

be-OI 2025

Finale - JUNIOR
 Samstag 22. März
 2025

**Füllt diesen Rahmen bitte in GROSSBUCHSTABEN und
 LESERLICH aus**

O**Reserviert**

VORNAME :
 NAME :
 SCHULE :

Belgische Informatik-Olympiade 2025 (Dauer: maximal 2 Stunden)

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

- 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**.
- Gib deinen Namen, Vornamen und deine Schule **nur auf dieser Seite** an.
- Die Antworten** sind auf den dafür vorgesehenen Antwortbogen einzutragen. Schreibe **gut lesbar** mit einem **blauen oder schwarzen Stift oder Kugelschreiber**.
- Verwende einen Bleistift und einen Radiergummi, wenn du am Entwurf in den Frageblättern arbeitest.
- Du darfst nur Schreibmaterial dabei haben; Taschenrechner, Mobiltelefone, ... sind **verboten**.
- Du kannst jederzeit weitere Kladdeblätter bei einer Aufsichtsperson fragen.
- 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.
- Alle Codeauszüge aus der Anweisung sind in **Pseudo-Code**. Auf den folgenden Seiten findest du eine **Beschreibung** des Pseudo-Code, den wir verwenden.
- 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:



©2025 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 (+, −, × 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 ein erstes Beispiel für einen Code, in dem die Variable *alter* den Wert 17 erhält.

```
geburtsjahr ← 2008
alter ← 2025 − 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 ( $i \leftarrow 0$  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 ( $n \geq 10$ )
{
    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 ( $i \leftarrow 0$  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 – Alkuin und sein Boot

Der Mönch Alkuin von York besitzt eine Maus, eine Katze und ein großes Stück Käse. Er muss alles über den Fluss bringen, aber sein Boot ist zu klein, um mehr als einen Passagier gleichzeitig zu transportieren: entweder die Katze, die Maus oder den Käse. Leider wird die Maus den Käse fressen, wenn sie allein am Ufer gelassen wird (Alkuin kann die Maus überwachen, wenn er am Ufer ist). Ebenso wird die Katze die Maus fressen, wenn sie allein gelassen wird. Diese beiden Situationen werden als Konflikte bezeichnet. Es gibt keinen Konflikt zwischen der Katze und dem Käse; sie können ohne Aufsicht zusammen bleiben.

Alkuin möchte die Katze, die Maus, den Käse und sich selbst vom Ufer A zum Ufer B des Flusses bringen. Er muss zwischen den beiden Ufern hin und her fahren, beginnend am Ufer A. Er erkundet mehrere Szenarien. Hier ist das erste Szenario, bei dem er die Richtung jedes Schrittes und die Ladung seines Bootes angibt: die Katze () , die Maus () , den Käse () oder nichts (nichts).




Szenario 1

1. Ufer A → Ufer B: 
2. Ufer B → Ufer A: nichts
3. Ufer A → Ufer B: 
4. Ufer B → Ufer A: nichts
5. Ufer A → Ufer B: 

Um zu überprüfen, ob sein Szenario gültig ist, füllt Alkuin eine Tabelle aus, die angibt, auf welchem Ufer (A oder B) sich Alkuin, die Katze, die Maus und der Käse bei jedem Schritt befinden. Die Tabelle beginnt bei Schritt '0', was die Ausgangssituation (vor dem ersten Transport) darstellt.

Die ersten beiden Zeilen der Tabelle sind bereits ausgefüllt. Du musst sie vervollständigen und in der letzten Spalte ein Kreuz setzen, wenn es beim entsprechenden Schritt einen Konflikt gibt.

Q1(a) /5 **Vervollständige Alkuins Tabelle für Szenario 1.**

| | Alkuin |  |  |  | Konflikt? |
|--------------------|--------|---|---|---|-----------|
| Schritt 0 : | A | A | A | A | |
| Schritt 1 : | B | A | B | A | |
| Schritt 2 : | A | A | B | A | |
| Schritt 3 : | B | A | B | B | |
| Schritt 4 : | A | A | B | B | X |
| Schritt 5 : | B | B | B | B | |

Alkuin überlegt sich dann ein neues Szenario, bei dem du die Tabelle erneut vervollständigen sollst.

Szenario 2

1. Ufer A → Ufer B:
2. Ufer B → Ufer A: nichts
3. Ufer A → Ufer B:
4. Ufer B → Ufer A:
5. Ufer A → Ufer B:
6. Ufer B → Ufer A: nichts
7. Ufer A → Ufer B:

Q1(b) /7 Vervollständige Alkuins Tabelle für Szenario 2.

| | Alkuin | | | | Konflikt? |
|--------------------|--------|---|---|---|-----------|
| Schritt 0 : | A | A | A | A | |
| Schritt 1 : | B | A | B | A | |
| Schritt 2 : | A | A | B | A | |
| Schritt 3 : | B | A | B | B | |
| Schritt 4 : | A | A | A | B | |
| Schritt 5 : | B | B | A | B | |
| Schritt 6 : | A | B | A | B | |
| Schritt 7 : | B | B | B | B | |

Alkuin erbt ein größeres Boot, das ihm erlaubt, zwei zusätzliche Passagiere außer sich selbst zu transportieren. Dieses Boot hat die Größe 2 (vorher hatte er ein Boot der Größe 1).

Q1(c) /3 Gib ein Szenario in drei Schritten an, das es ermöglicht, die Katze, die Maus und den Käse vom Ufer A zum Ufer B zu bringen, mit einem Boot der Größe 2. Für jeden Schritt musst du 0, 1 oder 2 Passagiere benennen, die Alkuin transportiert.


1. Ufer A → Ufer B :
2. Ufer B → Ufer A : nichts
3. Ufer A → Ufer B : ,

Es gibt mehrere alternativen Lösungen.

Angenommen, die Katze entscheidet sich, Käse zu mögen: es gibt einen **neuen Konflikt**, man kann die Katze und den Käse nicht mehr unbeaufsichtigt zusammen lassen.

| | |
|-----------------|---|
| Q1(d) /3 | Gib ein Szenario in drei Schritten mit einem Boot der Größe 2 an. Du musst den neuen Konflikt zwischen der Katze und dem Käse berücksichtigen. Die Maus muss im ersten Schritt transportiert werden. Für jeden Schritt musst du 0, 1 oder 2 Passagiere benennen, die Alkuin transportiert. |
|-----------------|---|

1. Ufer A → Ufer B : , 

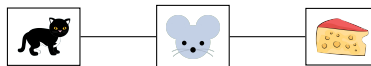
2. Ufer B → Ufer A : 

3. Ufer A → Ufer B : , 

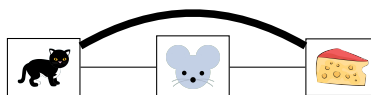
Es gibt mehrere alternativen Lösungen.

Aufgrund seiner Erfahrung beschließt Alkuin, das Klosterleben zu verlassen und ein Unternehmen zu gründen, das sich auf komplizierte Flussüberquerungen spezialisiert. Seine Kunden übergeben ihm n Passagiere (Objekte oder Tiere), die den Fluss überqueren müssen, sowie ein Diagramm, das die Konflikte zwischen ihnen anzeigt. Dieses Diagramm wird als 'Graph' bezeichnet und besteht aus beschrifteten Rechtecken, die 'Knoten' genannt werden, eines für jeden Passagier. Es enthält auch Verbindungen zwischen den Knoten, die 'Kanten' genannt werden: man zeichnet eine Kante zwischen zwei Knoten, wenn die beiden entsprechenden Passagiere in Konflikt stehen. Wenn zwei Passagiere jedoch unbeaufsichtigt zusammenbleiben können, wird keine Kante zwischen den entsprechenden Knoten gezeichnet.

Hier ist der Graph der Ausgangssituation, wo die Katze noch keinen Käse mag. Es gab einen Konflikt zwischen der Maus und der Katze, einen Konflikt zwischen der Maus und dem Käse, aber keinen Konflikt zwischen der Katze und dem Käse.



| | |
|-----------------|---|
| Q1(e) /1 | Ändere diesen Graphen (indem du Kanten zeichnest oder durchstreichst), um einen Konflikt zwischen der Katze und dem Käse hinzuzufügen. |
|-----------------|---|

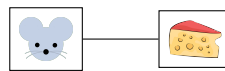


Alkuin möchte nun wissen, wie groß das Boot sein muss, um eine Gruppe von Passagieren zu transportieren, deren Konfliktgraph gegeben ist. Natürlich ist für den Transport von n Passagieren ein Boot der Größe n immer ausreichend, aber Alkuin möchte das kleinstmögliche Boot verwenden.

Er überlegt sich folgendes:

“ Ich muss die Passagiere in zwei Gruppen aufteilen. Die Passagiere der ersten Gruppe müssen ständig im Boot überwacht werden. Die anderen können unbeaufsichtigt bleiben, da es zwischen ihnen keinen Konflikt gibt. Die zu überwachenden Passagiere müssen so ausgewählt werden, dass, wenn ich alle Knoten, die ihnen entsprechen und die sie berührenden Kanten aus dem Graphen entferne, keine Kanten mehr im Graphen verbleiben.”

Zum Beispiel ist das Entfernen des Knoten 'Katze' keine gute Lösung, da es immer noch einen Konflikt zwischen der Maus und dem Käse gibt.



Mit anderen Worten, man kann die Katze nicht allein im Boot lassen und die Maus mit dem Käse am Ufer lassen.

Als die Katze jedoch noch keinen Käse mochte, führte das Entfernen der Maus zu folgendem Graphen, der keinen Konflikt enthält:



Mit anderen Worten, man konnte (als die Katze noch keinen Käse mochte) die Maus allein ins Boot laden und die Katze und den Käse am Ufer lassen.

Was Alkuin berechnen möchte, nennt man eine *Überdeckung*: **eine Menge von Knoten, so dass, wenn man diese Knoten und die sie berührenden Kanten aus dem Graphen entfernt, keine Kanten mehr übrig bleiben.**

Hier sind sechs Graphen, von denen einige Knoten grau eingefärbt sind:

Graph 1:



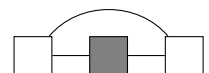
Graph 2:



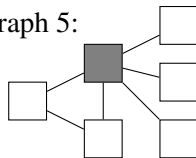
Graph 3:



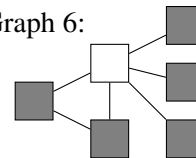
Graph 4:



Graph 5:



Graph 6:



| | Ja | Nein | Gib an, ob die grauen Knoten in jedem Graphen eine Überdeckung bilden. |
|-----------------|-------------------------------------|-------------------------------------|--|
| Q1(f) /1 | <input checked="" type="checkbox"/> | <input type="checkbox"/> | Die grauen Knoten im Graphen 1 bilden eine Überdeckung. |
| Q1(g) /1 | <input checked="" type="checkbox"/> | <input type="checkbox"/> | Die grauen Knoten im Graphen 2 bilden eine Überdeckung. |
| Q1(h) /1 | <input type="checkbox"/> | <input checked="" type="checkbox"/> | Die grauen Knoten im Graphen 3 bilden eine Überdeckung. |
| Q1(i) /1 | <input type="checkbox"/> | <input checked="" type="checkbox"/> | Die grauen Knoten im Graphen 4 bilden eine Überdeckung. |
| Q1(j) /1 | <input type="checkbox"/> | <input checked="" type="checkbox"/> | Die grauen Knoten im Graphen 5 bilden eine Überdeckung. |
| Q1(k) /1 | <input checked="" type="checkbox"/> | <input type="checkbox"/> | Die grauen Knoten im Graphen 6 bilden eine Überdeckung. |

| | |
|-----------------|---|
| Q1(l) /4 | Färbe die Knoten des untenstehenden Graphen. Die gefärbten Knoten müssen eine Überdeckung bilden, die eine minimale Anzahl von Knoten enthält. |
|-----------------|---|



Alkuin möchte einen Algorithmus haben, der eine Überdeckung eines Graphen berechnet. Nach langen Diskussionen mit der Katze findet er die folgende Strategie (die nicht unbedingt sehr intelligent ist, aber es ist die Idee der Katze). Die Knoten sind von 1 bis n nummeriert und werden im Algorithmus durch ihre Nummer dargestellt. Wenn es eine Kante zwischen den Knoten mit den Nummern i und j gibt, wird sie als (i, j) bezeichnet. Der Algorithmus betrachtet nacheinander alle Kanten (i, j) des Graphen und *behandelt* deren Enden, das heißt die Knoten mit den Nummern i und j . Behandeln eines Knoten besteht darin:

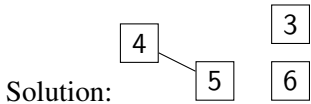
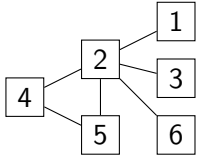
1. den Knoten in die Überdeckung aufzunehmen,
2. den Knoten und alle Kanten, von denen er ein Ende ist, aus dem Graphen zu entfernen.

Man fährt fort, bis keine Kante mehr im Graphen verbleibt.

Alkuin bemerkt, dass die Kanten keine Richtung haben, zum Beispiel ist die Kante $(3, 1)$ dieselbe wie $(1, 3)$. Er kann sich daher darauf beschränken, die Kanten (i, j) zu betrachten, bei denen $i < j$ (um zu vermeiden, dass dieselbe Kante zweimal behandelt wird).

Alkuin beschließt, die Kanten in der folgenden Reihenfolge zu behandeln (falls sie existieren): zuerst die Kanten $(1, 2), (1, 3), \dots (1, n)$; dann die Kanten $(2, 3), (2, 4), \dots (2, n)$; dann die Kanten $(3, 4), (3, 5), \dots (3, n)$; und so weiter bis zur letzten Kante $(n - 1, n)$.

| | |
|-----------------|--|
| Q1(m) /3 | Zeichne den Graphen, der zu Beginn des Algorithmus erhalten wird, nachdem nur die erste Kante $(1, 2)$ betrachtet wurde und somit die Knoten 1 und 2 behandelt wurden. |
|-----------------|--|



| | |
|-----------------|--|
| Q1(n) /4 | Welche Überdeckung wird nach vollständiger Ausführung des Algorithmus auf dem Graphen der vorherigen Frage berechnet? Du musst die Liste der Knoten in der berechneten Überdeckung angeben. |
|-----------------|--|

Solution : $\{1, 2, 4, 5\}$

Q1(o) /4 Welche Überdeckung wird vom Algorithmus auf dem untenstehenden Graphen berechnet? Du musst die Liste der Knoten in der berechneten Überdeckung angeben.

Solution : {1, 2, 3, 5}

Um den Algorithmus zu implementieren, verwendet Alkuin eine Matrix G der Größe $n \times n$, um die Kanten zu speichern. Da die Kanten keine Richtung haben, betrachtet er nur die Kanten (i, j) mit $i < j$. Die Matrix enthält **true** in der Zelle $G[i][j]$ (mit $i < j$) genau dann wenn es eine Kante (i, j) im Graphen gibt. Alle anderen Zellen der Matrix enthalten **false**.

Q1(p) /4 Erstelle die Matrix G , die dem untenstehenden Graphen entspricht, wobei die Zelle $G[i][j]$ sich in der Zeile i , Spalte j befindet. Schreibe ein T in die Zellen, die true enthalten.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | T | | T | | |
| 2 | | | T | T | | |
| 3 | | | | | T | T |
| 4 | | | | | T | |
| 5 | | | | | | |
| 6 | | | | | | |

Alkuin hat seinen Algorithmus auf ein Stück Papier geschrieben, aber leider hat die Maus einige Teile davon angeknabbert. Alkuin braucht deine Hilfe, um die fehlenden Teile wiederherzustellen.

Der Algorithmus beginnt mit zwei Funktionen:

- `InitCountEdges` zählt die Anzahl der Kanten im Graphen und speichert diese Zahl in der Variablen `countEdges`,
- `ProcessNode(i)` verarbeitet den Knoten `i` (wie oben beschrieben).

Zusätzlich zur Matrix `G[][]`, wird eine Boolesche Matrix `InCover[]` der Größe `n` verwendet, in dem alle Elemente zunächst auf `false` gesetzt sind. Der Algorithmus signalisiert, dass der Knoten `i` Teil der Überdeckung ist, indem er den Wert von `InCover[i]` auf `true` setzt.

Q1(q) /4 Vervollständige die `_____` in der Funktion `InitCountEdges` (Antworten so kurz wie möglich).

```
function InitCountEdges()
{
  CountEdges = 0
  for (i ← 1 to n-1 step 1)
  {
    for (j ← i+1 to n step 1)
    {
      if (G[i][j]) CountEdges ← CountEdges + 1
    }
  }
}
```

Q1(r) /6 Vervollständige die `_____` in der Funktion `ProcessNode` (Antworten so kurz wie möglich).

```
function ProcessNode(i)
{
  InCover[i] ← true
  for (k ← 1 to i-1 step 1)
  {
    if (G[k][i])
    {
      G[k][i] ← false
      CountEdges ← CountEdges - 1
    }
  }
  for (k ← i+1 to n step 1)
  {
    if (G[i][k])
    {
      G[i][k] ← false
      CountEdges ← CountEdges - 1
    }
  }
}
```

Der Algorithmus wird durch die Funktion `Cover` vervollständigt.

Diese Funktion erhält eine Matrix `G [] []` der Größe $n \times n$ das die vorhandenen Kanten zwischen den n Knoten darstellt. Die Matrix `InCover []` (der Größe n) wird wie oben erklärt vor der Ausführung der Funktion `Cover` auf **false** initialisiert.

Die berechnete Überdeckung besteht aus allen Knoten i , bei denen `InCover [i]` nach der Ausführung der Funktion `Cover` **true** ist.

Q1(s) /6 Vervollständige die in der Funktion `Cover` (Antworten so kurz wie möglich).

```
function Cover ()
{
  InitCountEdges ()
  i ← 1
  j ← 2
  while (  )
  {
    if (G[i][j])
    {
      ProcessNode (i)
      ProcessNode (j)
    }
    j ← j+1
    if (j = n+1) {
      i ← 
      j ← 
    }
  }
}
```

Frage 2 – Seeschlacht

Wir wollen das Spiel “Seeschlacht” programmieren, das auf einem 10×10 -Gitter gespielt wird, auf dem Schiffe platziert werden.

- Es gibt 10 Reihen, die von oben nach unten von 1 bis 10 nummeriert sind.
- Es gibt 10 Spalten, die von links nach rechts von 1 bis 10 nummeriert sind.
- Ein horizontales Schiff besetzt aufeinanderfolgende Felder in derselben Reihe.
- Ein vertikales Schiff besetzt aufeinanderfolgende Felder in derselben Spalte.
- Wir bezeichnen mit (r, c) die Koordinaten des Feldes an der Kreuzung der Reihe r und der Spalte c .

| | | | | | | | | | | |
|----|---|---|---|---|---|---|---|---|---|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | | | | | | | | | | |
| 2 | | | A | A | A | A | | | | |
| 3 | | | | | | | | | | |
| 4 | | | | | | | | | | |
| 5 | | | | | | | B | | | |
| 6 | | | | | | | B | | | |
| 7 | | | | | | | B | | | |
| 8 | | | | C | | | | | | |
| 9 | | | | | | | | | | |
| 10 | | | | | | | | | | |

Beispiel:

- 3 Schiffe **A**, **B** und **C** sind in Grau auf dem Bild dargestellt.
- **A** ist horizontal, **B** ist vertikal, **C** ist sowohl horizontal als auch vertikal.
- **B** besetzt die Felder mit den Koordinaten $(5, 7)$, $(6, 7)$ und $(7, 7)$.

Wir schwanken zwischen zwei Systemen, um die Schiffe in unserem Programm darzustellen. Jedes System verwendet vier Attribute, um die Position eines Schiffes auf dem Gitter anzugeben.

Attribute im System **Orientierung**.

- Die Reihe r des oberen linken Feldes des Schiffes.
- Die Spalte c des oberen linken Feldes des Schiffes.
- Die Länge L des Schiffes.
- Die Richtung hor des Schiffes:
true, wenn es horizontal ist, **false**, wenn es vertikal ist.

Orientierung-Attribute der Schiffe auf dem Bild.

| Schiff | r | c | L | hor |
|----------|---|---|---|---|
| A | 2 | 3 | 4 | true |
| B | 5 | 7 | 3 | false |
| C | 8 | 4 | 1 | true oder false nach Wahl |

Attribute im System **Rechteck**.

- Die Reihe $r1$ des oberen linken Feldes des Schiffes.
- Die Spalte $c1$ des oberen linken Feldes des Schiffes.
- Die Reihe $r2$ des unteren rechten Feldes des Schiffes.
- Die Spalte $c2$ des unteren rechten Feldes des Schiffes.

Rechteck-Attribute der Schiffe auf dem Bild.

| Schiff | r1 | c1 | r2 | c2 |
|----------|----|----|----|----|
| A | 2 | 3 | 2 | 6 |
| B | 5 | 7 | 7 | 7 |
| C | 8 | 4 | 8 | 4 |

Wir werden einige Funktionen in beiden Systemen schreiben, um zu entscheiden, welches wir für unser Programm wählen.

Konversionen zwischen den 2 Systemen

Lassen Sie uns zunächst üben, von einem System zum anderen zu wechseln.

Orientierung → **Rechteck**

Ein Boot wird durch seine 4 Attribute (r, c, L, hor) im **Orientierungs**-System beschrieben.

Geben Sie seine Attribute $(r1, c1, r2, c2)$ im **Rechteck**-System an.

| | |
|----------|---|
| Q2(a) /1 | $(4, 5, 2, \text{True}) \rightarrow (4, 5, 4, 6)$ |
|----------|---|

| | |
|----------|--|
| Q2(b) /1 | $(3, 6, 3, \text{False}) \rightarrow (3, 6, 5, 6)$ |
|----------|--|

| | |
|----------|---|
| Q2(c) /1 | $(7, 5, 1, \text{True}) \rightarrow (7, 5, 7, 5)$ |
|----------|---|

| | |
|----------|--|
| Q2(d) /1 | $(1, 9, 5, \text{False}) \rightarrow (1, 9, 5, 9)$ |
|----------|--|

Rechteck → **Orientierung**

Ein Boot wird durch seine 4 Attribute (r_1, c_1, r_2, c_2) im **Rechteck**-System beschrieben.

Geben Sie seine Attribute (r, c, L, hor) im **Orientierungs**-System an.

| | |
|----------|--|
| Q2(e) /1 | $(2, 7, 5, 7) \rightarrow (2, 7, 4, \text{False})$ |
|----------|--|

| | |
|----------|---|
| Q2(f) /1 | $(4, 3, 4, 3) \rightarrow (4, 3, 1, \text{True})$ |
|----------|---|

| | |
|----------|---|
| Q2(g) /1 | $(8, 3, 8, 6) \rightarrow (8, 3, 4, \text{True})$ |
|----------|---|

| | |
|----------|--|
| Q2(h) /1 | $(1, 4, 1, 10) \rightarrow (1, 4, 7, \text{True})$ |
|----------|--|

Erstellen wir Funktionen, um diese Konversionen zu automatisieren.

Funktion O2R

Die Funktion `O2R` konvertiert Attribute des **Orientierungs**-Systems in äquivalente Attribute des **Rechteck**-Systems.

Zum Beispiel gibt `O2R(2, 3, 4, true)` das Ergebnis $(2, 3, 2, 6)$ zurück, wenn man das Boot **A** aus dem Bild verwendet.

| | |
|----------|---|
| Q2(i) /5 | Vervollständigen Sie die _____ in der Funktion <code>O2R</code> . |
|----------|---|

```
function O2R(r, c, L, hor)
{
  if (hor)
    { return (r, c, r, c+L-1) }
  else
    { return (r, c, r+L-1, c) }
}
```

Funktion R2O

Die Funktion `R2O` konvertiert Attribute des **Rechteck**-Systems in äquivalente Attribute des **Orientierungs**-Systems.

Eine logische Ausdruck (siehe nächste Seite) muss verwendet werden, um den Wert von `hor` zu berechnen.

Zum Beispiel gibt `R2O(2, 3, 2, 6)` das Ergebnis $(2, 3, 4, \text{true})$ zurück, wenn man das Boot **A** aus dem Bild verwendet.

| | |
|----------|---|
| Q2(j) /5 | Vervollständigen Sie die _____ in der Funktion <code>R2O</code> . |
|----------|---|

```
function R2O(r1, c1, r2, c2)
{
  L ← (r2-r1) + (c2-c1) + 1

  hor ← (r1==r2)

  return (r1, c1, L, hor)
}
```

Gültigkeitstests

Das Programm muss überprüfen, ob Attribute ein Boot beschreiben, das in das 10x10 Gitter platziert werden kann. Dies geschieht durch die Auswertung von logischen Ausdrücken, deren Ergebnis entweder **true** oder **false** ist. Diese Art von Ausdruck verwendet die Vergleichsoperatoren (`==`, `!=`, `>`, `>=`, `<`, `<=`) und die logischen Operatoren (**and**, **or**, **not**), die in den folgenden Beispielen dargestellt werden.

| Logischer Ausdruck | Wert |
|------------------------------------|--|
| <code>c==4</code> | true , wenn <code>c</code> gleich 4 ist, false sonst. |
| <code>r!=5</code> | true , wenn <code>r</code> ungleich 5 ist. |
| <code>r1>3</code> | true , wenn <code>r1</code> größer als 3 ist. |
| <code>c2>=c1</code> | true , wenn <code>c2</code> größer oder gleich <code>c1</code> ist. |
| <code>r+L<9</code> | true , wenn <code>r+L</code> kleiner als 9 ist. |
| <code>c2<=c1+2</code> | true , wenn <code>c2</code> kleiner oder gleich <code>c1+2</code> ist. |
| <code>(c<2) and (r>3)</code> | true , wenn beide Ungleichungen wahr sind. |
| <code>(c1==3) or (r1==2)</code> | true , wenn mindestens eine der beiden Gleichungen wahr ist. |
| <code>not (hor)</code> | true , wenn <code>hor</code> gleich false ist (und false , wenn <code>hor</code> gleich true ist). |

Funktion Rok

Im **Rechteck**-System müssen die Felder `(r1, c1)` und `(r2, c2)` in derselben Reihe oder in derselben Spalte liegen, alle Koordinaten müssen zwischen 1 und 10 liegen und das erste Feld muss über oder links vom zweiten Feld sein. Vervollständigen Sie die Funktion `Rok`, die **true** zurückgibt, wenn die Attribute `(r1, c1, r2, c2)` all diese Bedingungen erfüllen. Da der Ausdruck sehr lang ist, wird er in mehreren Schritten ausgewertet, wobei die logische Variable `ok` verwendet wird.

Q2(k) /5 Vervollständigen Sie die _____ in der Funktion `Rok`.

```
function Rok(r1, c1, r2, c2)
{
  ok ← (r1==r2) or (c1==c2)
  ok ← ok and (1<=r1 and r1<=r2 and r2<=10)
  ok ← ok and (1<=c1 and c1<=c2 and c2<=10)
  return ok
}
```

Funktion Ook

Im **Orientierungs**-System muss die Länge überprüft werden, die größer oder gleich 1 sein muss. Vervollständigen Sie die Funktion `Ook`, die **true** zurückgibt, wenn das Boot `(r, c, L, hor)` auf dem 10x10 Gitter platziert werden kann.

Q2(l) /5 Vervollständigen Sie die _____ in der Funktion `Ook`.

```
function Ook(r, c, L, hor)
{
  ok ← (1<=L and 1<=r and 1<=c)
  if (hor)
  { return (ok and r<=10 and c+L-1<=10) }
  else
  { return (ok and r+L-1<=10 and c<=10) }
}
```

Schiff getroffen

Das Programm muss überprüfen, ob ein Boot getroffen wurde, wenn auf ein Feld mit den angegebenen Koordinaten geschossen wird.

Dies ist der Fall, wenn das Boot das angezielte Feld besetzt.

In den folgenden Fragen versuchen Sie, die kürzesten und einfachsten Antworten zu finden, indem Sie möglichst wenige Vergleiche und logische Operatoren verwenden.

Funktion hitR (Rechteck-System)

Die Funktion `hitR(rr, cc, r1, c1, r2, c2)` gibt **true** zurück, wenn das Boot mit den Attributen $(r1, c1, r2, c2)$ das Feld mit den Koordinaten (rr, cc) besetzt, und gibt **false** zurück, wenn dies nicht der Fall ist.

Mit der in dem Bild beschriebenen Situation:

- `hitR(2, 5, 2, 3, 2, 6)` gibt **true** zurück, weil das Boot **A** das Feld mit den Koordinaten $(2, 5)$ besetzt.
- `hitR(7, 6, 5, 7, 7, 7)` gibt **false** zurück, weil das Boot **B** das Feld mit den Koordinaten $(7, 6)$ nicht besetzt.

Q2(m) /6 Vervollständigen Sie die _____ in der Funktion `hitR` (Antwort so kurz wie möglich).

```
function hitR(rr, cc, r1, c1, r2, c2)
{
  return (r1<=rr) and (rr<=r2) and (c1<=cc) and (cc<=c2)
}
```

Funktion hitO (Orientierungs-System)

Die Funktion `hitO(rr, cc, r, c, L, hor)` gibt **true** zurück, wenn das Boot mit den Attributen (r, c, L, hor) das Feld mit den Koordinaten (rr, cc) besetzt, und gibt **false** zurück, wenn dies nicht der Fall ist.

Mit der in dem Bild beschriebenen Situation:

- `hitO(2, 5, 2, 3, 4, true)` gibt **true** zurück, weil das Boot **A** das Feld mit den Koordinaten $(2, 5)$ besetzt.
- `hitO(7, 6, 5, 7, 3, false)` gibt **false** zurück, weil das Boot **B** das Feld mit den Koordinaten $(7, 6)$ nicht besetzt.

Q2(n) /6 Vervollständigen Sie die _____ in der Funktion `hitO` (Antworten so kurz wie möglich).

```
function hitO(rr, cc, r, c, L, hor)
{
  if (hor)
  { return (rr==r) and (c<=cc) and (cc<c+L) }
  else
  { return (cc==c) and (r<=rr) and (rr<r+L) }
}
```

Kollision

Zwei Boote können nicht dasselbe Feld im Gitter besetzen, andernfalls kommt es zu einer Kollision.

Funktion collisionR (Rechteck-System)

Die Funktion `collisionR` gibt **true** zurück, wenn zwei Boote kollidieren, und **false**, wenn dies nicht der Fall ist.

Die Attribute der Boote sind $(r1A, c1A, r2A, c2A)$ und $(r1B, c1B, r2B, c2B)$.

Verwenden Sie die kürzesten logischen Ausdrücke als möglich. Es ist dennoch sehr lang, und der Ausdruck wird in mehreren Schritten mit den logischen Variablen `rbool` und `cbool` ausgewertet.

Q2(o) /7 Vervollständigen Sie die _____ in der Funktion `collisionR` (Antworten so kurz wie möglich).

```
function collisionR(r1A, c1A, r2A, c2A, r1B, c1B, r2B, c2B)
{
  rbool ← (r1A<=r1B and r1B<=r2A) or (r1B<=r1A and r1A<=r2B)

  cbool ← (c1A<=c1B and c1B<=c2A) or (c1B<=c1A and c1A<=c2B)
  return rbool and cbool
}
```

Funktion collisionO (Orientierungs-System)

Die Funktion `collisionO` gibt **true** zurück, wenn zwei Boote kollidieren, und **false**, wenn dies nicht der Fall ist.

Die Attribute der Boote sind $(rA, cA, LA, horA)$ und $(rB, cB, LB, horB)$.

Um noch längere logische Ausdrücke zu vermeiden, verwenden wir die Funktion `hitO`, die oben definiert wurde.

Q2(p) /7 Vervollständigen Sie die _____ in der Funktion `collisionO` unter Verwendung der Funktion `hitO`.

```
function collisionO(rA, cA, LA, horA, rB, cB, LB, horB)
{
  if (horA==horB)
  { return hitO(rB, cB, rA, cA, LA, horA) or hitO(rA, cA, rB, cB, LB, horB) }
  else
  {
    if (horA)
    { return hitO(rA, cB, rA, cA, LA, horA) and hitO(rA, cB, rB, cB, LB, horB) }
    else
    { return hitO(rB, cA, rA, cA, LA, horA) and hitO(rB, cA, rB, cB, LB, horB) }
  }
}
```


Kollisionsrisiko

Die Boote dürfen nicht zu nah beieinander stehen.

Genauer gesagt dürfen zwei Felder, die von zwei verschiedenen Booten besetzt sind, keine Seite oder Ecke gemeinsam haben.

Zum Beispiel sind alle Boote in dem Bild unten zu nahe beieinander.

A und **B** berühren sich, **C** und **D** berühren sich, **E** und **F** berühren sich an einer Ecke.

| | | | | | | | | | | |
|----|---|---|---|---|---|---|---|---|---|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | | | | | | | | C | | |
| 2 | | A | A | A | A | | | C | | |
| 3 | | | B | | | | | C | | |
| 4 | | | B | | | | | C | D | |
| 5 | | | | | | | | | D | |
| 6 | | | | | | | | | | |
| 7 | | | | | | | | | | |
| 8 | | E | E | E | E | | | | | |
| 9 | | | | | | F | F | F | | |
| 10 | | | | | | | | | | |

Funktion riskR (Rechteck-System)

Die Funktion `riskR` gibt **true** zurück, wenn zwei Boote zu nahe beieinander sind, und **false**, wenn dies nicht der Fall ist.

Die Attribute der Boote sind $(r1A, c1A, r2A, c2A)$ und $(r1B, c1B, r2B, c2B)$.

Am einfachsten ist es, die oben definierte Funktion `collisionR` zu verwenden.

| | |
|-----------------|--|
| Q2(q) /6 | Vervollständigen Sie die <code>_____</code> in der Funktion <code>riskR</code> unter Verwendung der Funktion <code>collisionR</code>. |
|-----------------|--|

```
function riskR(r1A,c1A,r2A,c2A,r1B,c1B,r2B,c2B)
{
    return collisionR( r1A,c1A,r2A,c2A, r1B-1,c1B-1,r2B+1,c2B+1 )
}
```

Funktion riskO (Orientierungs-System)

Diese Funktion ist schwieriger zu schreiben.

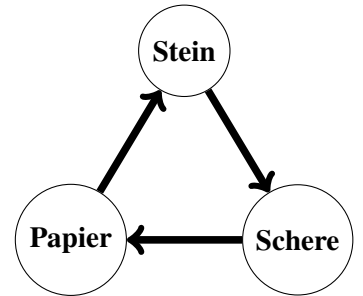
Wir bestehen nicht darauf und entscheiden uns, im Programm das Rechteck-System zu verwenden.

Frage 3 – Stein-Schere-Papier

In dem berühmten Spiel simulieren beide Spieler gleichzeitig mit der Hand eine Waffe: **Stein** oder **Schere** oder **Papier**.

Das Ergebnis des Kampfes wird mithilfe des nebenstehenden Kreisdiagramms ermittelt:

Stein schlägt **Schere**, die **Papier** schlägt, das wiederum **Stein** schlägt.
Es herrscht Gleichstand, wenn die Spieler die gleiche Waffe wählen.



Ein Spielzeughersteller will kleine Roboter auf den Markt bringen, die gegeneinander **Stein-Schere-Papier** spielen können.

Zwei Roboter, die sich in unmittelbarer Nähe zueinander befinden, kommunizieren drahtlos miteinander. Ein Spieler drückt eine Taste auf seinem Roboter, um einen Wettstreit vorzuschlagen, der andere Spieler drückt eine Taste auf seinem Roboter, um den Wettstreit anzunehmen.

In einem Wettstreit spielen die Roboter 10 Spiele **Stein-Schere-Papier**, wobei der siegreiche Roboter für jeden Sieg 1 Punkt erhält.

Spiele und Spielstand können auf den Bildschirmen der Roboter verfolgt werden.

Derjenige, der die meisten Spiele gewonnen hat, gewinnt den Wettstreit.

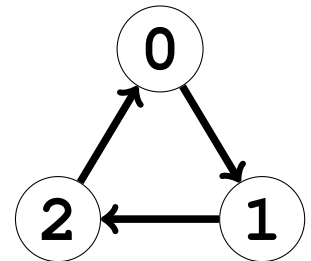
Es kann ein Unentschieden geben, wenn beide Roboter die gleiche Anzahl an Spielen gewinnen.

Roboter spielen nicht zufällig. Jeder Roboter folgt einer bestimmten **Strategie**, die in einem Programm festgelegt ist. Der Hersteller will eine große Anzahl von Robotern mit unterschiedlichen Farben, Formen und vor allem Strategien anbieten.

Kodierung

In den Programmen wird **Stein** durch 0, **Schere** durch 1 und **Papier** durch 2 kodiert. Demnach schlägt die 0 die 1 die 2 schlägt, die wiederum 0 schlägt und der Graph des Spiels wird zu dem nebenstehenden Graph.

Im Folgenden wird gesagt, dass ein Roboter den Zug 0, 1 oder 2 spielt.



Strategien

Die Strategie eines Roboters wird von zwei Elementen bestimmt.

- Die Variable `init`, die den Zug enthält, der im ersten Spiel gespielt werden soll.
- Die Tabelle `strat` mit drei Reihen und drei Spalten, die dazu dient, die Züge für andere Spiele zu bestimmen.
Wenn der Roboter `r` und sein Gegner `s` gespielt haben, wird er im nächsten Spiel `strat[r][s]` spielen.

Ein Beispiel für eine Tabelle `strat` befindet sich auf der rechten Seite.

Ihre Zeilen- und Spaltennummern beginnen bei 0 und sind in grau notiert.

Mit dieser Tabelle und basierend auf den Zügen des vorherigen Spiels muss der Roboter :

- `strat[1][2]=0` wenn der Roboter 1 und sein Gegner 2 gespielt haben.
- `strat[2][0]=2` wenn der Roboter 2 und sein Gegner 0 gespielt haben.
- 1 wenn es ein Unentschieden gegeben hat,
weil `strat[0][0]=strat[1][1]=strat[2][2]=1`.

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 2 | 0 |
| 1 | 2 | 1 | 0 |
| 2 | 2 | 2 | 1 |



Strategien kodieren

In den folgenden Fragen lassen wir die Variable `init` weg.

Eine Strategie wird durch einen Text beschrieben, der erklärt, was der Roboter spielen soll, basierend auf dem, was er und sein Gegner im vorherigen Spiel gespielt haben.

Du sollst die Tabelle `strat` kodieren, die der im Text beschriebenen Strategie entspricht.

Du kannst die Raster auf der nächsten Seite als Vorlage verwenden.

Vergiss nicht, **deine Lösungen auf die Antwortbögen zu kopieren.**

Q3(a) /4 Fülle die Tabelle `strat` mit der Strategie aus, die im Satz beschrieben wird.
 “Spiele vorherigen Zug des Gegners.”

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 0 | 1 | 2 |
| 1 | 0 | 1 | 2 |
| 2 | 0 | 1 | 2 |

Q3(b) /4 Fülle die Tabelle `strat` mit der Strategie aus, die im Satz beschrieben wird.
 “Spiele den gewinnenden Zug gegen den vorherigen Zug des Gegners.”

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 2 | 0 | 1 |
| 1 | 2 | 0 | 1 |
| 2 | 2 | 0 | 1 |

Q3(c) /4 Fülle die Tabelle `strat` mit der Strategie aus, die im Satz beschrieben wird.
 “Bei Gleichstand, spiele den vorherigen Zug des Roboters; ansonsten den Zug des vorherigen Gewinners.”

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 0 | 0 | 2 |
| 1 | 0 | 1 | 1 |
| 2 | 2 | 1 | 2 |

Q3(d) /4 Fülle die Tabelle `strat` mit der Strategie aus, die im Satz beschrieben wird.
 “Bei Gleichstand, spiele den gewinnenden Zug gegen den vorherigen Zug; ansonsten wiederhole den vorherigen Zug des Roboters.”

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 2 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 2 | 2 | 2 | 1 |



Im Folgenden werden die unten beschriebenen Notationen verwendet.

- r : der Zug, den der Roboter im vorherigen Spiel gemacht hat.
- s : der Zug, den der Gegner in der vorherigen Spiel gespielt hat.
- w : der Zug, den der Gewinner des vorherigen Spiels gemacht hat (oder beide Spieler bei Gleichstand).
- x : der Zug, den der Verlierer der vorherigen Partie gespielt hat (oder beide Spieler bei Gleichstand).
- $W(c)$: der Zug, der gegen den Zug c gewinnt (zum Beispiel $W(0) = 2$ da $2 > 0$ schlägt).
- $X(c)$: der Zug, der gegen den Zug c verliert (zum Beispiel $X(0) = 1$ da $0 > 1$ schlägt).

Q3(e) /4 Fülle die Tabelle $strat$ mit der Strategie aus, die im Satz beschrieben wird.
 “Bei Gleichstand, spiele $x(r)$; ansonsten spiele den nicht ausgeführten Zug.”

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 1 | 2 | 1 |
| 1 | 2 | 2 | 0 |
| 2 | 1 | 0 | 0 |

Q3(f) /4 Fülle die Tabelle $strat$ mit der Strategie aus, die im Satz beschrieben wird.
 “Wenn $s=0$, spiele r ; wenn $s=1$ und es keinen Gleichstand gab, spiele w ; wenn $s=2$ und es keinen Gleichstand gab, spiele x ; in allen anderen Fällen, spiele $W(r)$.”

| | | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 2 |
| 2 | 2 | 1 | 1 |



Eine Strategie schlagen

In den folgenden Fragen wird eine Strategie A mithilfe von `initA` und der Tabelle `stratA` vollständig angegeben. Finde eine Strategie B, die in einem Spiel gegen Strategie A mit 10:0 gewinnt.

Du musst den **ersten Zug `initB`** und ein **Minimum an Werten in der Tabelle `stratB`** angeben.

Lasse die Kästchen in `stratB` leer, die nie von deiner Strategie in einem Spiel gegen Strategie A genutzt werden.

Die Nummern der Reihen und Spalten werden nicht mehr notiert, füge sie bei Bedarf hinzu.

Q3(g) /4 Gebe eine Strategie B an, die 10 zu 0 gegen Strategie A gewinnt.
Geben Sie nur die Werte an, die in `stratB` unbedingt erforderlich sind.

`initA=0, stratA=`

| | | |
|---|---|---|
| 2 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 2 | 1 |

\rightarrow `initB=`2`,` `stratB=`

| | | |
|---|--|--|
| | | |
| | | |
| 2 | | |

Q3(h) /4 Gebe eine Strategie B an, die 10 zu 0 gegen Strategie A gewinnt.
Geben Sie nur die Werte an, die in `stratB` unbedingt erforderlich sind.

`initA=1, stratA=`

| | | |
|---|---|---|
| 2 | 0 | 1 |
| 0 | 1 | 2 |
| 1 | 2 | 0 |

\rightarrow `initB=`0`,` `stratB=`

| | | |
|---|---|--|
| | 2 | |
| | | |
| 0 | | |

Q3(i) /4 Gebe eine Strategie B an, die 10 zu 0 gegen Strategie A gewinnt.
Geben Sie nur die Werte an, die in `stratB` unbedingt erforderlich sind.

`initA=2, stratA=`

| | | |
|---|---|---|
| 2 | 0 | 1 |
| 2 | 0 | 1 |
| 2 | 0 | 1 |

\rightarrow `initB=`1`,` `stratB=`

| | | |
|---|---|---|
| | 1 | |
| | | 2 |
| 0 | | |

Q3(j) /4 Gebe eine Strategie B an, die 10 zu 0 gegen Strategie A gewinnt.
Geben Sie nur die Werte an, die in `stratB` unbedingt erforderlich sind.

`initA=1, stratA=`

| | | |
|---|---|---|
| 1 | 1 | 2 |
| 0 | 1 | 0 |
| 0 | 2 | 2 |

\rightarrow `initB=`0`,` `stratB=`

| | | |
|---|---|---|
| | 2 | |
| | | 1 |
| 1 | | |

Vergiss nicht, **deine Lösungen auf die Antwortbögen zu kopieren.**