	**Welche Liste ist gleich ihrer eigenen zugehörigen Liste?**
Lösung: $[1]$	

In allen folgenden Fragen betrachten wir Listen $L[\,]$, **die nur mit** 1 **und** 2 **aufgefüllt sind und die mit** 1 **beginnen**.
Zum Beispiel $L[\,] = [1, 2, 2, 2, 1, 2]$ oder $L[\,] = [1, 1, 1, 1]$.
$A[\,]$ wird die Liste sein, die mit $L[\,]$ verknüpft ist.
Für die beiden vorangegangenen Beispiele gilt $A[\,] = [1, 3, 1, 1]$ und $A[\,] = [4]$.

Q4(e) /2	**Was ist $A[\,]$ wenn $L[\,]$ und $A[\,]$ als Länge 6 haben ?**
Lösung: $A[\,] = [1, 1, 1, 1, 1, 1]$	

Q4(f) /2	**Was ist $L[\,]$ wenn $L[\,]$ und $A[\,]$ als Länge 6 haben?**
Lösung: $L[\,] = [1, 2, 1, 2, 1, 2]$	

Fülle das folgende **Programm B1** hier unten aus, das als Eingabe eine Liste $L[\,]$ und als Länge n (mit $n > 0$) hat und als Ausgabe die Liste $A[\,]$ erzeugt, die mit $L[\,]$ verknüpft ist.

Das Programm verwendet `A[]←[]`, um eine leere Liste $A[\,]$ zu erstellen.
Das Programm verwendet die Methode `append()`, die dazu dient, **ein Element an das Ende einer Liste anzuhängen**.
Wenn beispielsweise $A[\,]$ leer ist und wir `A[].append(3)` ausführen, dann wird $A[\,] = [3]$.
Wenn wir dann noch `A[].append(1)` ausführen, dann wird $A[\,] = [3, 1]$.

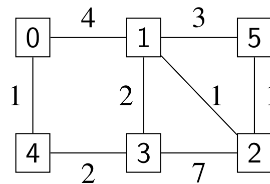
Q4(g) /6	**Vervollständige die ▭ im Programm B1.** **Fülle zwischen 0 und 6 aus. Du verlierst 1 Punkt für jeden Fehler oder jede fehlende Antwort.**
Lösung: Die Lösungen sind unten grau hinterlegt.	

```
Input  : n, L[]
Output : A[]
A[] ← []
value ← L[0]
length ← 0
for (i ← 0 to n-1 step 1)
{
    if (L[i] = value)
    {
        length ← length + 1
    }
    else
    {
        A[].append(length)
        value ← L[i]
        length ← 1
    }
}
A[].append(length)
return A[]
```

Frage 5 – Die Reise der Schlümpfe

Schlümpfe sind bekanntlich sehr reisefreudig! Sie müssen regelmäßig auf Expeditionen gehen, die von Papa Schlumpf geleitet werden. Aber diese Reisen sind sehr lang (« Ist es noch weit, Papa Schlumpf? ») und Papa Schlumpf verliert mit seinen 542 Jahren hin und wieder ein wenig die Geduld . . . .

Der Computerschlumpf beschließt, ihm zu helfen. Er wendet sich an den Geographen Schlumpf, der auf einer Karte (siehe unten) bekannte Wege im Zauberwald sowie die Zeit, die man braucht, um von einem Punkt zu einem anderen zu gelangen, darstellt:



Die Quadrate stehen für die verschiedenen Orte.
0 ist das Dorf (der Ausgangspunkt jeder Reise); 1 ist die große Eiche; 2 ist das Haus von Gargamel, *etc*.
Die Linien zwischen den Orten stellen die Wege dar.
Die Zahlen auf den Wegen geben die Zeit an, die man braucht, um von einem Ort zum anderen zu kommen.
Zum Beispiel dauert es 4 Stunden, um von 0 nach 1 zu gelangen.

Die Karte des Geographen Schlumpfs zeigt nur die direkten Wege. Der Computer-Schlumpf will die schnellste Route zu jedem Ort berechnen. Eine *Route* ist eine Reihe von Orten, die man vom Dorf aus auf direkten Wegen besuchen kann.
Beispielsweise ist 0, 4, 3, 2 eine Route, aber 0, 3, 2 nicht, da es keinen direkten Weg zwischen 0 und 3 gibt.
Die *Dauer* einer Route ist die Summe der Dauer der einzelnen Wege, aus denen sie besteht.
Zum Beispiel hat Route 0, 4, 3, 2 eine Dauer von $1 + 2 + 7 = 10$ Stunden.

Q5(a) /1	**Wie lange dauert die Route** 0, 1, 3, 2, 5**?**
Lösung: 14	

Q5(b) /1	**Kann man in genau 7 Stunden von** 0 **nach** 5 **fahren?** **Wenn deine Antwort "Jalautet, gib die Route an, andernfalls schreib einfach Nein".**
Lösung: 0, 1, 5	

Q5(c) /1	**Kann man in weniger als 5 Stunden von** 0 **nach** 5 **reisen?** **Wenn deine Antwort "Jalautet, gib die Route an, andernfalls einfach Nein".**
Lösung: Nein	

Q5(d) /1	**Welche ist die schnellste Verbindung von** 0 **nach** 5**?**
Lösung: 0, 1, 2, 5	

Der Computerschlumpf erklärt im Folgenden seinen Algorithmus zur Berechnung der Dauer der schnellsten Routen.

Mein Algorithmus verwendet eine Tabelle `D[]`, das mit den Nummern der Orte indiziert ist. Am Ende des Algorithmus enthält `D[x]` für jeden Ort `x` die Dauer der schnellsten Route zwischen dem Dorf und `x`.
Zu Beginn initialisiere ich (in einer **`for`** Schleife) `D[x]` auf $+\infty$ für alle Orte `x`.
Aber direkt danach ändere ich `D[0]` auf 0, da wir immer noch vom Dorf aus starten.

Das Symbol $+\infty$ bedeutet « Unendlich ».
Das ist sozusagen das mathematische Äquivalent zu « Weit weg, meine kleinen Schlümpfe! ».
Es ist ein besonderer Wert, der größer als jede andere Zahl ist.
Wenn man einen ganzzahligen Wert zu $+\infty$ addiert, erhält man immer $+\infty$.
Z.B. : $+\infty + 5 = +\infty$

Während seiner Ausführung unterscheidet mein Algorithmus zwischen zwei Arten von Orten.

- Orte, für die man bereits den schnellsten Weg vom Dorf aus kennt.
  An diesen Orten wird sich der Wert von `D[x]` bis zum Ende der Ausführung nicht mehr ändern.
  Der Einfachheit halber nennen wir sie *bekannte* Orte.
- Bei den Orten, für die die schnellste Route vom Dorf aus noch nicht bekannt ist, kann sich der Wert von `D[x]` während der Ausführung noch ändern.
  Der Einfachheit halber nennen wir sie *unbekannte* Orte.

Anfangs geht der Algorithmus davon aus, dass alle Orte *unbekannt* sind.
Am Ende, wenn alle schnellsten Routen berechnet wurden, sind alle Orte *bekannt*.

In der Praxis verwendet der Algorithmus eine Boolesche Tabelle `K[]`, so dass `K[x]` **`true`** für die *bekannten* Orte **`false`** für die *unbekannten* Orte lautet.

Sobald man sicher ist, dass `D[x]` sich nicht mehr ändert, das heißt `x` *bekannt* wird, können wir diese Information festhalten, indem wir **`true`** in `K[x]` schreiben.

Die wichtigsten Operationen des Algorithmus finden in einer **`while`** Schleife statt.
Hier ist, was in jeder Iteration dieser Schleife durchgeführt wird.

- Wähle aus allen *unbekannten* Orten den Ort `m` so aus, dass `D[m]` minimal ist.
- Ändere `m` von einem *unbekannten* Ort zu einem *bekannten* Ort.
- Untersuche alle *Nachfolger* von `m`, um zu sehen, ob ich eine schnellere Route zu ihnen finden kann als die, die du bereits kennst. Die Nachfolger von `m` sind alle Orte, die man von `m` aus auf direktem Weg erreichen kann.
  Die Nachfolger von 0 sind zum Beispiel 1 und 4.
  Ich schaue mir an, wie lange `D[s]` die schnellste bekannte Route für jeden Nachfolger `s` dauert. und versuche, sie mithilfe einer Route durch `m` zu verbessern.

Schließlich muss ich euch noch erklären, wie ich die Karte mit den Wegen darstelle.
Ich verwende eine Tabelle `G[][]`, die wie folgt aussieht `G[i][j]` ist die Dauer des direkten Weges vom Ort `i` zum Ort `j` falls dieser Weg möglich ist.
Wenn es keinen direkten Weg vom Ort i zum Ort j gibt, dann bezeichne ich das als $+\infty$ in `G[i][j]`. Mit der Karte vom Geographen Schlumpf hat man zum Beispiel `G[0][1]=4` und `G[0][3]=`$+\infty$.

Der Algorithmus verwendet die Funktion `minFalse(D[],K[])`, die die Nummer (zwischen 0 und $n-1$) vom *unbekannten* Ort mit dem kleinsten Wert im `D[]` zurückgibt.
Wenn es mehrere gleichrangige *unbekannte* Orte gibt, gibt die Funktion den Ort mit der niedrigsten Nummer zurück.
Wenn alle Orte bekannt sind, gibt die Funktion -1 zurück.
Beispiel: Wenn die unbekannten Orte 2, 4 und 5 sind und wenn `D[]=[0,7,9,15,17,9]` ist,
dann sucht `minFalse(D[],K[])` nach dem Minimum aus `D[2]=9`, `D[4]=17` und `D[5]=9`. Das ist dann 9.
Da die Orte 2 und 5 gleichauf liegen (also beide mit 9), gibt `minFalse(D[],K[])` die kleinste Zahl zurück, also 2.

Ich hoffe, du hast meine Erklärungen gut geschlumpft! Entschuldigung ich meine « verstanden » !

Hier ist der Algorithmus vom ComputerSchlumpf :

Input: `n`, die Menge der Orte; `G[][]`, die Karte der Wege.
Output: `D[]`, sodass `D[m]` die Dauer von `0` zu `m` in diesem Zyklus ist.

```
for (i ← 0 to n-1 step 1) {
     D[i] ← +∞
     K[i] ← false
}
D[0] ← 0
m ← minFalse(D[],K[])

while(m ≠ -1) {
    K[m] ← true
    for(s ← 0 to n-1 step 1) {
         if ((not K[s]) and (G[m][s] ≠ +∞)) {
              if (D[m] + G[m][s] < D[s]) {
                   D[s] ← D[m] + G[m][s]
              }
          }
    }
    m ← minFalse(D[],K[])
}
return D[]
```

Hier sind die ersten Schritte zur Ausführung des Algorithmus anhand vom Plan des Geographen Schlumpfs.
Hier die Tabellen `D[]` und `K[]`. Wir kürzen **`true`** durch T und **`false`** durch F.

- Anfangswerte:

	0	1	2	3	4	5
D:	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

	0	1	2	3	4	5
K:	F	F	F	F	F	F

- In der ersten Schleifenrunde wählt der Algorithmus `m`= 0 und überprüft seine Nachfolger 1 und 4.
  Wir haben `G[0][1]`= 4 und `G[0][4]`= 1.
  Da `D[0]+G[0][1]` = 0+4 = 4 < $+\infty$, aktualisieren wir `D[1]` mit diesem Wert.
  Dazu wird `D[4]` aktualisiert und wir erhalten:

	0	1	2	3	4	5
D:	0	4	$+\infty$	$+\infty$	1	$+\infty$

	0	1	2	3	4	5
K:	T	F	F	F	F	F

Was sind die nächsten Schritte, die der Algorithmus berechnen wird?

Q5(e) /4	Gib die Tabellen `D[]` und `K[]` am Ende der zweiten Iteration der `while` Schleife.
	Lösung: `D[] = [0,4,+∞,3,1,+∞] K[] = [T,F,F,F,T,F]`

Q5(f) /4	Gib die Tabellen `D[]` und `K[]` am Ende der dritten Iteration der `while` Schleife.
	Lösung: `D[] = [0,4,10,3,1,+∞] K[] = [T,F,F,T,T,F]`

Der Computer Schlumpf stellt sich nun Fragen über die Effektivität seines Algorithmus, der Berechnung der schnellsten Routen und über die Antworten, die dieser zurückschicken könnte…

Q5(g) /2	Wie viele Iterationen der `while` Schleife wird der Algorithmus mit der Karte des Geographen Schlumpfs ausführen?
Lösung: 6	

Dann zeigt er Papa Schlumpf seinen Algorithmus der darauf sagt: « Das ist sehr gut, mein kleiner Schlumpf, aber dein Algorithmus ist nicht sehr hilfreich, wenn er nur die *Dauer* der besten Routen berechnet! Ich will auch wissen, welche Routen es gibt! ».

Der Computer-Schlumpf macht sich also wieder an die Arbeit und merkt, dass er für jeden Ort `x` den Ort `y` berechnen muss, der direkt vor `x` kommt. Er nennt `y` den *Vorgänger* von `x`.
Die schnellste Route nach 3 ist zum Beispiel 0, 4, 3.
Der Vorgänger von 3 ist 4 und der Vorgänger von 4 ist 0.
Der Computer Schlumpf ändert seinen Algorithmus, indem er eine Tabelle `P[]` hinzufügt, die dazu dient, für jeden Ort `x` seinen Vorgänger `P[x]` festzuhalten.
Anfangs, muss `P[x]` einem speziellen Wert entsprechen, um anzuzeigen, dass noch nichts berechnet wurde.
Der Computer-Schlumpf wählt als besonderen Wert -1.
Anschließend ändert er die **`while`** Schleife um `P[x]` zu aktualisieren.
Diese Anweisungen wurden mit einem grauen Hintergrund hinzugefügt.

```
for (i ← 0 to n-1 step 1) {
     D[i] ← +∞
     K[i] ← false
     P[i] ← -1
}
D[0] ← 0
m ← minFalse(D[],K[])
while(m ≠ -1) {
    K[m] ← true
    for(s ← 0 to n-1 step 1) {
         if ((not K[s]) and (G[m][s] ≠ +∞)) {
              if (D[m] + G[m][s] < D[s]) {
                   D[s] ← D[m] + G[m][s]
                   P[s] ← m
              }
         }
    }
    m ← minFalse(D[],K[])
}
```

Du sollst vorhersagen, wie sich diese Zeilen auswirken werden.

Q5(h) /2	Was ist der Inhalt der Tabelle `P[]` am Ende der Ausführung des Algorithmus auf der Karte des GeografieSchlumpfs?
Lösung: `P[] = [-1, 0, 1, 4, 0, 2]`	