be-OI 2022 **Finale - SENIOREN** Samstag 23. April 2022	**Füllt diesen Rahmen bitte in GROSSBUCHSTABEN und LESERLICH aus** VORNAME : .... NAME : .... SCHULE :	901 **Reserviert**

Belgische Informatik-Olympiade 2022 (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 **Kadetten**.
   - Für Schüler im dritten oder vierten Jahr der Sekundarschule: Kategorie **Junioren**.
   - Für Schüler der Sekundarstufe 5 und höher: Kategorie **Senioren**.
2. Gebe deinen Namen, Vornamen und deine Schule **nur auf der erste Seite** an.
3. Gebe **deine Antworten** auf den in diesem Dokument **am Ende des Formulars** dafür vorgesehenen Seiten an.
4. Wenn du dich bei einer Antwort vertan hast, dann beantworte sie unbedingt **auf dem selbem Blatt** (ggf. auf der Rückseite).
5. Schreibe **gut lesbar** mit einem **blauen oder schwarzen Stift oder Kugelschreiber**.
6. Du kannst nur Schreibmaterial dabei haben; Taschenrechner, Mobiltelefone, … sind **verboten**.
7. Du kannst jederzeit weitere Kladdeblätter bei einer Aufsichtsperson fragen.
8. 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.
9. Alle Codeauszüge aus der Anweisung sind in **Pseudo-Code**. Auf den folgenden Seiten findest du eine **Beschreibung** des Pseudo-Code, den wir verwenden.
10. Wenn du mit einem Code antworten musst, dann benutze den **Pseudo-Code** oder eine **Sprache 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 durch unsere Mitglieder:

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 traditionelle Operatoren ($+$, $-$, $\times$ und $/$), kann man auch den Operator $\%$ verwenden. Wenn a und b ganze Zahlen sind, dann stellt a/b und $a\%b$ den Quotienten bzw. 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 ← 2003
alter ← 2020 − geburtjahr
```

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*. Beachte 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.
}
sonst
{
    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 ein Schleifencodefor 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 um 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.

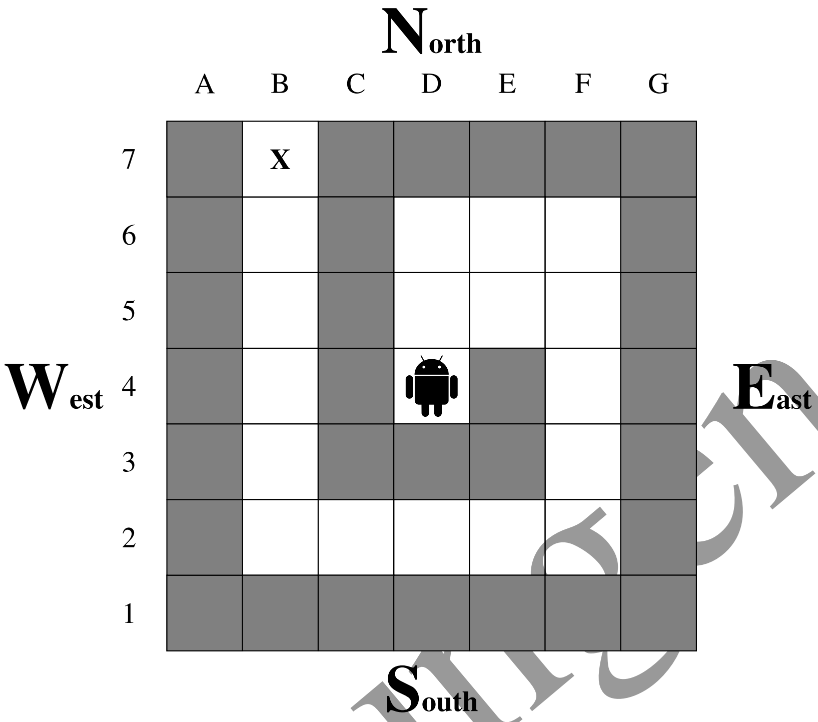
```
Eingabe: tab, ein Array mit N Zahlen.
         N, die Anzahl der Elemente des Arrays.
Ausgabe: sum, die Summe aller im Array enthaltenen Zahlen.

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 – Nono, der kleine Roboter

Nono, der kleine Roboter, steckt in der Mitte eines Labyrinths fest.



Nono kann sich auf weißen Feldern bewegen, aber nicht auf grauen Feldern, die Mauern darstellen. Zu Beginn befindet sich Nono in der Mitte des Labyrinths in dem Feld mit den Koordinaten **D4**. Er muss den Ausgang erreichen: das Feld mit den Koordinaten **B7**, das durch ein **X** gekennzeichnet ist.

Wir bewegen Nono jeweils um ein Feld mit dem Befehl `Go()`, der einen Parameter erhält, der vier verschiedene Werte haben kann: `N`, `S`, `E` oder `W`. Jeder Parameter entspricht einer der vier Himmelsrichtungen, die in der obigen Abbildung dargestellt sind, wobei Norden gemäß der üblichen Konvention oben liegt.

Auf der nächsten Seite werden mehrere Algorithmen vorgeschlagen, um Nono aus dem Labyrinth herauszuführen. Wir wollen wissen, ob Nono am Ende der Ausführung jedes Programms auf dem Feld **X** steht oder ob er mit einer Wand kollidiert (und in diesem Fall fragen wir nach den Koordinaten der Wand, mit der die Kollision stattfindet).

Einige Algorithmen verwenden Arrays. Zum Beispiel initialisiert `Dir←[N,E,S,W,N]` in Algorithmus 4 ein Array mit 5 Elementen, die `Dir[0]=N`, `Dir[1]=E`, `Dir[2]=S`, `Dir[3]=W` und `Dir[4]=N` sein werden.

Algorithmus 1

```
Go(N)
for (i ← 1 to 2 step 1)
{
  Go(E)
}
for (i ← 1 to 3 step 1)
{
  Go(S)
}
for (i ← 1 to 4 step 1)
{
  Go(W)
}
for (i ← 1 to 5 step 1)
{
  Go(N)
}
```

Algorithmus 2

```
Go(N)
i ← 1
while (i≤2)
{
  Go(E)
  i ← i+1
}
while (i≥0)
{
  Go(S)
  i ← i-1
}
while (i≤4)
{
  Go(W)
  i ← i+1
}
i ← 0
while (i<5)
{
  Go(N)
  i ← i+1
}
```

Algorithmus 3

```
Go(N)
j ← 2
for (i ← 1 to j step 1)
{
  Go(E)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(S)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(W)
}
j ← j+1
for (i ← 1 to j step 1)
{
  Go(N)
}
```

Algorithmus 4

```
Dir ← [N,E,S,W,N]
Stp ← [1,2,3,4,5]
for (j ← 0 to 4 step 1)
{
  for (i ← 1 to Stp[j] step 1)
  {
    Go(Dir[j])
  }
}
```

Algorithmus 5

```
Dir ← [N,E,W,E,S,N,S,W,E,W,N]
Stp ← [1,2,1,1,2,1,2,4,2,1,5]
for (j ← 0 to 10 step 1)
{
  for (i ← 1 to Stp[j] step 1)
  {
    Go(Dir[j])
  }
}
```

Q1(a) [5 Pkte]	**Bringt Algorithmus 1 Nono auf das Feld X? Wenn Ihre Antwort « Nein » ist und Nono mit einer Wand kollidiert, geben Sie auch die Koordinaten der ersten Wand an, mit der Nono kollidiert.**
Lösung: Ja	

Q1(b) [5 Pkte]	**Bringt Algorithmus 2 Nono auf das Feld X? Wenn Ihre Antwort « Nein » ist und Nono mit einer Wand kollidiert, geben Sie auch die Koordinaten der ersten Wand an, mit der Nono kollidiert.**
Lösung: Nein, F1	

Q1(c) [5 Pkte]	**Bringt Algorithmus 3 Nono auf das Feld X? Wenn Ihre Antwort « Nein » ist und Nono mit einer Wand kollidiert, geben Sie auch die Koordinaten der ersten Wand an, mit der Nono kollidiert.**
Lösung: Ja	

Q1(d) [5 Pkte]	**Bringt Algorithmus 4 Nono auf das Feld X? Wenn Ihre Antwort « Nein » ist und Nono mit einer Wand kollidiert, geben Sie auch die Koordinaten der ersten Wand an, mit der Nono kollidiert.**
Lösung: Ja	

Q1(e) [5 Pkte]	**Bringt Algorithmus 5 Nono auf das Feld X? Wenn Ihre Antwort « Nein » ist und Nono mit einer Wand kollidiert, geben Sie auch die Koordinaten der ersten Wand an, mit der Nono kollidiert.**
Lösung: Nein, C3	

Frage 2 – Scheitelpositionen

Bei einem Array $a[\,]$ mit n Zahlen ist eine *Scheitelposition von* $a[\,]$ eine Position i in diesem Array, so dass $a[i]$ mindestens so groß ist wie das Element links und das Element rechts von ihr (falls sie existieren). Beachten Sie, dass Arrays ab 0 indiziert werden: Das erste Element befindet sich an der Position 0 und das letzte Element an der Position $n-1$.

Wenn zum Beispiel $a = [2, 7, 4, 5]$, dann sind die Scheitelpositionen von $a[\,]$ die Positionen 1 und 3 (entsprechend den Werten 7 und 5). Ein weiteres Beispiel: Wenn $a = [6, 6, 6]$, dann sind alle Positionen (0, 1 und 2) Scheitelpositionen. Man kann leicht zeigen, dass jedes Array mindestens eine Spitzenposition hat.

Q2(a) [2 Pkte]	**Nennen Sie alle Scheitelpositionen in dem Array $a = [4, 2, 5, 3]$.**
Lösung: 0, 2	

Q2(b) [2 Pkte]	**Nennen Sie alle Scheitelpositionen in dem Array $a = [0, 10, 10, 0, 0]$.**
Lösung: 1, 2 et 4	

Es gibt viele Möglichkeiten, eine Scheitelposition zu finden, aber es ist sehr leicht, sich zu irren! Ihr Freund Alex hat Ihnen gestern Abend um 3 Uhr morgens die folgenden Programmlistings geschickt. Er behauptet, dass seine Programme eine Scheitelposition für jedes beliebige Array $a[\,]$ finden können. Sie denken, dass Alex wahrscheinlich sehr müde war, als er die Programme geschrieben hat, und beschließen, sie zu überprüfen.
Wir gehen davon aus, dass in jedem Programm das Zahlenarray $a[\,]$ $n \geq 2$ Elemente enthält.

Programm 1:

```
for (i ← 1 to n-2 step 1)
{
    if (a[i] >= i-1 and a[i] >= i+1)
    {
        return i
    }
}
if (a[0] > a[n-1])
{
    return 0
}
else
{
    return n-1
}
```

Q2(c) [1 Pkt]	**Welchen Wert gibt Programm 1 zurück, wenn $a = [9, 8, 7, 6, 5]$?**
Lösung: 1	

Q2(d) [1 Pkt]	**Welchen Wert gibt Programm 1 zurück, wenn $a = [0, 1, 3, 4]$?**
Lösung: 2	

Q2(e) [1 Pkt]	**Welchen Wert gibt Programm 1 zurück, wenn $a = [0, 0, 0, 0]$?**
Lösung: 3	

Q2(f) [1 Pkt]	Welchen Wert gibt das Programm 1 zurück, wenn $a = [1, 0, 0, 1]$?
Lösung: 3	

Q2(g) [2 Pkte]	Gibt Programm 1 eine Spitzenposition für jedes Array $a[\,]$ zurück?
Lösung: Nein	

Programm 2:

```
i ← 0
for (j ← 1 to n-1 step 1)
{
    if (a[j] > a[i])
    {
        i ← j
    }
}
return i
```

Q2(h) [1 Pkt]	Welchen Wert gibt Programm 2 zurück, wenn $a = [4, 3, 2, 1]$?
Lösung: 0	

Q2(i) [1 Pkt]	Welchen Wert gibt Programm 2 zurück, wenn $a = [0, 1, 0, 5, 0, 5, 0]$?
Lösung: 3	

Q2(j) [2 Pkte]	Gibt Programm 2 eine Scheitelposition für ein beliebiges Array $a[\,]$ zurück?
Lösung: Ja	

Programm 3:

```
if (a[0] > a[n-1])
{
    return 0
}
else
{
    return n-1
}
```

Q2(k) [1 Pkt]	Welchen Wert gibt das Programm 3 zurück, wenn $a = [1, 2, 3]$?
Lösung: 2	

Q2(l) [1 Pkt]	**Welchen Wert gibt Programm 3 zurück, wenn $a = [2, 2, 1]$?**
Lösung: 0	

Q2(m) [2 Pkte]	**Gibt Programm 3 eine Scheitelposition für jedes Array $a[\,]$ zurück?**
Lösung: Nein	

Programm 4:

```
for (i ← 0 to n-2 step 1)
{
    if (a[i] >= a[i+1])
    {
        return i
    }
}
return n-1
```

Q2(n) [1 Pkt]	**Welchen Wert gibt Programm 4 zurück, wenn $a = [1, 2, 3, 2, 1]$?**
	Lösung: 2

Q2(o) [1 Pkt]	**Welchen Wert gibt Programm 4 zurück, wenn $a = [0, 1, 0, 5, 0]$?**
	Lösung: 1

Q2(p) [1 Pkt]	**Welchen Wert gibt das Programm 4 zurück, wenn $a = [2, 2, 1, 0, 5]$?**
	Lösung: 0

Q2(q) [2 Pkte]	**Gibt Programm 4 eine Scheitelposition für jedes Array $a[\,]$ zurück?**
	Lösung: Ja

Programm 5:

```
i = n / 2
while (i > 0 and a[i-1] > a[i])
{
    i ← i-1
}
while (i < n-1 and a[i+1] > a[i])
{
    i ← i+1
}
return i
```

Q2(r) [1 Pkt]	**Welchen Wert gibt das Programm 5 zurück, wenn $a = [1, 2, 3, 4, 5]$?**
	Lösung: 4

Q2(s) [1 Pkt]	**Welchen Wert gibt das Programm 5 zurück, wenn $a = [5, 4, 3, 2, 1]$?**
	Lösung: 0

Q2(t) [1 Pkt]	**Welchen Wert gibt das Programm 5 zurück, wenn $a = [4, 3, 2, 2, 6, 5]$?**
	Lösung: 4

Q2(u) [1 Pkt]	**Welchen Wert gibt das Programm 5 zurück, wenn $a = [0, 4, 2, 5, 3]$?**
Lösung: 1	

Q2(v) [2 Pkte]	**Gibt Programm 5 eine Scheitelposition für ein beliebiges Array $a[\,]$ zurück?**
Lösung: Ja	

Am nächsten Tag schickt Barbara Ihnen ein weiteres Programm, um eine Scheitelposition zu finden. Es klingt kurios, aber Sie sind überzeugt, dass Barbara das Programm richtig geschrieben hat (nicht zuletzt, weil sie Ihnen das Programm im Gegensatz zu Alex nicht mitten in der Nacht geschickt hat). Da Barbara jedoch keine Gelegenheit auslässt, die Kombinationsgabe ihrer Freunde herauszufordern, hat sie einige Lücken gelassen, die es auszufüllen gilt.

```
l = 0
r = ...          // (e1)
while (...)      // (e2)
{
    mid ← (l + r) / 2
    if (...)     // (e3)
    {
        l ← mid + 1
    }
    else
    {
        r ← mid
    }
}
return l
```

Q2(w) [2 Pkte]	**Wie lautet der Ausdruck `(e1)` in Barbaras Programm?**
Lösung: `n-1`	

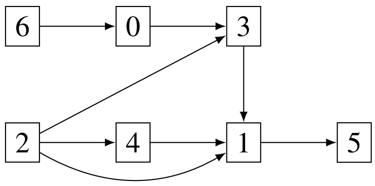
Q2(x) [2 Pkte]	**Wie lautet der Ausdruck `(e2)` in Barbaras Programm?**
Lösung: `l<r` (andere Lösung `l≠r`)	

Q2(y) [2 Pkte]	**Wie lautet der Ausdruck `(e3)` in Barbaras Programm?**
Lösung: `a[mid] <= a[mid+1]` oder `a[mid] < a[mid+1]`	

Frage 3 – Die Computerfabrik

Das Unternehmen be-OI Vater und Sohn stellt Computerteile her. Ein Computerbauteil ist ein komplexes Objekt und benötigt zu seiner Herstellung andere Bauteile, die wiederum aus anderen zusammengesetzt werden, *usw.*. Das Unternehmen verfügt über alle Maschinen, um alles herzustellen, kann aber nicht alle gleichzeitig betreiben. Es muss also *die richtige Reihenfolge für die Herstellung der Teile* finden, *wobei jede Maschine nur einmal in Betrieb genommen wird*.

Jede Maschine hat eine Nummer von 0 bis $n-1$ und das Unternehmen hat die Abhängigkeiten zwischen den Maschinen mithilfe eines Diagramms wie im folgenden Beispiel dargestellt.



In diesem Schema wird jede Maschine durch ein Rechteck mit der Maschinennummer dargestellt. Ein Pfeil von der Maschine i zur Maschine j bedeutet, dass die Maschine i vor der Maschine j laufen muss, da die von i produzierten Teile für die von j produzierten Teile benötigt werden (die Mengen sind hier nicht relevant: i muss nur einmal laufen, damit j laufen kann). Beispielsweise ist es notwendig, dass Maschine 2 und Maschine 0 vor Maschine 3 laufen. Jede Reihenfolge, in der 3 vor 2 oder vor 0 eingeschaltet wird, ist daher unmöglich. Damit das Diagramm Sinn macht, gehen wir davon aus, dass es niemals einen *Zyklus* enthält, d. h. es ist niemals möglich, von einer Maschine aus eine Reihe von Pfeilen zu verfolgen, um zu derselben Maschine zurückzukehren.

Wie man sieht, können einige Maschinen sofort in Betrieb genommen werden.

Q3(a) [1 Pkt]	**Welche Maschinen kann man nach dem obigen Schema sofort in Betrieb nehmen?**
Lösung: 2 und 6	

Q3(b) [1 Pkt]	**Welche Maschinen kann man gemäß dem obigen Schema starten, wenn man zuvor nur Maschine 2 gestartet hat (zur Erinnerung: Jede Maschine wird nur einmal gestartet)?**
Lösung: 4 und 6	

Welche der folgenden Reihenfolgen sind unter Berücksichtigung der im obigen Diagramm gegebenen Einschränkungen möglich?

	Möglich	Unmöglich	Reihenfolge der Maschinen
Q3(c) [1 Pkt]	☐	☒	0,1,2,3,4,5,6
Q3(d) [1 Pkt]	☒	☐	6,0,2,3,4,1,5
Q3(e) [1 Pkt]	☐	☒	6,5,4,3,2,1,0
Q3(f) [1 Pkt]	☒	☐	2,4,6,0,3,1,5
Q3(g) [1 Pkt]	☐	☒	2,4,1,6,0,3,5

Ganz klar: Es gibt eine ganz bestimmte Bedingung, die die Maschinen (in einem Diagramm wie dem obigen) erfüllen müssen, damit man sie sofort in Betrieb nehmen kann.

Q3(h) [2 Pkte]		**Welche der folgenden Aussagen charakterisiert genau die Maschinen, die man sofort in Betrieb nehmen kann?**
	☐	Maschinen, die eine gerade Anzahl an eingehenden Pfeilen haben.
	☐	Maschinen, die höchstens zwei eingehende Pfeile haben.
	☒	Maschinen, die keine eingehenden Pfeile haben.
	☐	Maschinen, die mindestens einen eingehenden Pfeil haben.
	☐	Maschinen, die keine eingehenden Pfeile und eine gerade (Maschinen-)Nummer haben.

Wir suchen nun nach einem Algorithmus, um eine Reihenfolge zu finden, in der die Maschinen laufen sollen, die mit dem Diagramm vereinbar ist. Das Diagramm ist als Matrix $M[n][n]$ mit Boolschen Werten gegeben, wobei $M[i][j]$ nur dann **`true`** ist, wenn es einen Pfeil gibt, der von Maschine i zu Maschine j führt.

Hier ist die Matrix für das als Beispiel angegebene Diagramm.
Kästchen, die den Wert **`true`** enthalten, wurden hervorgehoben, um die Lesbarkeit zu verbessern.

	0	1	2	3	4	5	6
0	false	false	false	**true**	false	false	false
1	false	false	false	false	false	**true**	false
2	false	**true**	false	**true**	**true**	false	false
3	false	**true**	false	false	false	false	false
4	false	**true**	false	false	false	false	false
5	false	false	false	false	false	false	false
6	**true**	false	false	false	false	false	false

M[ ][ ]

$M[2][4]$ ist mit einer durchgezogenen Linie umrandet, während $M[4][2]$ mit einer gestrichelten Linie umrandet ist.
$M[2][4] =$ **`true`**, da es einen Pfeil von Maschine 2 zu Maschine 4 gibt.
$M[4][2] =$ **`false`**, da es keinen Pfeil von Maschine 4 zu Maschine 2 gibt.

Der von uns vorgeschlagene Algorithmus basiert auf einer Datenstruktur namens *Warteschlange*. Dabei handelt es sich um eine Warteschlange mit einem Anfang und einem Ende. Wir fügen die Elemente am Ende der Warteschlange hinzu und verarbeiten die Elemente beginnend mit dem Element am Anfang der Warteschlange. Wir haben die Funktionen: `Empty()`, die die Warteschlange leert (sie enthält keine Elemente mehr); `Insert(v)`, die das Element v an das **Ende** der Warteschlange einfügt.; `Remove()`, die das Element am **Anfang** der Warteschlange zurückgibt und es aus der Warteschlange entfernt; und schließlich `IsEmpty()`, die **`true`** zurückgibt, wenn die Warteschlange leer ist, und **`false`** , wenn sie mindestens ein Element enthält.

Der Algorithmus ist wie folgt.

1. Zuerst werden alle Maschinen in die Warteschlange eingefügt, die man sofort zum Laufen bringen kann.
2. Dann wiederholen wir die gleichen Aktionen, bis die Warteschlange leer ist:
   (a) eine Maschine am Anfang einer Reihe nehmen und sie bedienen;
   (b) wenn durch die Ausführung dieser Maschine andere Maschinen laufen können, werde diese neuen Maschinen in die Warteschlange eingefügt.

Bemerkungen.
Ein boolscher Wert **`bool`** kann direkt in einer **`if (bool){...}`**. Der Operator **`not`** wirkt sich wie folgt auf boolesche Werte aus: **`not true`** ist gleich **`false`** und **`not false`** ist gleich **`true`**.

Hier ist eine Implementierung des Algorithmus, bei der einige Teile des Codes fehlen:

```
Input  : Eine Matrix M[n][n] der Groesse n × n, die die Abhaengigkeiten zwischen den n
     Machinen darstellt.
Output :
    Laesst die Maschinen in einer Reihenfolge laufen, die kompatibel ist mit M.

Empty()
D ← [0,...,0] // (ein Array der Laenge n, initialisiert mit 0)

for (j ← 0 to n−1 step 1)
{
  for (i ← 0 to n−1 step 1)
  {
    if(M[i][j])
    {
      D[...] ← ... // (A) und (B)
    }
  }
  if (...) // (C)
  {
    Insert(j)
  }
}

while (not IsEmpty())
{
  m ← Remove()
  Die Maschine m wird gestartet
  for (i ← 0 to n−1 step 1)
  {
    if (M[m][i])
    {
      D[i] ← ... // (D)
      if (...) // (E)
      {
        Insert(i)
      }
    }
  }
}
```

Q3(i) [2 Pkte]	Welcher Ausdruck (A) fehlt im Algorithmus?
Lösung: j	

Q3(j) [2 Pkte]	Welcher Ausdruck (B) fehlt im Algorithmus?
Lösung: $D[j]+1$	

Q3(k) [2 Pkte]	Welche Bedingung (C) fehlt im Algorithmus?
	Lösung: $D[j] = 0$

Q3(l) [2 Pkte]	Welcher Ausdruck (D) fehlt im Algorithmus?
	Lösung: $D[i] - 1$

Q3(m) [2 Pkte]	Welche Bedingung (E) fehlt im Algorithmus?
	Lösung: $D[i] = 0$

Schließlich möchten wir die Warteschlange und ihre vier Funktionen, die im Algorithmus verwendet werden, mithilfe eines Arrays $f[n]$ der Größe n und der beiden Indizes s und e realisieren. Die Elemente, die sich in der Warteschlange befinden, sind alle Elemente mit den Indizes $s, s+1, \ldots, e-1$. Das Element mit dem Index s ist das Element am Anfang und das mit dem Index $e-1$ ist das Element am Ende. Wenn beispielsweise $s = 3$ und $e = 6$, ist der Anfang der Warteschlange $f[3]$, danach enthält die Warteschlange die Elemente $f[4]$ und $f[5]$. Wenn einer der Indizes das letzte Feld $n-1$ des Arrays erreicht, kann man die Warteschlange trotzdem erweitern, indem man das Array zirkulär behandelt, was bedeutet, dass der Index wieder auf 0 gesetzt wird. Wenn beispielsweise $n = 7$, $s = 5$ und $e = 2$, enthält die Warteschlange die Elemente $f[5], f[6], f[0], f[1]$ in dieser Reihenfolge, wobei $f[5]$ am Anfang steht. Sie werden gebeten, den unten stehenden Code zu vervollständigen, um diese Implementierung der Warteschlange zu erhalten. Tipp: Verwenden Sie die mathematische Funktion *modulo* (dargestellt durch den Operator %): $a\%b$ ist der Rest der Division von a durch b.

```
Empty()
{
  s ← 0
  e ← 0
}

Insert(v)
{
  f[e] ← v
  e ← ... // (F)
}

IsEmpty()
{
  return (...) // (G)
}

Remove()
{
  v ← f[s]
  s ← ... // (H)
  return v
}
```

Q3(n) [2 Pkte]	Welcher Ausdruck (F) fehlt im Algorithmus?
	Lösung: $(e+1)\%n$

Q3(o) [2 Pkte]	**Welche Bedingung (G) fehlt im Algorithmus?**
Lösung: $e = s$	

Q3(p) [2 Pkte]	**Welcher Ausdruck (H) fehlt im Algorithmus?**
Lösung: $(s+1)\%n$	

Frage 4 – LIDAR

LIDAR ist eine Technologie, die Licht oder Laser verwendet, um Entfernungen oder Höhen von Objekten zu bestimmen. Sie gehören zu einem Team, das die genaue Höhe verschiedener Teile der Chinesischen Mauer messen will. Während Sie über die Mauer fliegen, bemerken Sie plötzlich, dass Ihre LIDAR-Ausrüstung ein Problem hat : statt die Höhe eines einzelnen Segments zu messen, misst und addiert er die Höhen mehrerer aufeinanderfolgender Segmente. Sie haben nur die Genehmigung erhalten, heute über die Mauer zu fliegen, und haben keine Zeit, das LIDAR neu zu konfigurieren. Da Sie der Algorithmenexperte des Teams sind, verlassen sich Ihre Kollegen darauf, dass Sie die tatsächlichen Höhen der Segmente aus den vom Scan gesammelten Daten abrufen.

Der Einfachheit halber stellen wir die Wand durch eine Liste $W[\,]$ von n positiver ganzer Zahlen dar ($W[i] \geq 0$).
Jedes $W[i]$ steht für die Höhe eines Segments der Mauer.
Das LIDAR wird q Scans machen, jeder ist die Summe aufeinanderfolgender Elemente von $W[\,]$.

Beispiel: nehmen wir eine Wand mit $n = 4$ Segmenten: $W = [4, 3, 5, 8]$.
Wie üblich beginnen die Indizes bei 0, hier also $W[0] = 4$, $W[1] = 3$, $W[2] = 5$ und $W[3] = 8$.
Die Ergebnisse von zwei LIDAR-Scans könnten lauten (beachten Sie die Schreibweise $W[i \ldots j] = W[i] + \ldots + W[j]$).
$W[1 \ldots 3] = W[1] + W[2] + W[3] = 3 + 5 + 8 = 16$ und $W[0 \ldots 1] = W[0] + W[1] = 4 + 3 = 7$.

Für die nächsten vier Fragen betrachten wir eine Wand $W[\,]$ mit vier Segmenten. Das LIDAR hat $W[0 \ldots 2] = 31$, $W[0 \ldots 3] = 42$ und $W[2 \ldots 3] = 17$ gemessen.

Q4(a) [1 Pkt]	**Wie hoch ist $W[3]$?**
Lösung: 11, da $W[3] = W[0 \ldots 3] - W[0 \ldots 2] = 42 - 31 = 11$	

Q4(b) [1 Pkt]	**Wie groß ist die Höhe von $W[2]$?**
Lösung: 6, da $W[2] = W[0 \ldots 2] + W[2 \ldots 3] - W[0 \ldots 3] = 31 + 17 - 42 = 6$	

Q4(c) [1 Pkt]	**Wie viel ist $W[0 \ldots 1]$?**
Lösung: 25, da $W[0 \ldots 1] = W[0 \ldots 3] - W[2 \ldots 3] = 42 - 17 = 25$	

Q4(d) [1 Pkt]	**Wie viele verschiedene Wände sind mit diesen LIDAR-Messungen kompatibel?**
Lösung: 26, da nur $W[0]$ und $W[1]$ unbekannt sind, ihre Summe aber 25 betragen muss (denken Sie daran, dass eine Höhe 0 wert sein kann).	

Bevor wir das eigentliche Problem lösen, versuchen wir zunächst, die Messungen effizient zu simulieren. Dadurch können wir Testfälle generieren, um zu überprüfen, ob wir alles richtig gemacht haben.

Implementieren wir zunächst einen einfachen, aber langsamen Algorithmus. Die Einträge sind die Wand $W[\,]$ und ihre Länge n, die Anzahl der Scans q und die linken und rechten Enden der Scans in den Arrays $left[\,]$ und $right[\,]$. Die Ausgabe ist ein Array $Lidar[\,]$, dessen Elemente die Summen $W[left[i] \ldots right[i]]$ sind, die von den Scans gemessen wurden.

```
Input  : n, W[], q, left[], right[]
Output : Lidar[]

Lidar[] is initialized to q zeroes.

for (i ← 0 to ... step 1) // (i)
```

```
{
    for (j ← left[i] to ... step 1) // (ii)
    {
        Lidar[i] ← Lidar[i] + ... //(iii)
    }
}
```

Es gibt einige Lücken in dieser Implementierung, können Sie diese füllen?

Q4(e) [1 Pkt]	Wie lautet der Ausdruck (i) im obigen Algorithmus ?
Lösung: $q-1$	

Q4(f) [1 Pkt]	Wie lautet der Ausdruck (ii) im obigen Algorithmus ?
Lösung: $right[i]$	

Q4(g) [1 Pkt]	Wie lautet der Ausdruck (iii) im obigen Algorithmus ?
Lösung: $W[j]$	

Hier ist ein schnellerer Algorithmus mit nur einem Durchgang über die Werte von $W[\,]$ und $left[\,]$ und $right[\,]$. Die Idee ist, zunächst ein Präfixarray $P[\,]$ zu berechnen, so dass $P[0]=0$ und $P[i+1]=W[0\ldots i]$ gilt, und diese dann zur Berechnung von $Lidar[\,]$ zu verwenden. Die Eingaben und Ausgaben sind dieselben wie im ersten Algorithmus.

```
Input  : n, W[], q, left[], right[]
Output : Lidar[]

P[] ← [0,0,---,0] //n+1 "0".

for (i ← 0 to ... step 1) // (i)
{
    P[i + 1] ← ...  // (ii)
}

for (i ← 0 to ... step 1) // (iii)
{
    Lidar[i] ← ... // (iv)
}
```

Sie müssen nur einige Leerstellen ausfüllen.

Q4(h) [1 Pkt]	Wie lautet der Ausdruck (i) im obigen Algorithmus ?
Lösung: $n-1$	

Q4(i) [2 Pkte]	Wie lautet der Ausdruck (ii) im obigen Algorithmus ?
Lösung: $P[i]+W[i]$	

Q4(j) [1 Pkt]	**Wie lautet der Ausdruck (iii) im obigen Algorithmus ?**
Lösung: $q-1$	

Q4(k) [3 Pkte]	**Wie lautet der Ausdruck (iv) im obigen Algorithmus ?**
Lösung: $P[right[i]+1]-P[left[i]]$	

Wenn wir das Array $P[\,]$ des vorherigen Algorithmus aus den Scans berechnen könnten, könnten wir daraus leicht alle Werte $W[i]$ ableiten. Lassen Sie uns versuchen, dies effektiv zu tun. Die Eingaben sind die Anzahl der Segmente n in der Wand, die Anzahl der Scans q, die Enden der Scans in den Arrays $left[\,]$ und $right[\,]$ sowie die Werte der Scans in $Lidar[i] = W[left[i] \dots right[i]]$. **Wichtige Anmerkung: Bei diesem Algorithmus gehen wir davon aus, dass die Scans alle konsistent sind. (d. h. es gibt keine Widersprüche), und dass die Scans nur $P[\,]$ bestimmen.**

Die Matrix $K[\,][\,]$ wird verwendet, um alle Messungen zu enthalten, die wir für jedes Ende eines jeden Durchlaufs kennen. Wenn ein Scan zum Beispiel $W[3\dots7] = 61$ ist, dann wissen wir, dass $P[8] = W[0\dots7] = W[0\dots2] + W[3\dots7] = P[3] + 61$ und auch, dass $P[3] = P[8] - 61$. Um dies zu berücksichtigen, speichern wir die Bedingung $(8, 61)$ in der Liste $K[3]$ und die Bedingung $(3, -61)$ in der Liste $K[8]$. Achten Sie auf die Hinweise !

Beachten Sie, dass $K[\,][\,]$ ein Array von Listen ist und dass die Elemente jeder Liste $K[i][\,]$ Zahlenpaare $(Index, Wert)$ sind.
Die Zeile `foreach ((index, scan)in K[current])` des Codes wird eine Schleife starten, die einmal für jedes Element in der Liste $K[current][\,]$ ausgeführt wird. Innerhalb dieser Schleife können die Werte von `index` und `scan` verwendet werden.

Die Operation `append x in y` im Code fügt x als neues Element auf der rechten Seite der Liste y hinzu. Wenn z. B. $x = 4$ und $y = [6, 2]$, dann wird y zu $[6, 2, 4]$.
Die umgekehrte Operation ist `remove last element`: Sie wandelt $y = [6, 2, 4]$ in $[6, 2]$ um und weist der angegebenen Variablen 4 zu.

```
Input  : n, q, left[], right[], Lidar[]
Output : P[]

Initialize K[][] to n+1 empty arrays.

for (i ← 0 to q - 1 step 1)
{
    append (..., Lidar[i]) to K[left[i]]     // (i)
    append (...) to K[...] // (ii), (iii)
}

Initialize processed[] to n values false.
P[0] ← 0
processed[0] ← true
Initialize todo[] to [0]

while (todo[] is not empty)
{
    current ← remove last element of todo[]
    foreach ((index, scan) in K[current][])
    {
        P[index] ← ... // (iv)
        if (not processed[index]])
```

```
        {
            processed[index] ← true
            append index to todo
        }
    }
}
```

Q4(l) [1 Pkt]	Wie lautet der Ausdruck (i) im obigen Algorithmus ?
Lösung: $right[i] + 1$	

Q4(m) [3 Pkte]	Wie lautet der Ausdruck (ii) im obigen Algorithmus ?
Lösung: $left[i], -Lidar[i]$	

Q4(n) [1 Pkt]	Wie lautet der Ausdruck (iii) im obigen Algorithmus ?
Lösung: $right[i] + 1$	

Q4(o) [5 Pkte]	Wie lautet der Ausdruck (iv) im obigen Algorithmus ?
Lösung: $P[current] + scan$	

Die folgenden Scans von **einer Wand mit 9 Segmenten** sind konsistent und bestimmen eindeutig $P[\,]$. (Sie können den vorherigen Algorithmus verwenden) und $W[\,]$.

- $W[0\ldots0] = 4$
- $W[0\ldots3] = 18$
- $W[0\ldots5] = 20$
- $W[1\ldots4] = 14$
- $W[2\ldots8] = 23$
- $W[3\ldots4] = 1$
- $W[4\ldots7] = 15$
- $W[5\ldots7] = 15$
- $W[5\ldots8] = 18$
- $W[7\ldots8] = 7$
- $W[8\ldots8] = 3$

Q4(p) [4 Pkte]	Wie hoch sind die 9 Segmente der Mauer $W[\,]$?
Lösung: $[4, 9, 4, 1, 0, 2, 9, 4, 3]$	

Die folgenden Scans von **einer Wand mit 9 Segmenten** sind konsistent, bestimmen $W[\,]$ aber nicht eindeutig.

- $W[0\ldots3] = 22$
- $W[0\ldots4] = 22$
- $W[1\ldots1] = 2$
- $W[1\ldots4] = 16$
- $W[1\ldots5] = 24$
- $W[2\ldots4] = 14$
- $W[3\ldots6] = 19$
- $W[3\ldots8] = 30$
- $W[7\ldots8] = 11$

Q4(q) [4 Pkte]	**Wie viele Wände $W[\,]$ sind mit den gegebenen Scans kompatibel ?**
Lösung: 144, ist eine beliebige Wand $[6, 2, 14 - x, x, 0, 8, 11 - x, y, 11 - y]$ möglich, in der x und y jeweils einen beliebigen Wert zwischen 0 und 11 annehmen können.	

Frage 5 – Binairo

Binairo alias **Takuzu** ist ein Logikspiel mit einfachen Regeln, aber komplexen Lösungen.
Gespielt wird auf einem quadratischen Gitter mit gerader Größe ($2n$ Zeilen mit $2n$ Zellen).
Zu Beginn enthalten einige Zellen eine Spielfigur, entweder schwarz oder weiß. Alle anderen Zellen sind leer.
Das Ziel ist es, einen Spielstein in jede Zelle zu setzen und dabei die folgenden Regeln zu beachten.

- **Regel 1:** Jede Zeile oder Spalte muss n weiße Steine und n schwarze Steine enthalten.
- **Regel 2:** In jeder Zeile oder Spalte dürfen zwei aufeinanderfolgende Spielsteine derselben Farbe stehen, aber nicht drei.
- **Regel 3:** Jede Zeile und jede Spalte ist einzigartig (wir werden diese Regel im Folgenden nicht mehr benötigen).

Hier ist ein Beispiel für ein 6x6 Binairo-Gitter (links) und die einzige Lösung (rechts).

●					
○	○				
				○	
	●		●		●
			●		

●	○	●	○	●	○
○	○	●	●	○	●
●	●	○	○	●	○
●	○	●	○	○	●
○	●	○	●	○	●
○	●	○	●	●	○

Hier sind zwei Beispiele für Zeilen, die sich nicht an die Regeln halten.

○○●○●○○● Regel 1 nicht beachtet (es gibt mehr weiße als schwarze Steine).

○○●●●○○● Regel 2 nicht befolgt (es gibt mehr als zwei aufeinanderfolgende schwarze Steine).

Vervollständigen Sie die folgenden Zeilen **unter Beachtung der ersten beiden Regeln**.
Wenn nötig, antworten Sie hier im Entwurf, bevor Sie Ihre Antworten **auf den Antwortbogen kopieren**.

Q5(a) [1 Pkt]	**Vervollständigen Sie die Zeile** ● ● _ _
	Lösung: ●●○○

Q5(b) [1 Pkt]	**Vervollständigen Sie die Zeile** ○ _ _ ● ● _ _ ○
	Lösung: ○●○●●○●○

Q5(c) [1 Pkt]	**Vervollständigen Sie die Zeile** ● _ ● _ _ _ _ _ ● ●
	Lösung: ●○●○○●○○●●

Wie viele Möglichkeiten gibt es, die folgenden Zeilen zu vervollständigen **und die Regeln zu befolgen** ?

Q5(d) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten** ○ ● _ _
	Lösung: 2

Q5(e) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	○ ○ _ _
Lösung: 1		

Q5(f) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	_ ○ _ _ ● _
Lösung: 3		

Q5(g) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	○ _ _ ● ○ _ _ ●
Lösung: 4		

Q5(h) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	○ ○ ● ○ _ _ _ _
Lösung: 2		

Q5(i) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	○ _ ○ _ _ _ _ _ ○ ○
Lösung: 1		

Q5(j) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	○ _ _ ○ _ _ ○ _ _ ○
Lösung: 6		

Q5(k) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	● _ ○ ○ _ ● _ ○ _ ●
Lösung: 0		

Q5(l) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	_ _ _ _
Lösung: 6		

Q5(m) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	_ _ _ _ _ _
Lösung: 14		

Q5(n) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	○ ○ ● _ _ _ _ _ _ ● ○ ○
Lösung: 1		

Q5(o) [1 Pkt]	**Anzahl der Ergänzungsmöglichkeiten**	_ ○ ● _ _ ○ _ _ ● _ _ ○
Lösung: 5		

Das **Programm 1** prüft, ob eine Zeile von schwarzen und weißen Steinen die ersten beiden Regeln erfüllt. Einige Teile des Codes fehlen, sie müssen ergänzt werden.

Die zu überprüfende Zeile wird durch ein Ganzzahlarray $binairo[\,]$ dargestellt. Weiße Spielsteine werden durch 1 und schwarze Spielsteine durch 2 dargestellt. Eine Zahl n ist ebenfalls gegeben: Das Array $binairo[\,]$ enthält $2n$ Elemente (nummeriert von 0 bis $2n-1$) und gemäß **Regel 1** muss es die gleiche Anzahl n an weißen und schwarzen Steinen geben. Das Programm soll **true** zurückgeben, wenn die ersten beiden Regeln erfüllt sind, und sonst **false**.

Beispiele.

- Für ●○○●○●○●○ $n = 5$, $binairo = [2, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1]$, gibt das Programm **true** zurück.
- Für ○●●○○ $n = 3$, $binairo = [1, 2, 2, 1, 1]$, gibt das Programm **false** zurück.
- Für ●○●●○●○● $n = 4$, $binairo = [2, 1, 2, 2, 1, 2]$, gibt das Programm **false** zurück.

Das Programm durchläuft $binairo[\,]$ und verwendet die Variable $t1$, um die **Gesamt**anzahl der weißen Steine zu zählen, und die Variable $c1$, um die Anzahl **aufeinanderfolgender** weißer Steine unter den zuletzt überprüften Steine zu zählen. Die Variablen $t2$ und $c2$ funktionieren auf die gleiche Weise für die schwarzen Spielsteine.

Programm 1: Die Lösungen stehen an der Stelle der ....

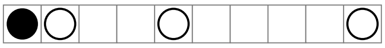
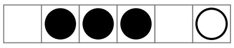
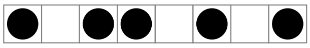
```
Input  : n, binairo[]
Output : true or false
t1 ← 0; t2 ← 0; c1 ← 0; c2 ← 0
for (i ← 0 to 2*n-1 step 1){
    if (binairo[i] = 1) {
        t1 ← ...t1+1
        if (t1 > ...n) {return false}
        c1 ← ...c1+1
        if (c1 = ...3) {return false}
        c2 ← ...0
    }
    else {
        t2 ← ...t2+1
        if (t2 > ...n) {return false}
        c2 ← ...c2+1
        if (c2 = ...3) {return false}
        c1 ← ...0
    }
}
return ...true
```

Q5(p) [6 Pkte]	**Ergänzen Sie die ... im Programm 1.** **Note zwischen 0 und 6. Für jeden Fehler verlieren Sie einen Punkt.**
Lösung: Die Lösungen sind im Anschluss an die ... zu finden.	

Der **Programm 2** ist eine Verbesserung des Programms 1. Damit kann man überprüfen, ob eine Zeile *in der eventuell noch einige Zellen leer sind*, die ersten beiden Regeln erfüllt. Achtung: Das Programm prüft nicht, ob es möglich ist, die Zeile aufzufüllen indem man auch die Regeln befolgt. Es prüft nur, ob die ersten beiden Regeln eingehalten werden, *und zwar von den Spielsteinen, die sich bereits in der Zeile befinden*.

Das Array $binairo[\,]$ kann 0 enthalten, die für leere Zellen stehen (zur Erinnerung: 1 und 2 stehen für Spielfiguren).
Die Variablen $t1$ und $t2$ werden durch ein Array $t[\,]$ mit drei Elementen ersetzt.
$t[0]$ ist unbenutzt, $t[1]$ zählt die Gesamtzahl der weißen Steine, $t[2]$ zählt die Gesamtzahl der schwarzen Steine.
Die Variablen $c1$ und $c2$ werden durch ein Array $c[\,]$ mit drei Elementen ersetzt.
$c[0]$ ist unbenutzt, $c[1]$ zählt die Anzahl der aufeinanderfolgenden weißen Steine, $c[2]$ zählt die Anzahl der aufeinanderfolgenden schwarzen Steine.

Beispiele.

- Für  $n = 5$, $binairo = [2, 1, 0, 0, 1, 0, 0, 0, 0, 1]$, gibt das Programm **true** zurück.
- Für  $n = 3$, $binairo = [0, 2, 2, 2, 0, 1]$, gibt das Programm **false** zurück, da es mehr als 2 aufeinanderfolgende schwarze Steine gibt.
- Für  $n = 4$, $binairo = [2, 0, 2, 2, 0, 2, 0, 2]$, gibt das Programm **false** zur*uck, da es zuviele schwarze Steine gibt.

Programm 2: Die Lösungen stehen an der Stelle der ....

```
Input  : n, binairo[]
Output : true or false
t ← [0,0,0]
c ← [0,0,0]
for (i ← 0 to 2*n-1 step 1){
     j = binairo[i]
     if (...j ≠ 0) {
          t[j] ← ...t[j]+1
          if (...t[j] > n) {return false}
          c[j] ← ...c[j]+1
          if (...c[j] = 3) {return false}
          c[...3-j] ← ...0
     }
}
return ...true
```

Q5(q) [6 Pkte]	**Ergänzen Sie die ... im Programm 2.** **Note zwischen 0 und 6. Für jeden Fehler verlieren Sie einen Punkt.**
Lösung: Die Lösungen stehen an der Stelle der	

Das **Programm 3** zählt die Anzahl der Möglichkeiten, eine Zeile mit n weißen Steinen und n schwarzen Steinen zu füllen (unter Einhaltung der ersten beiden Regeln, wir werden das nicht mehr wiederholen).

Veranschaulichen wir die Idee des Programms für den Fall, dass $n = 2$. Notation: Man schreibt # vor eine Zeile, um die Anzahl der Möglichkeiten anzugeben, die Zeile zu vervollständigen. Wir haben nacheinander:

□□□□ = # ○□□□ + # ●□□□
= (# ○○□□ + # ○●□□) + (# ●○□□ + # ●●□□)
= (# ○○●□ + (# ○●○□ + # ○●●□)) + ((# ●○○□ + # ●○●□) + # ●●○□)
= (# ○○●● + (# ○●○● + # ○●●○)) + ((# ●○○● + # ●○●○) + # ●●○○)
= (1 + (1 + 1)) + ((1 + 1) + 1) = 6

Das Programm 3 verwendet nur eine einzige Eingabe: die Zahl n.
Zunächst definieren wir eine Funktion $TotalBinairo(i, t1, c1, t2, c2)$, die die Anzahl der Wege berechnet, auf denen eine Zeile vervollständigt werden kann, wobei wir wissen, dass:

- die Zellen mit einer Nummer kleiner als i schon aufgefüllt sind,
- es schon $t1$ weiße Steine und $t2$ schwarze Steine gibt,
- die zuletzt gesetzten Zellen sind $c1$ aufeinanderfolgende weiße Steine (und in diesem Fall $c2 = 0$) oder $c2$ aufeinanderfolgende schwarze Steine (und in diesem Fall $c1 = 0$).

Um diese Berechnung durchzuführen, prüft die Funktion zunächst, ob $i = 2n$ ist, was bedeutet, dass die Zeile gefüllt ist. In diesem Fall wird der Wert 1 zurückgegeben, um diese Art des Ausfüllens der Zeile zu zählen. Andernfalls prüft die Funktion, ob man einen weißen Stein in das Feld i setzen kann. Ist dies der Fall, ruft sie sich selbst mit angepassten Werten der Parameter auf und erhält die Anzahl der Möglichkeiten, die Zellen zu vervollständigen, die nach dieser neuen weißen Spielfigur noch übrig sind. Anschließend macht die Funktion dasselbe, indem sie versucht, einen schwarzen Stein in das Feld i zu setzen. Im Falle einer Unmöglichkeit (es ist nicht möglich, einen weißen oder einen schwarzen Stein in Zelle i hinzuzufügen) muss die Funktion 0 zurückgeben.

Die letzte Zeile des Programms ruft die Funktion auf, um die Anzahl der Möglichkeiten zu berechnen, wie eine leere Zeile vervollständigt werden kann.

Programme 3: Die Lösungen stehen an der Stelle der . . . .

```
TotalBinairo( i, t1, c1, t2, c2){
    if (i = 2*n){return 1}
    else {
        TotBin ← 0
        if (t1<n and c1<2){
            TotBin ← TotBin + TotalBinairo( ...i+1, ...t1+1, ...c1+1, ...t2, ...0)
        }
        if (...t2<n and c2<2){
            TotBin ← TotBin + TotalBinairo( ...i+1, ...t1, ...0, ...t2+1, ...c2+1)
        }
        return ...TotBin
    }
}
return TotalBinairo(0, 0, 0, 0, 0)
```

Q5(r) [6 Pkte]	**Ergänzen Sie die . . . im Programm 3.** **Note zwischen 0 und 6. Für jeden Fehler verlieren Sie einen Punkt.**
Lösung: Die Lösungen stehen an der Stelle der . . .	