

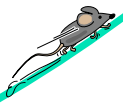


INFORMATIK-BIBER SCHWEIZ
CASTOR INFORMATIQUE SUISSE
CASTORO INFORMATICO SVIZZERA

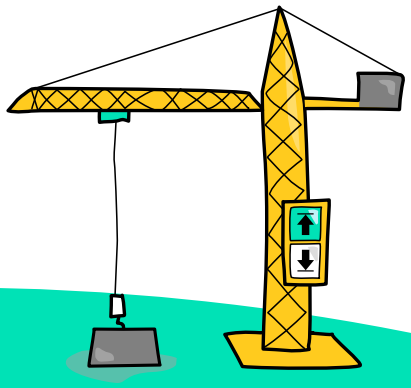
Aufgaben und Lösungen 2026

Schuljahre 11/12/13



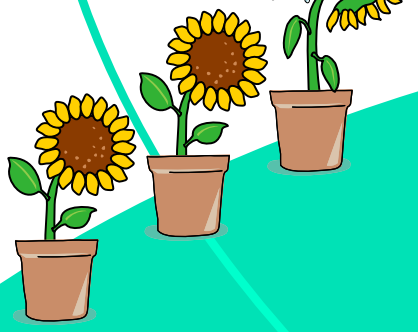


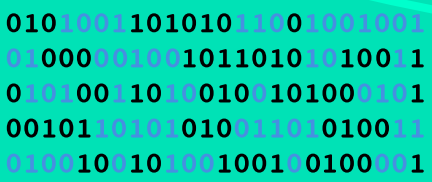
https://www.informatik-biber.ch/



Herausgeber:

Susanne Falkenberg-Thut, Nora A. Escherle,
Jean-Philippe Pellet

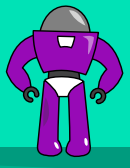




SVIA

www.svia-ssie-ssii.ch
schweizerischer verein für informatik in
der ausbildung // société suisse pour l'infor
matique dans l'enseignement // società sviz
zera per l'informatica nell'insegnamento







HASLERSTIFTUNG ETH zürich | ABZ AUSBILDUNGS- UND BERATUNGSZENTRUM FÜR INFORMATIKUNTERRICHT | Scuola universitaria professionale della Svizzera italiana SUPSI | hep/ | PH LUZERN PÄDAGOGISCHE HOCHSCHULE

Mitarbeit Informatik-Biber 2026

Masiar Babazadeh, Susanne Falkenberg-Thut, Gabriel Parriaux, Jean-Philippe Pellet, Andrea Maria Schmid, Giovanni Serafini

Projektleitung: Nora A. Escherle, Janine Blattner

Herzlichen Dank für die Aufgabenentwicklung für den Schweizer Wettbewerb an:
Juraj Hromkovič, Regula Lacher: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht
Christian Datzko, Gymnasium Liestal
Nadine Fröhlich, Universität Basel
Urs Meier, Pädagogische Hochschule Luzern
Bernadette Spieler, Pädagogische Hochschule Zürich
Nils Böhnke, Mohadesa Ekhlassi, Dirk Schmerenbeck, Andreas Schneider, Jacqueline Staub und Alexander Wiß: Universität Trier, Deutschland

Die Aufgabenauswahl wurde erstellt in Zusammenarbeit mit den Organisatoren von Bebras in Deutschland, Österreich, Ungarn und Luxembourg. Besonders danken wir:
Wolfgang Pohl, Karsten Schulz, Margaretha Schlüter, Kirsten Schlüter, Michael Weigend: Bundesweite Informatikwettbewerbe (BWINF), Deutschland
Wilfried Baumann, Liam Baumann: Österreichische Computer Gesellschaft
Gerald Futschek, Lukas Lehner: Technische Universität Wien
Zsuzsa Pluhár, Bence Gaal: ELTE Informatikai Kar, Ungarn

Die Online-Version des Wettbewerbs wurde auf cuttle.org realisiert. Für die gute Zusammenarbeit danken wir:
Eljakim Schrijvers, Justina Oostendorp, Alieke Stijf, Kyra Willekes: cuttle.org, Niederlande
Andrew Csizmadia: Raspberry Pi Foundation, Vereinigtes Königreich

Die Programmieraufgaben wurden speziell für die Online-Plattform erstellt und entwickelt. Wir danken herzlich für die Initiative:
Jacqueline Staub: Universität Trier, Deutschland
Dirk Schmerenbeck: Universität Trier, Deutschland
Dave Oostendorp: cuttle.org, Niederlande

Für den Support während der Wettbewerbswochen danken wir:
Valérie Verdan: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht

Für die Organisation und Durchführung des Finales 2025 an der ETH danken wir:
Dennis Komm, Hans-Joachim Böckenhauer, Angélica Herrera Loyo, Andre Macejko, Adeline Pittet, Valérie Verdan: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht
Für die Korrektur der Finalaufgaben:
Niu Daniel, Jay Forrer, Alessandra Frank, Oliver Graf, Kassak Jakuk, Simon Krause, Adeline Richter, Colin Rüegg, Benjamin Schlaich, Alexander Skodinis, Emanuel Skodinis, Moritz Stocker, Philip

© Informatik-Biber 2026, SVIA

Whittington, Ursus Wigger, Guiheng Zheng, Mateo Zuluaga
Für die Übersetzung der Finalaufgaben ins Französische:
Jean-Philippe Pellet: Haute école pédagogique du canton de Vaud
Workshop-Leitende für die Finalisten:
Nora A. Escherle, Susanne Falkenberg-Thut, Jean-Philippe Pellet, Gloria Pittet, Clemens Pohle, Hannah Rauterberg, Peter Rossmanith, Dirk Schmerenbeck, Benjamin Schmid, Giovanni Serafini

Christoph Frei: Chragokyberneticks (Logo Informatik-Biber Schweiz)
Andrea Leu, Sarah Beyeler, Lea Hasler, Maggie Winter: Senarclens Leu + Partner AG

Ganz besonderen Dank gilt unseren grossen Förderern Juraj Hromkovič, Dennis Komm, Gabriel Parriaux und der Haslerstiftung. Ohne sie würde es diesen Wettbewerb nicht geben.

Die deutschsprachige Fassung der Aufgaben wurde ähnlich auch in Deutschland, Österreich und Luxemburg verwendet. Die französischsprachige Version wurde ebenfalls in Luxemburg verwendet.

Die französischsprachige Übersetzung wurde von Elsa Pellet und die italienischsprachige Übersetzung von Christian Giang erstellt.

© Informatik-Biber 2026, SVIA

INFORMATIK-BIBER SCHWEIZ
CASTOR INFORMATIQUE SUISSE
CASTORO INFORMATICO SVIZZERA

Der Informatik-Biber 2026 wurde vom Schweizerischen Verein für Informatik in der Ausbildung (SVIA) durchgeführt und massgeblich und grosszügig von der Hasler Stiftung unterstützt. Weitere Partner*innen und Wettbewerbssponsoren, die den Wettbewerb finanziell unterstützt haben, sind das Amt für Kindergarten, Volksschule und Beratung (AKVB) des Kantons Bern, Amt für Wirtschaft AWI des Kantons Zürich, die Basellandschaftliche Gebäudeversicherung, die CYON AG, die Informatik Stiftung Schweiz, sowie die UBS.

Folgende Akademischen Partner unterstützen uns bei der Aufgabenerstellung: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht, Haute école pédagogique du canton de Vaud, La Scuola universitaria professionale della Svizzera italiana, Pädagogische Hochschule Luzern und die Universität Trier.

Dieses Aufgabenheft wurde am 23. November 2026 mit dem Textsatzsystem LaTeX erstellt. Wir bedanken uns bei Christian Datzko für die Entwicklung und langjährige Pflege des Systems zum Generieren der 36 Versionen dieser Broschüre (nach Sprachen und Schulstufen). Das System wurde analog zum Vorgänger-System neu programmiert, welches ab 2014 gemeinsam mit Ivo Blöchliger entwickelt wurde. Jean-Philippe Pellet danken wir für die Entwicklung der `bebras` Toolchain, die seit 2020 für die automatisierte Konvertierung der Markdown- und YAML-Quelldokumente verwendet wird.

Hinweis: Alle Links wurden am 20. November 2026 geprüft.

Die Aufgaben sind lizenziert unter einer Creative Commons Namensnennung – Nicht-kommerziell – Weitergabe unter gleichen Bedingungen 4.0 International Lizenz. Die Autoren sind auf S. 73 genannt.

Vorwort

Der Wettbewerb «Informatik-Biber», der in verschiedenen Ländern der Welt schon seit über 20 Jahren bestens etabliert ist, will das Interesse von Kindern und Jugendlichen an der Informatik wecken. Der Wettbewerb wird in der Schweiz auf Deutsch, Französisch und Italienisch vom Schweizerischen Verein für Informatik in der Ausbildung SVIA durchgeführt und von der Hasler Stiftung unterstützt.

Der Informatik-Biber ist der Schweizer Partner der Wettbewerbs-Initiative «Bebras International Challenge on Informatics and Computational Thinking» (`https://www.bebras.org/`), die in Litauen ins Leben gerufen wurde.

Der Wettbewerb wurde 2010 zum ersten Mal in der Schweiz durchgeführt. 2012 wurde zum ersten Mal der «Kleine Biber» (Stufen 3 und 4) angeboten.

Der Informatik-Biber regt Schülerinnen und Schüler an, sich aktiv mit Themen der Informatik auseinander zu setzen. Er will Berührungsängste mit dem Schulfach Informatik abbauen und das Interesse an Fragestellungen dieses Fachs wecken. Der Wettbewerb setzt keine Anwenderkenntnisse im Umgang mit dem Computer voraus – ausser dem «Surfen» im Internet, denn der Wettbewerb findet online am Computer statt. Für die Fragen ist strukturiertes und logisches Denken, aber auch Phantasie notwendig. Die Aufgaben sind bewusst für eine weiterführende Beschäftigung mit Informatik über den Wettbewerb hinaus angelegt.

Der Informatik-Biber 2026 wurde in fünf Altersgruppen durchgeführt:

- Stufen 3 und 4
- Stufen 5 und 6
- Stufen 7 und 8
- Stufen 9 und 10
- Stufen 11 bis 13

Jede Altersgruppe erhält Aufgaben in drei Schwierigkeitsstufen: leicht, mittel und schwierig. In den Altersgruppen 3 und 4 waren 9 Aufgaben zu lösen, mit je drei Aufgaben in jeder der drei Schwierigkeitsstufen. Für die Altersklassen 5 und 6 waren es je vier Aufgaben aus jeder Schwierigkeitsstufe, also 12 insgesamt. Für die restlichen Altersklassen waren es 15 Aufgaben, also fünf Aufgaben pro Schwierigkeitsstufe.

Für jede richtige Antwort wurden Punkte gutgeschrieben, für jede falsche Antwort wurden Punkte abgezogen. Wurde die Frage nicht beantwortet, blieb das Punktekonto unverändert. Je nach Schwierigkeitsgrad wurden unterschiedlich viele Punkte gutgeschrieben beziehungsweise abgezogen:

	leicht	**mittel**	**schwer**
richtige Antwort	6 Punkte	9 Punkte	12 Punkte
falsche Antwort	−2 Punkte	−3 Punkte	−4 Punkte

© Informatik-Biber 2026, SVIA

Dieses international angewandte System zur Punkteverteilung soll den Anreiz zum blossen Erraten der Lösung eliminieren.

Jede Teilnehmerin und jeder Teilnehmer hatte zu Beginn 45 Punkte (Stufen 3 und 4: 27 Punkte; Stufen 5 und 6: 36 Punkte) auf dem Punktekonto.

Damit waren maximal 180 Punkte (Stufen 3 und 4: 108 Punkte; Stufen 5 und 6: 144 Punkte) zu erreichen, das minimale Ergebnis betrug 0 Punkte.

Bei vielen Aufgaben wurden die Antwortalternativen am Bildschirm in zufälliger Reihenfolge angezeigt. Manche Aufgaben wurden in mehreren Altersgruppen gestellt. Diese Aufgaben hatten folglich in den verschiedenen Altersgruppen unterschiedliche Schwierigkeitsstufen.

Einige Aufgaben werden für bestimmte Altersgruppen als «Bonus» angegeben: sie haben keinen Einfluss auf die Berechnung der Gesamtpunktzahl. Diese Übungen dienen vielmehr dazu, bei mehreren TeilnehmerInnen mit identischer Punktzahl zu entscheiden, wer sich für eine mögliche nächste Runde qualifiziert.

Für weitere Informationen:

Schweizerischer Verein für Informatik in der Ausbildung
SVIA-SSIE-SSII
Informatik-Biber
Janine Blattner

`https://www.informatik-biber.ch/kontaktieren/`
`https://www.informatik-biber.ch/`

© Informatik-Biber 2026, SVIA

Inhaltsverzeichnis

Mitarbeit Informatik-Biber 2026 i

Vorwort iv

Inhaltsverzeichnis vi

1. Video-Kontakte 2026-HU-01b 1

2. Feuergeist 2026-CZ-02 5

3. Muddys Methode 2026-DE-10 9

4. Aika und Benji 2026-JP-05 13

5. Am Rad drehen 2026-AU-04 17

6. Das beste Ramyeon 2026-KR-09 21

7. Biberdamm 2026-CH-08 25

8. Fehler beim Giessen 2026-JP-02b 31

9. Murmelmusik 2026-DE-05 35

10. Münzspiel 2026-AT-02 39

11. Sprung-Spiel 2026-NL-01 43

12. Tacos für lau! 2026-MX-02b 47

13. Versteckspiel 2026-US-03b 51

14. Tinto Azul 2026-PT-01a 55

15. Petri-Ampeln 2026-DE-11 59

16. Schiebespiel 2026-SA-03 63

17. Quartierhilfe 2026-DE-90 67

A. Aufgabenautoren 73

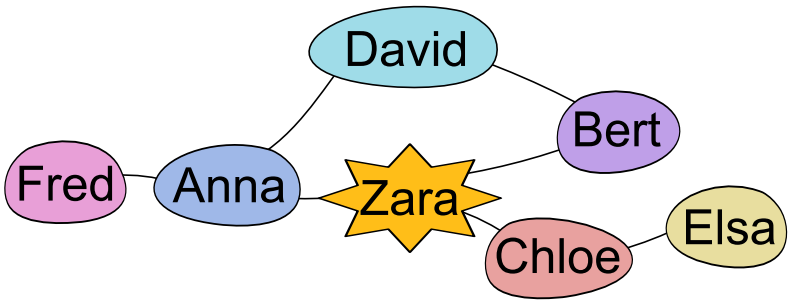
B. Akademische Partner 75

C. Sponsoring 76

© Informatik-Biber 2026, SVIA

1. Video-Kontakte

Zara schaut gerne Videos bei ZickZock. Die nutzen die Kontakte der User, um neue Videos vorzuschlagen. Das Bild zeigt Zaras Kontakte: Anna, Bert und Chloe sind *direkte* Kontakte; David, Elsa und Fred sind *indirekte* Kontakte.



ZickZock schlägt alle Videos vor, die die Kontakte gesehen haben, man selbst aber nicht, und zwar genau dann, wenn:

- ein direkter Kontakt mindestens 2 gleiche Videos gesehen hat wie man selbst.
- ein indirekter Kontakt mindestens 3 gleiche Videos gesehen hat wie man selbst.

Die Tabelle zeigt, welche Videos Zara und ihre Kontakte gesehen haben (✔).

								
Zara	✔	✔	✔	✔				
Anna	✔				✔	✔	✔	
Bert			✔	✔			✔	
Chloe	✔					✔	✔	✔
David		✔			✔	✔		
Elsa	✔	✔	✔			✔		
Fred			✔			✔	✔	✔

Welche neuen Videos schlägt ZickZock Zara vor?

Lösung

ZickZock schlägt Zara die Videos  und  vor.

Die richtige Antwort findet man, wenn man die Tabelle mit den angesehenen Videos genau so analysiert, wie ZickZock das macht:

Person	**Kontakt**								
Zara		✓	✓	✓	✓				
Anna	direkt	✓				✓	✓	✓	
Bert	**direkt**			✓	✓			★	
Chloe	direkt	✓					✓	✓	✓
David	indirekt		✓			✓	✓		
Elsa	**indirekt**	✓	✓	✓			★		
Fred	indirekt			✓			✓	✓	✓

In der ersten Zeile sehen wir, welche Videos Zara gesehen hat. Nun vergleichen wir diese Videos mit den Videos der Kontakte.

- Zuerst betrachten wir die direkten Kontakte Anna, Bert und Chloe. Bert ist der einzige direkte Kontakt, der mindestens 2 gleiche Videos gesehen hat wie Zara (✓). ZickZock schlägt deshalb das einzige Video vor, das Bert gesehen hat, Zara aber noch nicht (★). Videos von Anna und Chloe schlägt ZickZock nicht vor, weil die nicht mindestens 2 gleiche Videos gesehen haben wie Zara.
- Danach betrachten wir die indirekten Kontakte David, Elsa und Fred. Elsa ist der einzige indirekte Kontakt, der mindestens 3 gleiche Videos gesehen hat wie Zara. ZickZock schlägt deshalb das einzige Video vor, das Elsa gesehen hat, Zara aber noch nicht. Videos von David und Fred schlägt ZickZock nicht vor, weil die nicht mindestens 3 gleiche Videos gesehen haben wie Zara.

Dies ist Informatik!

ZickZock ist ein *Empfehlungsdienst*. So etwas kennst du vielleicht von Apps wie Netflix, YouTube oder Spotify: Sie schlagen dir Videos, Lieder oder andere Inhalte vor, die dir gefallen könnten.

Die App ZickZock aus dieser Biberaufgabe verwendet eine einfache Form einer Methode, die in der Informatik *kollaboratives Filtern* heisst. Dabei entstehen Vorschläge aus dem Vergleich mit anderen Personen. Der Vorschlags-Algorithmus der App sucht nach anderen Usern, die gleiche oder ähnliche Dinge mögen wie du. Wenn diese User ein Video gesehen oder ein Lied gehört haben, das du noch nicht kennst, kann dir die App genau dieses Video oder Lied vorschlagen.

In dieser Aufgabe wurden nicht alle anderen User berücksichtigt, sondern nur die Kontakte. Auch soziale Netzwerke wie Facebook oder LinkedIn nutzen Kontakte oder andere «soziale» Informationen

 © Informatik-Biber 2026, SVIA

für Vorschläge. Sie können zum Beispiel berücksichtigen, wer mit wem verbunden ist, welche Gruppen oder Themen jemanden interessieren oder welche Beiträge ähnliche Personen ansehen. So können sie neue Kontakte, Gruppen, Beiträge oder Inhalte vorschlagen.

Empfehlungsdienste können aber auch andere Verfahren nutzen. Bei *inhaltsbasierten Empfehlungen* achtet der Vorschlags-Algorithmus der App stärker darauf, welche Inhalte dich bisher interessiert haben. Wenn du in einer Nachrichten-App oft Artikel über Raumfahrt liest, schlägt dir die App vielleicht weitere Artikel über Raketen, Planeten oder neue Weltraummissionen vor.

Solche Vorschläge sind nützlich, weil du dadurch leichter Dinge findest, die zu deinen Interessen passen. Sie können aber auch zu einer *Filterblase* beitragen. Das bedeutet: Die App zeigt dir vor allem Dinge, die dem ähneln, was du ohnehin schon magst. Dadurch verpasst du vielleicht wichtige oder überraschende Dinge aus anderen Gebieten.

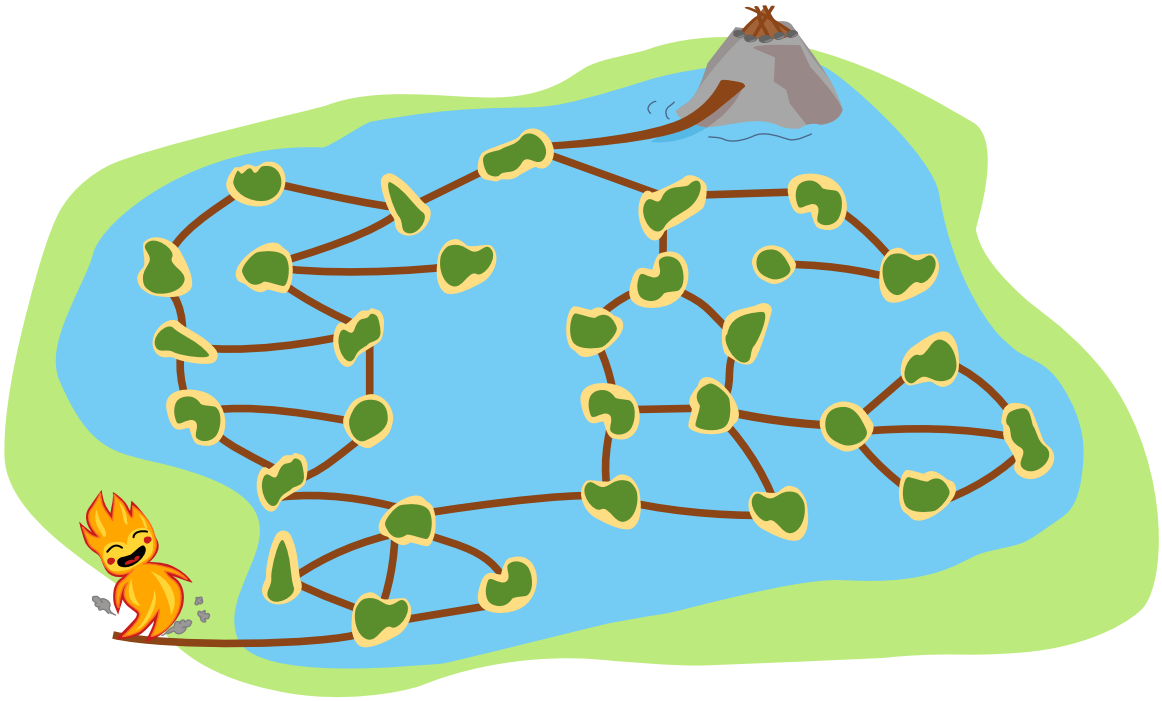
Stichwörter und Webseiten

- Empfehlungsdienst: `https://de.wikipedia.org/wiki/Empfehlungsdienst`
- Kollaboratives Filtern: `https://de.wikipedia.org/wiki/Kollaboratives_Filtern`
- Filterblase: `https://de.wikipedia.org/wiki/Filterblase`

© Informatik-Biber 2026, SVIA

2. Feuergeist

Auf dem See gibt es viele Inseln, die durch Stege miteinander verbunden sind.



Der Feuergeist  geht zur Feuerstelle  auf dem Fels. Auf dem Weg dahin will er so viele Inseln wie möglich besuchen.

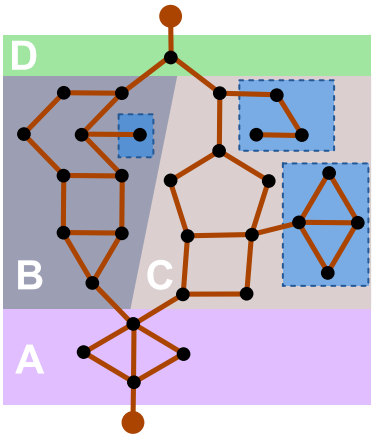
Der Feuergeist muss über die Stege gehen. Das Wasser im See würde ihn löschen. Die Stege aber sind aus Holz und brennen ab, sobald der Feuergeist darüber geht. Daher kann er über keinen Steg zweimal gehen. Die Inseln kann er aber mehrmals besuchen.

Wie viele Inseln kann der Feuergeist höchstens besuchen?

Lösung

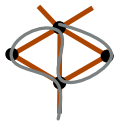
Die richtige Antwort ist 13 Inseln.

Für die Erklärung vereinfachen wir die Darstellung: Wir zeichnen Punkte für die Inseln und Linien für die Stege.

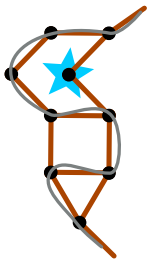


Wir können die Inseln im See in vier Abschnitte unterteilen: vier Inseln unten (A), die letzte Insel vor dem Felsen ganz oben (D) und dazwischen ein Abschnitt links (B) und einer rechts (C). Diese Abschnitte können wir unabhängig voneinander untersuchen, denn: Von jedem Abschnitt gibt es höchstens einen Steg zu einem anderen Abschnitt. Der Feuergeist kann also nicht zwischen den Abschnitten hin und her gehen. Insbesondere kann er von den mittleren Abschnitten B und C nur einen besuchen, da es keinen Steg zwischen diesen Abschnitten gibt. Schliesslich stellen wir noch fest, dass es Gruppen von Inseln gibt, die jeweils nur über einen einzigen Steg mit den anderen Inseln verbunden sind (siehe blaue Rechtecke). Würde der Feuergeist sie betreten, käme er nicht zurück und könnte nicht zur Feuerstelle gehen. Wir brauchen diese Gruppen also nicht berücksichtigen.

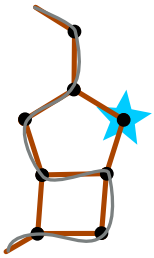
Nun schauen wir uns die einzelnen Abschnitte an.



In Abschnitt A gibt es 4 Inseln und genügend Stege, um alle 4 Inseln zu besuchen und dabei jeden Steg höchstens einmal zu benutzen.



In Abschnitt B gibt es (ohne die Insel im blauen Rechteck) 9 Inseln. Der Feuergeist kann nicht alle besuchen; mindestens 1 Insel bleibt unbesucht. Würde der Geist genau diese Insel (mit einem Stern markiert) besuchen, müsste er mindestens 2 andere Inseln auslassen. In Abschnitt B kann der Feuergeist also höchstens 8 Inseln besuchen.



Abschnitt C hat (ohne die Inseln in den blauen Rechtecken) 8 Inseln. Auch hier bleibt mindestens 1 Insel unbesucht. Auch hier gilt: Würde der Geist genau diese Insel (mit einem Stern markiert) besuchen, müsste er mindestens 2 andere Inseln auslassen. In Abschnitt C kann der Feuergeist also höchstens 7 Inseln besuchen.

In Abschnitt A kann der Feuergeist also höchstens 4 Inseln besuchen, in den Abschnitten B und C höchstens 8 Inseln (indem er durch Abschnitt B geht) und in Abschnitt D 1 Insel; insgesamt also 13 Inseln.

 © Informatik-Biber 2026, SVIA

Dies ist Informatik!

Die Inseln im See und die Stege dazwischen haben wir für die Erklärung als *Graph* modelliert. Diese mathematische Struktur besteht aus einer Menge von *Knoten* und einer weiteren Menge von Knotenpaaren, die als *Kanten* bezeichnet werden. Die Inselwelt des Sees lässt sich recht unmittelbar als Graph darstellen, indem man die Inseln als Knoten und die Stege als Kanten wählt.

Eine Modellierung mit Hilfe von Graphen ist nützlich, weil die Informatik für viele Fragen oder Probleme, die sich für Graphen stellen, Algorithmen zu deren Lösung kennt. Häufig spielen *Wege* durch Graphen eine wichtige Rolle: Ein Weg ist eine Abfolge von Knoten, so dass es zwischen zwei aufeinanderfolgenden Knoten eine Kante gibt.

In zahlreichen Graphenproblemen geht es darum, einen Weg durch einen Graphen zu finden, der bestimmte Bedingungen erfüllt. In dieser Biberaufgabe zum Beispiel ist der Weg mit der grössten Anzahl von Knoten gesucht, der bei der «Start-Insel» des Feuergeistes unten links beginnt, mit dem Felsen endet und in dem jede Kante höchstens einmal vorkommen darf. In anderen Fällen müssen ähnliche Bedingungen erfüllt werden, zum Beispiel bei der Routenplanung für eine Inspektion von Rohrleitungen durch einen Roboter: Hier soll der Roboter alle Rohre (= Kanten) besuchen, aber insgesamt möglichst wenig Strecke zurücklegen müssen.

Stichwörter und Webseiten

- Graph: `https://de.wikipedia.org/wiki/Graph_(Graphentheorie)`
- Graphentheorie: `https://de.wikipedia.org/wiki/Graphentheorie`

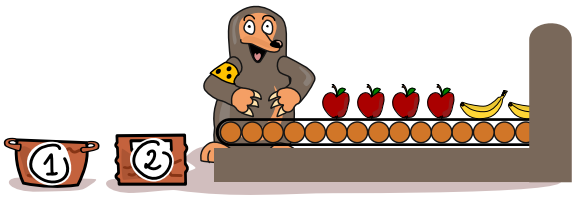


© Informatik-Biber 2026, SVIA

3. Muddys Methode

Auf einem Förderband kommen Früchte in folgender Reihenfolge an: zuerst Äpfel, dann Bananen und schliesslich Birnen.

Muddy der Maulwurf möchte wissen, ob von jeder Sorte gleich viele Früchte über das Förderband kommen. Muddy kann Äpfel, Bananen und Birnen voneinander unterscheiden. Leider kann Muddy nicht zählen, hat aber zwei Körbe zur Hand, die er auch voneinander unterscheiden kann.



Mit den Früchten und den Körben kann er Folgendes machen:



eine Frucht in einen der Körbe legen



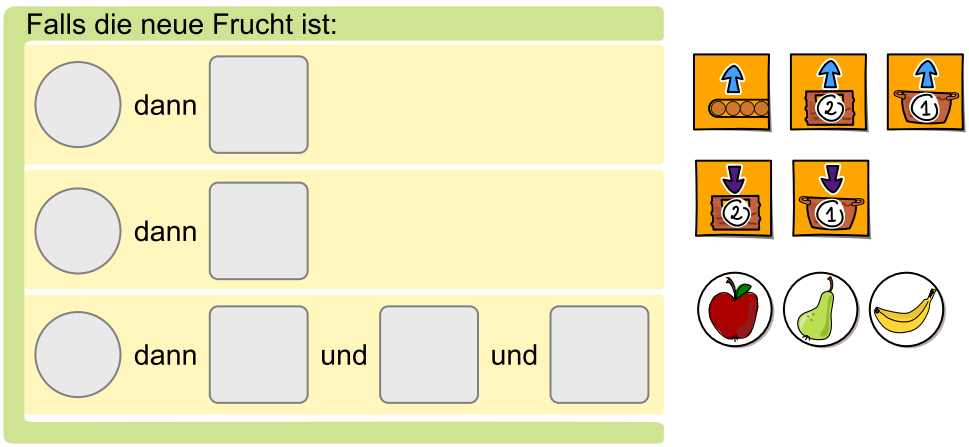
eine beliebige Frucht (Muddy kann nicht sehen, welche) aus einem der Körbe nehmen und beiseite legen



eine Frucht vom Band nehmen und beiseite legen

Damit hat Muddy sich eine Methode überlegt, wie er mit den Früchten auf dem Förderband umgeht. Er wendet diese Methode auf jede neue Frucht an, die bei ihm ankommt. Das macht er solange, bis das Förderband leer ist oder er versucht, eine Frucht aus einem leeren Korb zu nehmen. Wenn dann das Förderband leer ist und beide Körbe ebenfalls leer sind, sind gleich viele Äpfel, Bananen und Birnen über das Band gekommen.

Wie sieht Muddys Methode aus?



Lösung

Muddy hat sich diese Methode überlegt: Er legt alle Äpfel in Korb 1 und alle Bananen in Korb 2 - oder umgekehrt. Für jede Birne auf dem Förderband nimmt er anschliessend genau einen Apfel und eine Banane aus den Körben und legt dann auch die Birne beiseite.



Nur wenn gleich viele Äpfel, Bananen und Birnen über das Band gekommen sind, werden gleichzeitig beide Körbe und das Förderband leer sein. Genauer betrachtet gibt es mehrere Fälle, in denen Muddy aufhört, seine Methode anzuwenden:

Fall 1: Mindestens ein Korb ist leer, aber es ist noch mindestens eine Birne auf dem Band. Dann sind, ...

- falls der Apfelkorb leer ist, mehr Birnen als Äpfel über das Band gekommen;
- falls der Bananenkorb leer ist, mehr Birnen als Bananen über das Band gekommen.

Fall 2: Das Förderband ist leer, aber in mindestens einem Korb liegen noch Früchte. Dann sind, ...

- falls noch Äpfel im Korb liegen, weniger Birnen als Äpfel über das Band gekommen;
- falls noch Bananen im Korb liegen, weniger Birnen als Bananen über das Band gekommen.

Fall 3: Das Förderband und beide Körbe sind leer. Dann sind von allen Sorten gleich viele Früchte über das Band gekommen.

Dies ist Informatik!

Muddy nutzt zwei Körbe, um die Anzahl der Früchte zu vergleichen. Genauer betrachtet verwendet er die Körbe, um die Äpfel und Bananen solange zu «speichern», bis die Birnen kommen. Weil er Äpfel und Bananen getrennt voneinander speichert, kann er, ohne hinzusehen, für jede Birne auf dem Band aus jedem Korb eine Frucht nehmen und sicher sein, dass er insgesamt einen Apfel und eine Banane genommen hat.

In der Informatik gibt es eine ähnliche Struktur wie in dieser Biberaufgabe: Sie besteht aus einem Eingabeband (wie Muddys Förderband) und zwei *Kellern* (das sind Speicher, wie Muddys Körbe) und wird als *Zweikellerautomat* bezeichnet. Der Unterschied ist, dass im Gegensatz zu Muddys

 © Informatik-Biber 2026, SVIA

Körben in den Kellern Ordnung herrscht: Was zuletzt in den Keller gelegt wurde, wird als erstes wieder herausgenommen.

Generell spielen *Automaten* in der theoretischen Informatik eine wichtige Rolle. Ein Informatik-Automat liest Eingabesymbole, führt abhängig vom Symbol und seinem aktuellen Zustand Aktionen aus und geht in einen Folgezustand über. Die Eingabe wird akzeptiert, wenn das Wort komplett gelesen und ein Zielzustand erreicht wurde. Die Menge aller akzeptierten Wörter ist die Sprache, die der Automat erkennt. Ein Automat definiert also eine *formale Sprache*, die alternativ auch durch eine *formale Grammatik* beschrieben werden kann.

Generell besteht in der theoretischen Informatik eine fundamentale Verbindung zwischen *formalen Sprachen* (Mengen von Wörtern), *formalen Grammatiken* (den Regelwerken, die diese Sprachen erzeugen) und *Automaten* (den Maschinen-Modellen, die diese Sprachen erkennen bzw. akzeptieren). Bestimmte Arten von Automaten entsprechen dabei bestimmten Arten von Grammatiken und damit Sprachen. Zum Beispiel wird eine Sprache, die von einem Kellerautomaten mit nur einem Keller akzeptiert wird, in der *Chomsky-Hierarchie* formaler Sprachen als kontextfreie Sprache bezeichnet. Typische Computersprachen wie Programmier- oder Auszeichnungssprachen (z. B. HTML) sind kontextfrei und lassen sich durch effiziente Algorithmen schnell verarbeiten.

Mit einem Keller bzw. Korb könnte Muddy zwei Fruchtsorten vergleichen, mit zwei Kellern drei. Und so weiter? Nein, tatsächlich macht der zweite Keller schon den entscheidenden Unterschied, und weitere Keller bringen keinen zusätzlichen Vorteil. Es ist nämlich eine bekannte Tatsache in der Informatik, dass ein Kellerautomat mit zwei Kellern bereits eine Turingmaschine simulieren kann. Die Turingmaschine steht an der Spitze der Chomsky-Hierarchie und kann jede formale Sprache erkennen, die überhaupt berechenbar ist. Sie - und damit auch der Zweikellerautomat - ist genauso mächtig wie jeder moderne Computer.

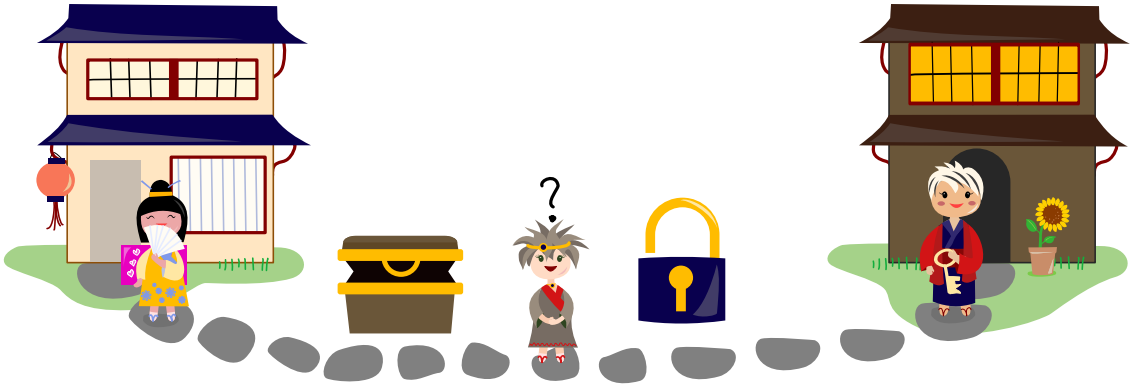
Stichwörter und Webseiten

- kontextfreie Sprache (CFL – context-free language): `https://de.wikipedia.org/wiki/Kontextfreie_Sprache`
- Zweikellerautomat: `https://de.wikipedia.org/wiki/Zweikellerautomat` (`https://de.wikipedia.org/wiki/Kellerautomat`)

© Informatik-Biber 2026, SVIA

4. Aika und Benji


Aika möchte Benji 
etwas geben, ohne dass jemand anders weiss, was es ist. Dafür tut sie es in eine Kiste. Aika und Benji bleiben am liebsten zu Hause. Aber Mana besucht beide regelmässig und kann die Kiste für sie transportieren.



Damit auch Mana nicht weiss, was Aika Benji geben möchte, verschliesst Aika die Kiste mithilfe eines besonderen Vorhängeschlosses von Benji:

- Aika kann das Schloss ohne Schlüssel verschliessen; es schnappt ein.
- Nur Benji kann das Schloss öffnen, denn er hat den einzigen Schlüssel.
- Aika weiss sicher, dass das Schloss von Benji ist, denn er kann ein Schloss auf eine Weise markieren, die niemand anderes nachmachen kann.

Aika und Benji haben sich eine Reihenfolge von Schritten überlegt, mit der Mana die Kiste zwischen Benji und Aika hin und her transportieren kann, ohne dass Mana irgendwie herausfinden kann, was Aika Benji geben möchte. Leider sind die Schritte durcheinander geraten.

Das sind die Schritte. Wie ist die richtige Reihenfolge?

- *Mana bringt die Kiste zu Aika.*
- *Aika legt etwas in die Kiste und verschliesst sie mit dem Vorhängeschloss aus der Kiste.*
- *Benji legt das von ihm markierte Vorhängeschloss in die Kiste und behält den Schlüssel.*
- *Benji öffnet das Vorhängeschloss mit dem Schlüssel.*
- *Benji nimmt das offene Vorhängeschloss und markiert es.*
- *Aika gibt Mana die Kiste.*
- *Benji gibt Mana die Kiste.*
- *Mana bringt die Kiste zu Benji.*

Lösung

Aika möchte Benji in der Kiste etwas schicken, ohne dass Mana irgendwie herausfinden kann, was es ist. Dazu muss Benji zunächst sein besonderes und von ihm markiertes (oder: *signiertes*) Vorhängeschloss ohne Schlüssel an Aika geben:

1. Benji nimmt das offene Vorhängeschloss und markiert es.
2. Benji legt das von ihm markierte Vorhängeschloss in die Kiste und behält den Schlüssel.
3. Benji gibt Mana die Kiste.
4. Mana bringt die Kiste zu Aika.

Nun hat Aika eine Kiste und ein Vorhängeschloss. An der Markierung erkennt sie, dass es Benjis Schloss ist, also nur er das Schloss öffnen kann. Deshalb kann Aika nun beruhigt etwas in die Kiste tun, sie mit Benjis Vorhängeschloss verschliessen und an Benji zurückgeben. Er kann sie mit seinem Schlüssel wieder öffnen:

5. Aika legt etwas in die Kiste und verschliesst sie mit dem Vorhängeschloss aus der Kiste.
6. Aika gibt Mana die Kiste.
7. Mana bringt die Kiste zu Benji.
8. Benji öffnet das Vorhängeschloss mit dem Schlüssel.

Nur Benji kann das Schloss und damit die Kiste öffnen, weil nur er den Schlüssel hat.

Entscheidend ist die Markierung des Schlosses und dass niemand anderes als Benji sie nachmachen kann. Ohne das kann Mana ein anderes Schloss nehmen, für das sie einen Schlüssel hat, und mit Benjis Markierung versehen. Auf dem Weg von Benji zu Aika tauscht Mana dann Benjis Schloss gegen ihres. Aika erkennt Benjis Markierung und kann nicht entdecken, was Mana gemacht hat. Also verschliesst sie die Kiste mit diesem Schloss. Später, auf dem Weg von Aika zu Benji kann Mana das Schloss mit ihrem Schlüssel öffnen und so herausfinden, was Aika Benji geben möchte. Hätte Mana sogar ein Schloss, dass auch Benji mit seinem Schlüssel öffnen könnte, könnte sie die Kiste wieder verschliessen, und auch Benji könnte nicht entdecken, was Mana gemacht hat.

Dies ist Informatik!

Das Vorhängeschloss aus dieser Biberaufgabe, das jede Person ohne Schlüssel schliessen, aber nur eine Person mit Schlüssel öffnen kann, funktioniert so ähnlich wie *asymmetrische Kryptographie*. Bei asymmetrischer Kryptographie gibt es zwei Schlüssel, einen *öffentlichen Schlüssel* zum *Verschlüsseln* einer Nachricht und einen *privaten Schlüssel*, ohne den eine Nachricht nicht oder nur mit unvertretbar hohem Aufwand *entschlüsselt* werden kann. Der öffentliche Schlüssel darf jedem in die Hände gelangen und ist oftmals sogar frei im Internet lesbar. Den privaten Schlüssel darf niemand anders als der Empfänger haben, denn nur er soll die Nachricht wieder entschlüsseln können.

Asymmetrische Kryptographie ist sicher – wenn der *Austausch der Schlüssel* abgesichert werden kann. Vor allem, wenn sich zwei Menschen nicht persönlich treffen (können), muss klar sein, dass ein öffentlicher Schlüssel von einer bestimmten Person ist. Dies kann beispielsweise passieren, indem man

 © Informatik-Biber 2026, SVIA

vertrauenswürdigen Dritten (sogenannten *Zertifizierungsstellen*, die in der Regel im Betriebssystem oder im Webbrowser voreingestellt sind) gegenüber seine Identität nachweist und diese dann bestätigen, dass ein öffentlicher Schlüssel von einer bestimmten Person ist. Wenn das nicht funktionieren würde, könnte ein bösartiger Akteur den öffentlichen Schlüssel des Empfängers durch seinen eigenen ersetzen, die Nachricht abfangen und lesen oder verändern.

Zertifizierungen bzw. Zertifikate werden in Informatiksystemen sehr häufig verwendet, ohne dass man sich dessen bewusst ist. Wenn man beispielsweise auf `https://wettbewerb.informatik-biber.ch/` geht, ist die Übertragung der Webseite verschlüsselt (durch das Protokoll «https» erkennbar). Wenn man genauer hinschaut, stellt man fest, dass die Domäne ein Zertifikat hat, das von einer Zertifizierungsstelle (in diesem Fall «Let's Encrypt») als für diese Domäne gültig bestätigt wurde.

Was ebenfalls nicht zu unterschätzen ist, ist der sichere Umgang mit privaten Schlüsseln. Regelmässig gibt es Berichte, dass beispielsweise durch Social Engineering, Einbruch in Computersysteme oder schlichtweg Verarbeitungsfehler private Schlüssel öffentlich bekannt werden. Da kann es schon genügen, einen Browser zu verwenden, der Passwörter in eine Cloud synchronisiert, oder dass ein Akteur seine bösartige Zertifizierungsstelle im Betriebssystem oder Browser hinterlegt. In einigen Ländern nehmen sich staatliche Stellen das Recht heraus, ansonsten sichere Kommunikation auf diese Weise zu knacken. Wenn dann die vermeintlich sichere Kommunikation nicht mehr sicher ist, kann das höchst unangenehme Folgen für die Betroffenen haben.

Stichwörter und Webseiten

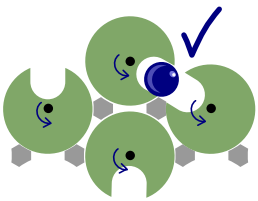
- Asymmetrische Kryptographie:
  `https://de.wikipedia.org/wiki/Asymmetrisches_Kryptosystem`
- Ende-zu-Ende-Verschlüsselung:
  `https://de.wikipedia.org/wiki/Ende-zu-Ende-Verschlüsselung`
- `https://de.wikipedia.org/wiki/Transport_Layer_Security`
- `https://de.wikipedia.org/wiki/Chatkontrolle`

© Informatik-Biber 2026, SVIA

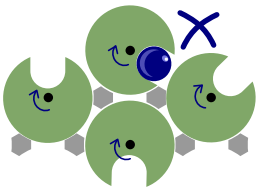
5. Am Rad drehen

Eine Kugel soll sich durch eine Anordnung identischer Kugelräder bewegen. Ein Kugelrad hat ein Fach, in das die Kugel perfekt hineinpasst.

Ein Motor dreht die Räder langsam, alle in derselben Richtung und mit derselben konstanten Geschwindigkeit. Die Drehrichtung der Räder kann mit einem Schalter jederzeit geändert werden.



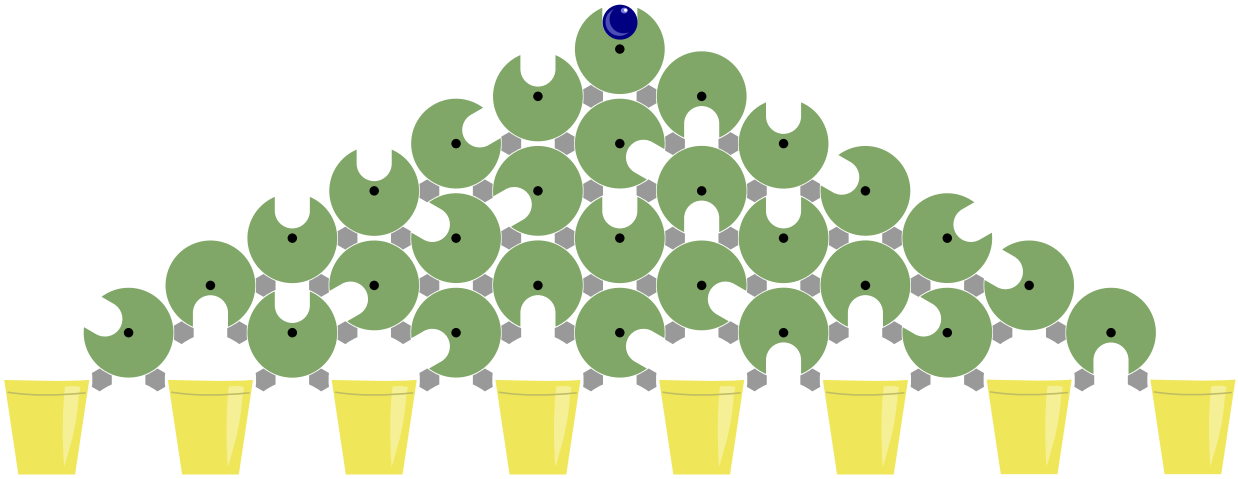
Durch die Drehung können die Fächer zweier benachbarter Räder aufeinander treffen. Dann rollt die Kugel nach unten.



Bei dieser Anordnung treffen keine zwei Fächer jemals aufeinander. Die Kugel bleibt im oberen Rad.

Die Stopper zwischen den Rädern verhindern, dass die Kugel zwischen die Räder rollen oder ganz aus der Anordnung herausfallen kann.

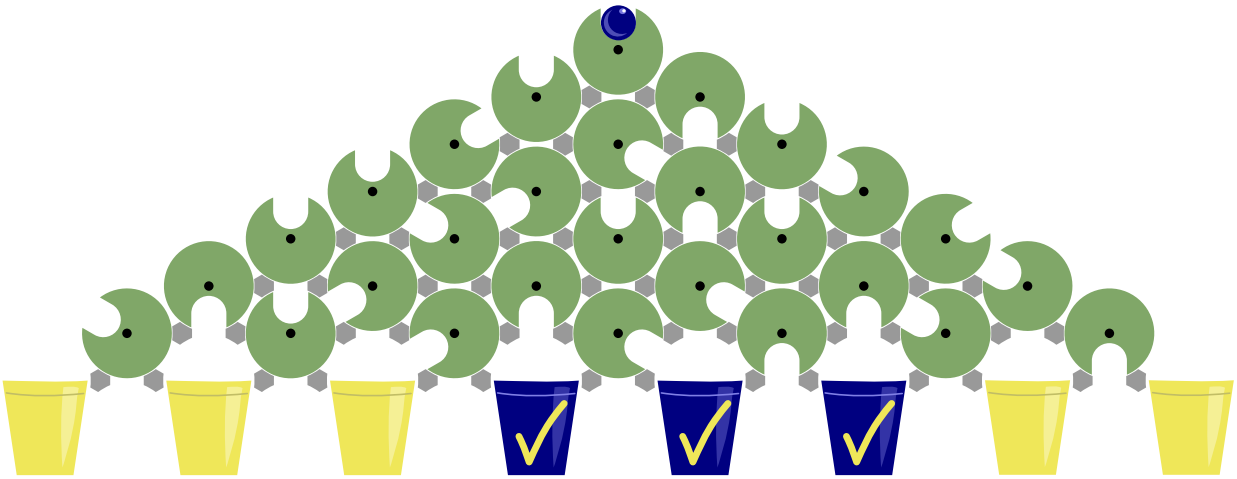
Hier siehst du eine grössere Anordnung von Kugelrädern. Die Kugel wird in das oberste Rad gelegt. Unter den Rädern sind einige Eimer.



In welchen Eimern kann die Kugel landen?

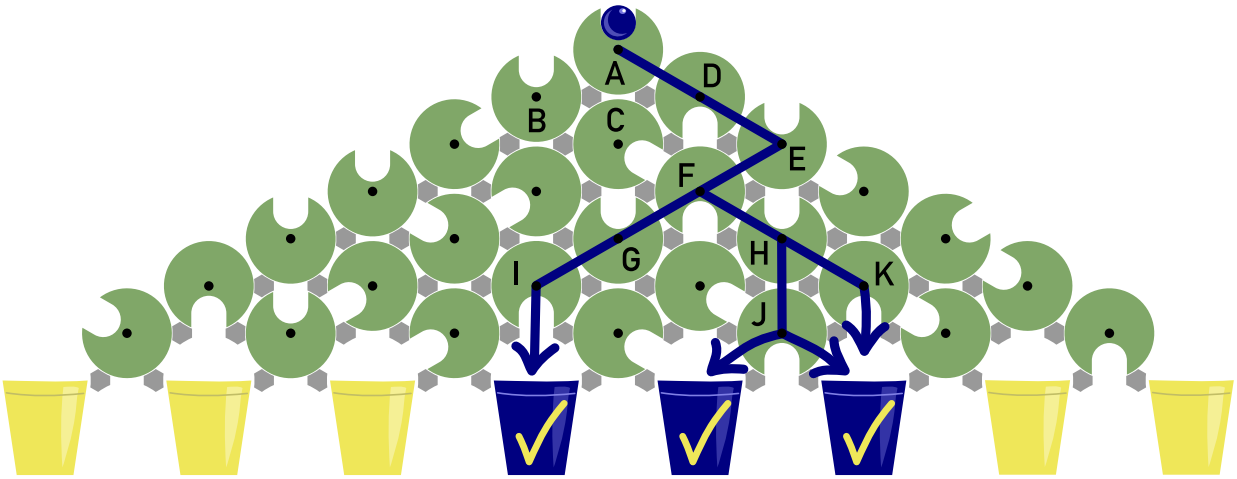
Lösung

So ist es richtig:



Um die richtige Antwort zu finden, könnten wir versuchen, im Kopf die Drehungen aller Räder zu simulieren. Das ist aber sehr schwierig. Wir helfen uns deshalb mit einer Beobachtung: Damit die Kugel von einem Rad in ein anderes Rad rollen kann, müssen die Fächer der beiden Räder in diesem Moment in die entgegengesetzte Richtung zeigen. Weil die Räder sich alle in derselben Richtung und mit derselben Geschwindigkeit drehen, stellen wir fest: Wenn zwei Fächer zu einem Zeitpunkt in entgegengesetzte Richtungen zeigen, trifft das auch zu jedem anderen Zeitpunkt zu.

Das bedeutet: In der Startanordnung können wir für benachbarte Räder untersuchen, ob ihre Fächer in die entgegengesetzte Richtung zeigen. Wenn ja, wissen wir dank der obigen Feststellung, dass die Kugel vom oberen ins untere Rad rollen kann. Wenn nicht, können die Fächer niemals aufeinander treffen und die Kugel wird niemals von oben nach unten rollen. Wir beginnen die Untersuchung beim obersten Rad, in das die Kugel hineingesetzt wird und können so alle Wege bestimmen, die die Kugel durch die Anordnung nehmen kann.



- Die Kugel wird in Rad A gelegt. Von den drei benachbarten Rädern (B, C und D) zeigt nur das Fach von Rad D in entgegengesetzte Richtung wie das Fach von Rad A. Die Kugel kann also von Rad A nur zu Rad D rollen.

 © Informatik-Biber 2026, SVIA

- Von Rad D aus kann die Kugel entsprechen nur zu Rad E rollen.
- Von Rad E aus kann die Kugel nur zu Rad F rollen.
- Von Rad F aus gibt es mehrere Möglichkeiten: Die Fächer von sowohl Rad G als auch Rad H zeigen in entgegengesetzte Richtung wie das Rad von Fach F. Je nach Drehrichtung rollt die Kugel zu Rad G oder zu Rad H.
- Von Rad G aus kann die Kugel nur zu Rad I rollen. Von dort aus landet sie im linken der drei angehakten Eimer.
- Von Rad H aus kann die Kugel zu Rad K oder zu Rad J rollen.
- Von Rad K aus landet sie im rechten der drei Eimer, von Rad J aus im mittleren oder rechten Eimer.

Dies ist Informatik!

In dieser Biberaufgabe solltest du das Verhalten einer Anordnung von Kugelrädern analysieren, um herauszufinden, in welchen Eimern die Kugel landen *kann*. Wer diese Anordnung bzw. das gesamte Räder-System selbst entwickelt hat, könnte auf die gleiche Weise überprüfen, dass die Kugel in genau den Eimern landet, in denen sie landen *soll*. Die Überprüfung, ob ein (informatisches) System bestimmte Eigenschaften besitzt oder bestimmten Anforderungen genügt, wird in der Informatik *Systemverifikation* genannt. Je gravierender die Folgen der Fehlfunktion eines Systems sein kann, desto wichtiger ist die Verifikation: Zum Beispiel sollte bei Software zur Anzeige von Röntgenbildern verifiziert werden, dass das Bild niemals links-rechts gespiegelt angezeigt wird, weil das fatale Folgen für Patientinnen und Patienten haben könnte.

Oft ist es jedoch schwierig zu beweisen, wie sich ein komplexes System verhält. Zuerst könnte man versuchen, das System zu *testen*. Allerdings kann man das immer nur in einer begrenzten Anzahl unterschiedlicher Situationen tun. Häufig gibt es aber unendlich viele mögliche Situationen, zum Beispiel verschiedene Texteingaben für einen Chatbot, Wetterbedingungen beim autonomen Fahren oder die möglichen Abfolgen von Drehungen in dieser Aufgabe. Dann ist Testen nicht ausreichend, um sicher zu sein, dass das System Anforderungen immer erfüllt.

Ein anderer Ansatz ist die *formale Verifikation*, bei der man formal beweist, wie sich das System unter bestimmten Bedingungen verhält. Für die Kugelräder haben wir gezeigt, dass die Kugel nur von einem Rad zu einem anderen rollen kann, wenn ihre Fächer in entgegengesetzte Richtungen zeigen. Wenn ein System sich dynamisch verändert, aber eine bestimmte Eigenschaft dabei immer gleich bleibt, nennt man dies eine *Invariante*. Invarianten sind ein wichtiges Hilfsmittel, um komplexe Systeme zu verstehen, nicht nur in der Informatik.

Stichwörter und Webseiten

- Verifikation: `https://de.wikipedia.org/wiki/Verifizierung_und_Validierung`, `https://www.johner-institut.de/blog/iec-62304-medizinische-software/verifizierung-und-validierung-von-medizinprodukten/`, `https://www.em.cs.uni-frankfurt.de/index.php?id=160`

- Programmverifikation:
  `https://www.studysmarter.de/studium/ingenieurwissenschaften/automatisierung-in-der-informationstechnologie/programmverifikation/`,
  `https://wirtschaftslexikon.gabler.de/definition/programmverifikation-46218`
- Invarianten: `https://loeh.app.uni-regensburg.de/seminars/invarianten.pdf`,
  `https://de.wikipedia.org/wiki/Invariante_(Informatik)`,
  `https://de.wikipedia.org/wiki/Invariante_(Mathematik)`

 © Informatik-Biber 2026, SVIA

6. Das beste Ramyeon

Chefkoch Dong betreibt ein Restaurant für Ramyeon, eine koreanische Nudelsuppe. Auf der Suche nach dem ultimativen Geschmackserlebnis testet er verschiedene Einstellungen: für Wassertemperatur und Kochzeit, sowie für die Mengen an Eiern, Frühlingszwiebeln und Knoblauch.



Jedes Mal, wenn Dong ein neues Rezept umsetzt, geben ihm Stammgäste eine Bewertung auf einer Skala von 0 bis 100. Dong weiss, dass es für jede Einstellung einen einzigen optimalen Wert gibt, unabhängig von den anderen Einstellungen. Ausserdem ist ihm klar: Je weiter die Einstellung vom optimalen Wert entfernt ist, je schlechter wird die Bewertung. Er hat schon einige Rezepte getestet und die Ergebnisse in einer Tabelle festgehalten.

Scanne den QR-Code, um auf die interaktive Version dieser Übung zuzugreifen.

Lösung

Eine perfekte Bewertung von 100 wird mit einer Wassertemperatur von 100 °C, 1 Ei, 20 g Frühlingszwiebeln, 1 TL Knoblauch und einer Kochzeit von 270 s erreicht.

Es ist bekannt, dass der optimale Wert jeder Einstellung unabhängig von den anderen Einstellungen ist. Das können wir nutzen, indem wir nach und nach für jeweils einzelne Einstellung den optimalen Wert bestimmen. Die bisherigen Tests zeigen für einige Einstellungen bereits, wie das funktioniert.

Mit den Rezepten 2 und 3 wird zum Beispiel die optimale Wassertemperatur gefunden. Von den drei getesteten Werten liefert 100 °C die beste Bewertung. Von dort aus sinkt die Bewertung: Für 90 °C ist sie schlechter als für 100 °C und für 80 °C noch schlechter. Weil wir wissen, dass die Bewertung schlechter wird, je weiter der Wert einer Einstellung vom optimalen Wert entfernt ist, steht ohne weitere Tests fest, dass noch niedrigere Temperaturen zu noch schlechteren Bewertungen führen. Höhere Werte als 100 °C kann die Wassertemperatur nicht haben, so dass es nicht möglich ist, mit höheren Temperaturen eine bessere Bewertung zu erreichen. Somit ist 100 °C der optimale Wert, und wir behalten diesen für weitere Tests bei.

Mit den Rezepten 3 bis 7 wird die optimale Menge an Frühlingszwiebeln gefunden. Die Bewertung verändert sich mit einer Erhöhung von 10 g über 15 g auf 20 g, aber verschlechtert sich bei einer weiteren Erhöhung auf 25 g. Also ist 20 g die optimale Menge.

Auf die gleiche Weise kann man mit weiteren Tests die optimalen Werte der anderen Einstellungen bestimmen. Bei den Eiern verschlechtert sich die Bewertung sowohl für weniger als auch für mehr als ein Ei; also ist 1 die optimale Stückzahl. Für Knoblauch ist eine Menge von 1 TL optimal, und die optimale Kochzeit beträgt 270 Sekunden.

Setzt man jede einzelne Einstellung auf ihren optimalen Wert, erhält man eine Bewertung von 100.

Dies ist Informatik!

In dieser Biberaufgabe sollen die Einstellungen eines Rezepts so gewählt werden, dass das Ramyeon möglichst gut schmeckt, also eine möglichst gute Bewertung bekommt. Das ist ein Beispiel für ein *Optimierungsproblem*: Es werden Werte für verschiedene Parameter (hier die Einstellungen) gesucht, die eine Zielfunktion (hier die Bewertung) minimieren oder maximieren. Zur Lösung solcher Probleme kennt die Informatik verschiedene *Optimierungsalgorithmen*.

Optimierungsalgorithmen sind von grosser Bedeutung für *maschinelles Lernen*. Dazu werden Modelle mit vielen (manchmal Billionen!) Parametern «trainiert», indem ihre Parameter so optimiert werden, dass möglichst gute Ergebnisse auf sogenannten Trainingsdaten erreicht werden. Einer der am häufigsten dafür verwendeten Algorithmen ist der *Gradientenabstieg*, der die Steigung der Zielfunktion nutzt, um viele Parameter gleichzeitig in die richtige Richtung anzupassen.

Welcher Optimierungsalgorithmus am besten geeignet ist, hängt von den Eigenschaften des jeweiligen Problems ab. In dieser Aufgabe wissen wir, dass die Parameter unabhängig voneinander sind in ihrem Einfluss auf die Bewertung. Das vereinfacht das Optimieren deutlich, gilt aber bei echten

 © Informatik-Biber 2026, SVIA

Problemen meistens nicht: Es ist zum Beispiel naheliegend, dass bei niedrigerer Wassertemperatur eine längere Kochzeit notwendig ist. Ein geeigneter Algorithmus für den Fall unabhängiger Parameter ist der *Koordinatenabstieg*. Dabei wird immer nur ein Parameter gleichzeitig verändert und die Auswirkung auf das Ergebnis beobachtet. So werden nacheinander die einzelnen Parameter angepasst, um schrittweise eine immer bessere Lösung zu finden.

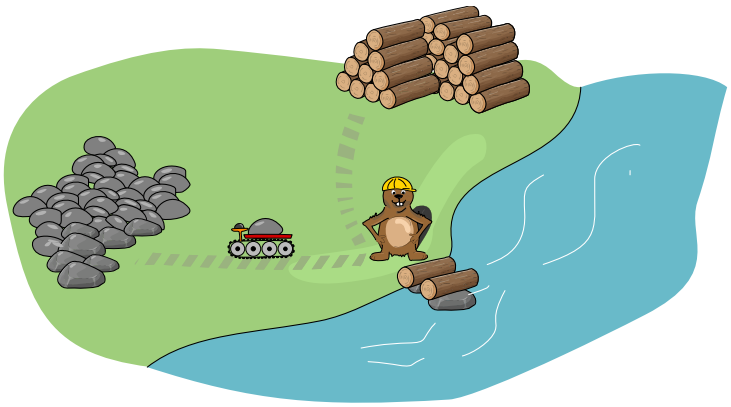
Stichwörter und Webseiten

- Optimierungsprobleme `https://mathepedia.de/Optimierung.html`
- Optimierungsalgorithmen `https://de.statisticseasily.com/Glossar/`

© Informatik-Biber 2026, SVIA

7. Biberdamm

Die Biber wollen einen Damm bauen, indem sie immer zuerst einen Stein und dann ein Stück Baumstamm dem Damm hinzufügen – so lange noch Steine *und* Stämme vorhanden sind. Der Roboter hilft ihnen beim Dammbau.



Ein Programm für den Roboter ist eine Folge von Anweisungen mit Nummern. Die Ausführung eines Programms beginnt bei Anweisung 1. Es gibt diese Anweisungen:



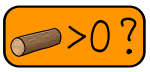
Füge dem Damm einen Stein hinzu und gehe zur nächsten Anweisung.



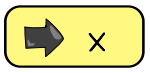
Füge dem Damm einen Stamm hinzu und gehe zur nächsten Anweisung.



Wenn noch Steine vorhanden sind, gehe zur nächsten Anweisung, sonst zur übernächsten.



Wenn noch Baumstämme vorhanden sind, gehe zur nächsten Anweisung, sonst zur übernächsten.

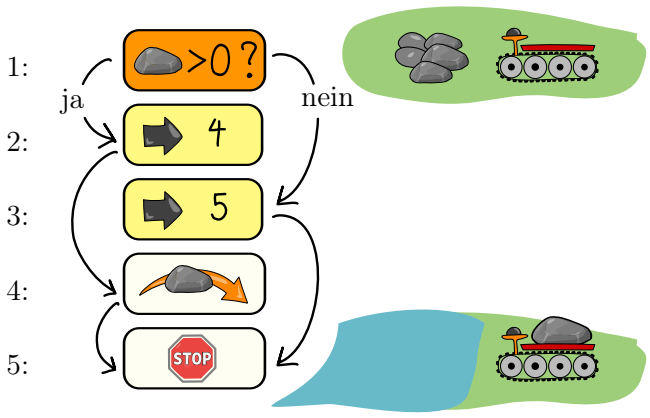


Gehe zur Anweisung mit Nummer X.



Stoppe die Ausführung des Programms.

Hier ist ein Beispielprogramm. Damit fügt der Roboter genau einen Stein hinzu, falls noch Steine vorhanden sind.

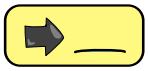


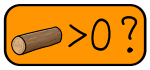
Die Biber wollen ein Programm schreiben, mit dem der Roboter den Damm genau so baut, wie sie es wollen. Sie sind fast fertig. Es fehlen nur noch die Nummern in den «Gehe»-Anweisungen.

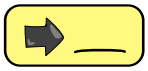
Hilf den Bibern und trage alle fehlenden Anweisungsnummern richtig ein.

1: 

2: 

3: 

4: 

5: 

6: 

7: 

8: 

9: 

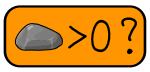
10: 

11: 

 © Informatik-Biber 2026, SVIA

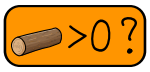
Lösung

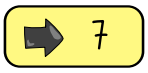
So ist es richtig:

1: 

2: 

3: 

4: 

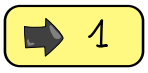
5: 

6: 

7: 

8: 

9: 

10: 

11: 

Anweisung 1 prüft, ob noch Steine vorhanden sind.

Falls ja, führt der Roboter die nächste Anweisung aus:

2: 

... und springt zu Anweisung 4.

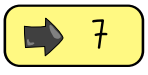
Falls nicht, führt er Anweisung 3 aus:

3: 

... springt zu Anweisung 11 und stoppt. Wenn also kein Stein mehr vorhanden ist, stoppt das Programm.

Wenn noch Steine vorhanden sind, prüft Anweisung 4, ob auch noch Stämme vorhanden sind.

Falls ja, führt der Roboter die nächste Anweisung 5 aus:

5: 

... springt zu Anweisung 7 und fügt damit einen Stein hinzu.

Falls nicht, führt er Anweisung 6 aus:

6: 

... springt zu Anweisung 11 und stoppt. Wenn also kein Stamm mehr vorhanden ist, stoppt das Programm, gleich wie bei Anweisung 3 beim Stein.

Die Anweisung 8 ist interessant. Wenn der Roboter die Anweisung 8 ausführt, hat er gerade in Anweisung 7 einen Stein hinzugefügt und «weiss», dass Steine und Stämme vorhanden sind. Das nächste was er tun muss, ist ein Stamm hinzufügen, also Anweisung 9. Das heisst, das Programm muss nur zur nächsten Anweisung springen. Dies macht die Anweisung 8 für das Programm eigentlich nutzlos, da das Programm immer jeweils die nächste Anweisung ausführt.

Nun führt das Programm also Anweisung 9 aus und fügt einen Stamm hinzu:

9: 

... geht zu Anweisung 10:

10: 

... und springt zu Anweisung 1, beginnt also wieder von vorne.

Insgesamt bewirkt das Programm also, dass der Roboter zuerst einen Stein und dann einen Stamm dem Damm hinzufügt, solange noch Steine und Stämme vorhanden sind – so wie die Biber das wollen.

Die gleiche Wirkung wird erzielt, wenn die Anweisungen 3 und 6 nicht beide direkt zu Anweisung 11 und damit zum Stopp des Programms führen, sondern genau eine der beiden Anweisungen einen Umweg nimmt: Entweder führt Anweisung 3 zu Anweisung 6 (und von dort zu Anweisung 11) oder Anweisung 6 zu Anweisung 3 (und von dort zu Anweisung 11). Es gibt also noch zwei weitere richtige Antworten.

Dies ist Informatik!

Die Programme, die Computerprozessoren direkt ausführen können, sind – wie die Programme dieser Biberaufgabe – einfache Folgen von Anweisungen; nach der Ausführung einer Anweisung macht der Prozessor normalerweise mit der nächstfolgenden Anweisung weiter. Im Maschinencode des Prozessors wird eine Anweisung binär beschrieben, und die ersten Bits (der OpCode) geben an, um welche Anweisung es sich handelt. Ein Programm in Maschinencode ist also eine lange Kette aus Nullen und Einsen.

In den Anfängen der Computerära in den 1940er Jahren musste man Programme noch in reinem Maschinencode schreiben. Dies war extrem fehleranfällig, und die Programme waren schwer zu lesen. Um diesen Prozess zu erleichtern, wurde in den frühen 1950er Jahren die Assemblersprache

 © Informatik-Biber 2026, SVIA

entwickelt. Sie ersetzte die binären OpCodes durch lesbare Abkürzungen, sogenannte Mnemonics, wie MOV für Bewegen, ADD für Addieren oder JMP für Springen. Ein spezielles Programm, der Assembler, übersetzte u.a. diese Kürzel dann in den Maschinencode. Assembler gilt als die erste «höhere» Programmiersprache. Sie baute dem Menschen eine Brücke zur Maschine, ohne dass er die direkte Kontrolle über die Hardware verlor.

Ein fundamentaler Unterschied zwischen Assembler und modernen Programmiersprachen besteht darin, dass es in Assembler weder bedingte Anweisungen («if-then-else») noch Wiederholungsanweisungen («for», «while», ...) gibt. Bedingungen und Wiederholungen werden stattdessen mit Hilfe von Sprungbefehlen (jumps) gebaut. Diese sagen dem Prozessor: «Mache mit Anweisung X weiter anstatt mit der nächsten.»

In der modernen Softwareentwicklung gelten Sprungbefehle (die, wenn es sie gibt, meist «goto» heissen) jedoch als Todfeind guten Programmierens. In den späten 1960er Jahren begründete der Informatiker Edsger W. Dijkstra mit seinem berühmten Aufsatz «Go To Statement Considered Harmful» das Prinzip der strukturierten Programmierung.

Die Basis für «sauberen» und verständlichen Code in Hochsprachen ist es, auf unkontrollierte Sprünge komplett zu verzichten. Stattdessen nutzt man klare Kontrollstrukturen (wie Schleifen und Funktionen), die einen festen Anfang und ein festes Ende haben. Das macht den Code lesbar, wartbar und weitaus weniger fehleranfällig.

Stichwörter und Webseiten

- Assemblersprache: `https://de.wikipedia.org/wiki/Assemblersprache`
- Maschinensprache: `https://de.wikipedia.org/wiki/Maschinensprache`
- Sprunganweisung: `https://de.wikipedia.org/wiki/Sprunganweisung#Sprunganweisungen_auf_Maschinenebene`

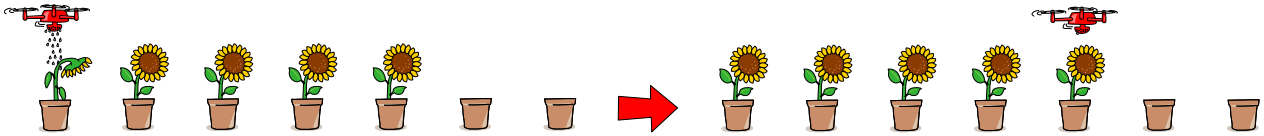


© Informatik-Biber 2026, SVIA

8. Fehler beim Giessen

Archie hat eine Drohne .

Er will, dass die Drohne über eine Reihe von Blumentöpfen fliegt und alle durstigen Blumen  darin giesst.



Archie hat das folgende Programm für die Drohne geschrieben.

Schritt 1:

Starte und fliege zum Topf ganz links.

Schritt 2:

Falls im Topf eine durstige Blume ist, giesse sie.
Sonst fliege einen Topf weiter nach rechts.

Schritt 3:

Falls im Topf keine Blume ist, lande.
Sonst fliege einen Topf weiter nach rechts und gehe zurück zu Schritt 2.

Die Drohne führt die Schritte des Programms nacheinander aus, bis sie landet.

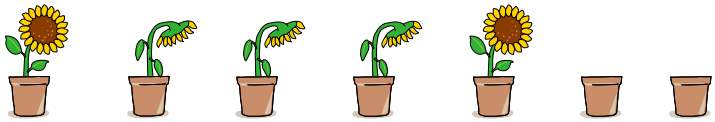
Dabei macht die Drohne nicht immer das, was Archie will.

Gib eine Topfreihe an, in der eine durstige Blume nicht *gegossen wird.*



Lösung

Das ist eine richtige Lösung:



Archies Programm hat ein Problem. Sowohl Schritt 2 als auch Schritt 3 führen eine von zwei möglichen Anweisungen aus, abhängig davon, was der Blumentopf unter der Drohne enthält:

Optionen in Schritt 2:

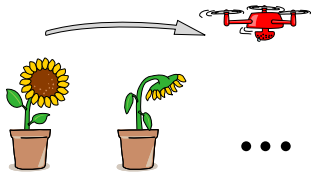
- Wenn der Topf eine durstige Blume enthält → *Blume giessen*
- Wenn der Topf keine durstige Blume enthält → *einen Topf weiter nach rechts fliegen*

Optionen in Schritt 3:

- Wenn der Topf keine Blume enthält → *landen*
- Wenn der Topf eine Blume enthält → *einen Topf weiter nach rechts fliegen*

Das bedeutet: Wenn ein Topf eine durstige Blume enthält, macht die Drohne den «Giesstest» von Schritt 2 (und giesst), fliegt in Schritt 3 zum nächsten Topf nach rechts, wo sie dann wieder Schritt 2 ausführen kann. Insgesamt «springt» die Drohne bezüglich der Ausführung von Schritt 2 um *einen* Topf nach rechts. Wenn ein Topf aber eine nicht-durstige Blume enthält (oder leer ist) und der Topf rechts daneben irgendeine Blume enthält, fliegt die Drohne sowohl in Schritt 2 als auch in Schritt 3 nach rechts. Für den so überflogenen Topf wird Schritt 2 also nicht ausgeführt (und eine durstige Blume darin nicht gegossen), sondern erst wieder für den Topf danach. In diesem Fall springt die Drohne also um *zwei* Töpfe nach rechts.

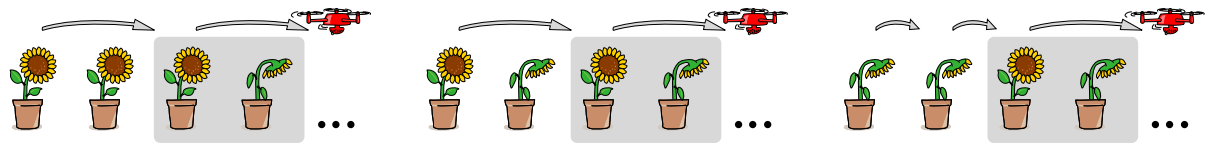
Weil in den Topfreihen leere Töpfe nur am Ende vorkommen, lassen wir leere Töpfe im Weiteren aussen vor und stellen fest: Eine durstige Blume wird übersprungen und *nicht* gegossen, wenn links neben ihr eine Blume steht, die nicht durstig ist und selbst nicht übersprungen wird.



Deshalb kann man auf folgende Weise eine Topfreihe erstellen, in der eine durstige Blume nicht gegossen wird: Man stellt zunächst links neben die durstige Blume eine nicht-durstige Blume. Weiter links davon stellt man durstige bzw. nicht-durstige Blumen so, dass die Drohne mit ihren Sprüngen um 1 oder 2 die nicht-durstige Blume erreicht.

Am einfachsten geht das, indem man die Blumen  direkt an den Anfang der Topfreihe stellt, so wie in der oben angegebenen Topfreihe: Dann wird direkt die zweite Blume in der Reihe nicht gegossen. Aber auch jede andere Topfreihe, die auf diese Weise erstellt wird, ist eine richtige Antwort. Hier sind zum Beispiel alle drei Möglichkeiten dafür zu sorgen, dass die vierte Blume in der Reihe nicht gegossen wird:

 © Informatik-Biber 2026, SVIA



Dies ist Informatik!

Die Drohne in dieser Biberaufgabe arbeitet einen *Algorithmus* ab. Ein Algorithmus ist eine Folge von Anweisungen, die eine konkrete Aufgabe löst. Informatiksysteme wie Computer, Smartphones oder Drohnen werden durch Algorithmen gesteuert. Diese werden typischerweise von Menschen erfunden und in für Informatiksysteme verständlichen Programmiersprachen beschrieben. Ein so enstandenes Programm nennt man in der Informatik auch die *Implementierung* des zugrundeliegenden Algorithmus.

Algorithmen oder auch ihre Implementierungen können Fehler enthalten, wenn etwas im Ablauf nicht den Absichten der Entwicklerin oder des Entwicklers entspricht. Solche Fehler nennt man in der Informatik oft *Bugs*; sie zu finden und zu beheben heisst deshalb *Debugging*. In dieser Biberaufgabe enthält der Algorithmus selbst einen Fehler, der darin besteht, dass in bestimmten Fällen ein Blumentopf mit einer durstigen Blume nicht gegossen wird.

Entwicklerinnen und Entwickler von Programmen testen diese mit verschiedenen Eingaben, um zu prüfen, ob ein Programm bzw. sein Algorithmus einen Fehler enthält. Eine solche Eingabe wird dann *Testfall* genannt. Jede richtige Antwort zu dieser Biberaufgabe ist ein Testfall, der Archies Fehler aufdeckt.

Stichwörter und Webseiten

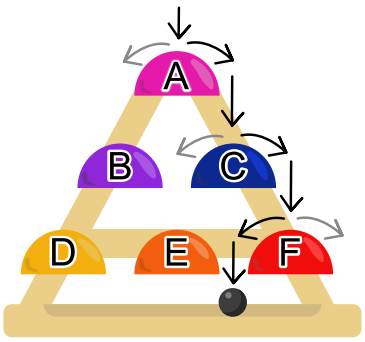
Algorithm, Debugging, Test Case

- Algorithm: `https://de.wikipedia.org/wiki/Algorithmus`
- Debugging: `https://de.wikipedia.org/wiki/Debuggen`
- Softwaretest: `https://de.wikipedia.org/wiki/Softwaretest`

© Informatik-Biber 2026, SVIA

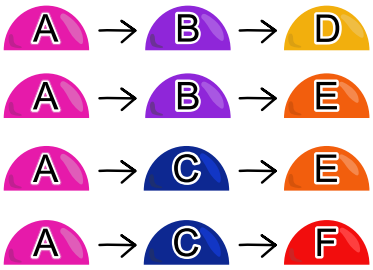
9. Murmelmusik

Marta hat einen Baukasten für musikalische Murmelbahnen. Er hat sechs Glocken, A bis F, die alle unterschiedlich klingen, wenn eine Murmel darauf fällt. Die Glocken können an einem Gestell an sechs Stellen befestigt werden. Rechts ist eine fertige Murmelbahn.



Eine Murmel läuft von oben nach unten durch eine Bahn und fällt auf ihrem Weg auf die oberste Glocke, eine Glocke in der zweiten und eine in der dritten Reihe. So erklingt eine Melodie aus drei Tönen. Auf jeder Glocke rollt die Murmel zufällig nach links oder rechts, so dass es verschiedene Murmelwege entlang den Glocken gibt.

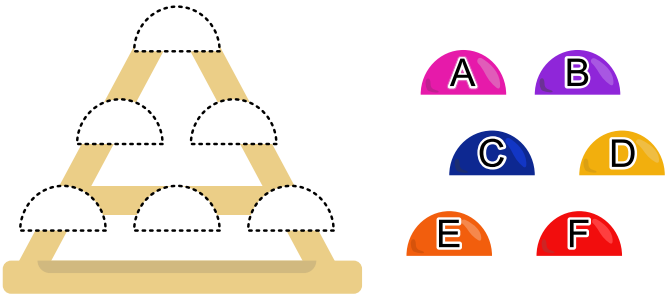
Bei der Murmelbahn oben gibt es diese Murmelwege:



Marta mag bestimmte Melodien sehr und andere gar nicht. Deshalb will sie eine Murmelbahn bauen, so dass es …

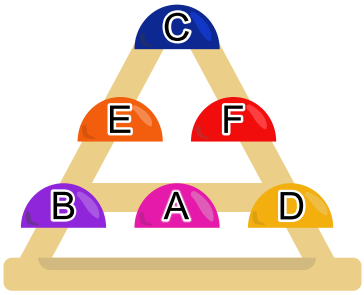
… mindestens diese Murmelwege, …	**… aber keinesfalls diese Murmelwege gibt:**
irgendein Weg mit den drei Glocken A, C, E	alle Wege mit den beiden Glocken B, F
irgendein Weg mit den beiden Glocken B, E	der Weg B → A → D
der Weg C → F → D	

Hilf Marta, die Glocken an den richtigen Stellen zu befestigen.

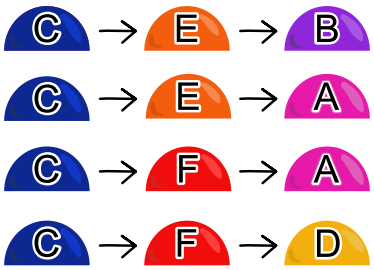


Lösung

So sind die Glocken richtig befestigt:



Es gibt diese «erlaubten» Murmelwege:



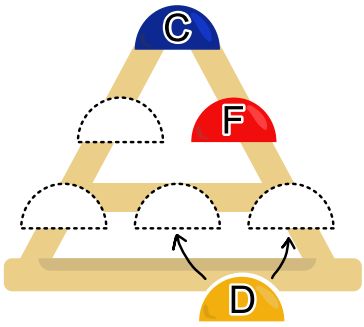
Darunter sind alle gewünschten, aber keine «verbotenen» Wege:

- Der Weg C → E → A ist ein Weg mit den drei Glocken A, C, E; der Weg C → E → B ist ein Weg mit den beiden Glocken B und E; und der Weg C → F → D ist auch enthalten.
- Die Murmel kann weder auf einem Weg mit den Tönen B und F noch auf dem Weg B → A → D laufen.

Wie lässt sich diese Befestigung der Glocken bestimmen? Wir müssen dafür sorgen, dass die Murmel auf allen gewünschten Wegen, aber nicht auf einem verbotenen Weg laufen kann. Dazu fügen wir nach und nach die gewünschten Wege in die Murmelbahn ein und vermeiden dann die verbotenen Wege.

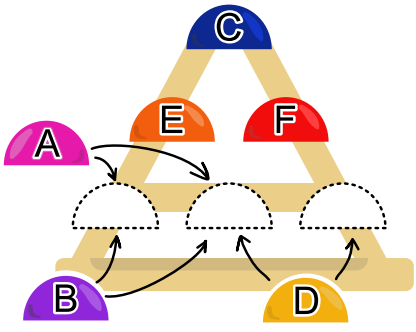
Schritt 1: Weg C → F → D einfügen

C muss oben, F in der zweiten Reihe und D in der dritten Reihe befestigt werden. Wir entscheiden uns für F für die rechte Stelle; dann muss D an die mittlere oder rechte Stelle der dritten Reihe.



Schritt 2: Wege mit A, C, E bzw. B, E einfügen

C ist bereits oben befestigt. Darunter muss E sowohl in einem Weg mit A als auch in einem mit B vorkommen. Dazu muss E oberhalb von A oder B befestigt werden. Also müssen E in der zweiten Reihe links sowie A und B in der dritten Reihe links oder mittig befestigt werden.



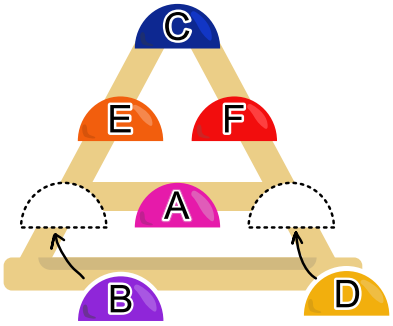
 © Informatik-Biber 2026, SVIA

Schritt 3: Weg B → A → D vermeiden

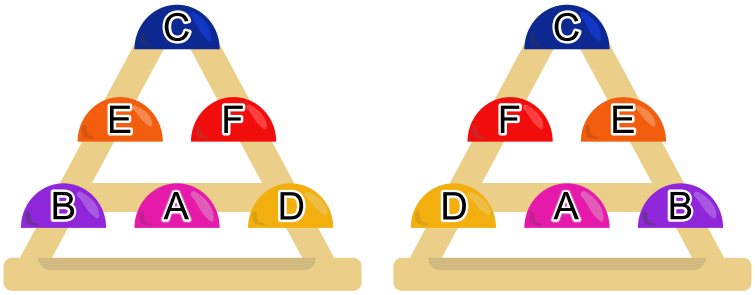
Weil C bereits oben befestigt ist, kann es keinen Weg geben, der mit B beginnt.

Schritt 4: alle Wege mit B und F vermeiden

Weil C bereits ganz oben und F in der zweiten Reihe befestigt ist, muss nur vermieden werden, dass die Murmel von F auf B rollen kann. Also müssen in der dritten Reihe B links, A in der Mitte und D rechts befestigt werden.



Insgesamt gibt es 2 richtige Lösungen, da die Anordnung der Glocken auch gespiegelt die selben 4 Murmelwege ergeben.



Dies ist Informatik!

Du hast die Glocken so befestigt und zu einer Murmelbahn kombiniert, dass es für die Murmel nur gewünschte Wege und keine verbotenen Wege gibt. Deine Kombination der Stellen für die verschiedenen Glocken erfüllt (engl.: satisfies) also Martas Bedingungen (engl.: constraints). In der Informatik kommen solche *constraint satisfaction problems* (CSPs) sehr häufig vor, zum Beispiel bei der Erstellung von Stundenplänen in einer Schule.

Dabei geht es üblicherweise darum, einzelne Elemente so auszuwählen, dass deren Kombination eine Menge von vorgegebenen Bedingungen erfüllt und damit eine Lösung darstellt. Ein einfacher Algorithmus für CSPs wäre also, solange verschiedene Kombinationen von Elementen auszuprobieren, bis eine Lösung gefunden ist. Wenn es aber viele Kombinationen gibt, kann das so viel Arbeit sein, dass es selbst für einen Computer sehr aufwändig wäre, eine Lösung so zu berechnen.

Eine bessere Strategie ist *Backtracking*: Dabei baut man mögliche Lösungen Schritt für Schritt auf und nimmt den letzten Schritt zurück, wenn ein Widerspruch auftaucht. So werden gleich viele ungeeignete Möglichkeiten ausgeschlossen. Auch in dieser Biberaufgabe lässt sich diese Strategie anwenden: Man wählt nach und nach für eine Stelle eine Glocke aus, so dass alle Bedingungen erfüllt sind. Wenn das an einer Stelle nicht möglich ist, geht man zur vorigen Stelle zurück und tauscht die dafür gewählte Glocke gegen eine andere aus.

Stichwörter und Webseiten

- Constraint satisfaction problems (CSP): `https://de.wikipedia.org/wiki/Constraint-Satisfaction-Problem`
- Backtracking: `https://de.wikipedia.org/wiki/Backtracking`

 © Informatik-Biber 2026, SVIA

10. Münzspiel

In einem Spiel gibt es

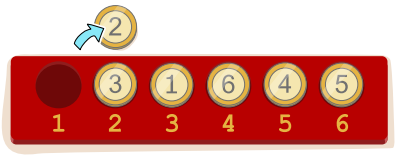
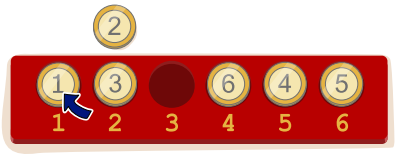
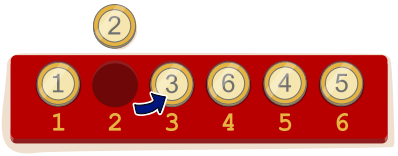
- 6 Münzen mit den Nummern 1 bis 6 und
- ein Spielbrett mit Plätzen 1 bis 6.

Zu Beginn des Spiels werden die Münzen beliebig auf die Plätze gelegt, zum Beispiel so:



Ziel des Spiels ist es, in möglichst wenigen Runden die Münzen auf die richtigen Plätze zu legen: Münze 1 auf Platz 1, Münze 2 auf Platz 2 und so weiter.

Eine Runde besteht aus den folgenden Schritten:

	Schritt	**Beispiel**
1	Nimm die erste Münze von links, die am falschen Platz liegt. Lege sie nach oben, über ihren richtigen Platz. Ein Platz wird frei.	
2	Lege so lange die richtige Münze auf den aktuell freien Platz (und mache dadurch einen anderen Platz frei), bis der Platz unter der nach oben gelegten Münze frei wird.	 
3	Lege die Münze von oben nun auf den freien, richtigen Platz. Damit ist die Runde zu Ende, und alle Plätze sind wieder belegt.	

Die Runde wird so lange wiederholt, bis alle Münzen am richtigen Platz liegen.

Nun beginnt das Spiel von vorne. Die Münzen werden wieder beliebig auf die Plätze gelegt, aber so, dass keine Münze auf dem richtigen Platz liegt.

*Wie viele Runden sind dann **mindestens** nötig, um jede Münze auf den richtigen Platz zu legen? Wie viele Runden sind dann **höchstens** dazu nötig?*

A) mindestens 1 Runde, höchstens 6 Runden

B) mindestens 2 Runden, höchstens 3 Runden

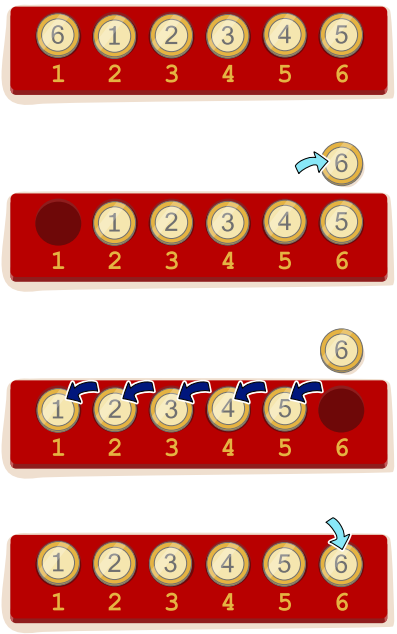
C) mindestens 1 Runde, höchstens 3 Runden

D) mindestens 2 Runden, höchstens 6 Runden

Lösung

Antwort C ist richtig.

Zuerst überlegen wir, wie viele Runden *mindestens* nötig sind, um jede Münze auf den richtigen Platz zu legen.



Da am Anfang keine Münze auf dem richtigen Platz liegt, ist sicher mindestens eine Runde nötig. Aber kann man mit einer Runde auskommen?

Links sieht man, dass das tatsächlich möglich ist. Alle Münzen liegen genau einen Platz rechts neben dem richtigen Platz; mit Ausnahme der Münze 6, die auf Platz 1 ganz links liegt.

- Im Runden-Schritt 1 wird Münze 6 nach oben gelegt, über Platz 6.
- In Schritt 2 können alle anderen Münzen um einen Platz nach links, also auf den richtigen Platz gelegt werden. Dann ist Platz 6 frei, so dass
- in Schritt 3 auch die Münze 6 auf den richtigen Platz gelegt werden kann.

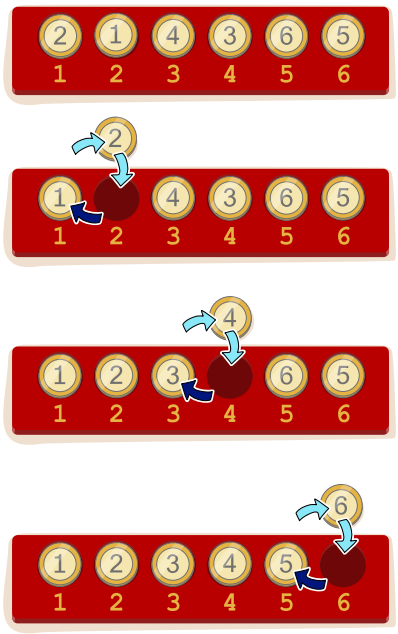
Und wie viele Runden sind *höchstens* nötig?

Im obigen Beispiel für eine Runde bilden alle sechs Münzen eine besondere Gruppe: Jede Münze liegt auf einem Platz, der für eine andere Münze aus dieser Gruppe richtig ist.

Die Münzen einer solchen Gruppe bilden einen *Zyklus*; im obigen Beispiel: $6 \to 1$ (bedeutet: Münze 6 liegt auf dem Platz von Münze 1) $\to 2 \to 3 \to 4 \to 5 \to 6$.

Im Beispiel aus der Aufgabenstellung gibt es zwei Zyklen: $2 \to 1 \to 3 \to 2$ und $6 \to 4 \to 5 \to 6$.

 © Informatik-Biber 2026, SVIA



In jeder Runde des Spiels wird genau ein Zyklus aufgelöst, werden also alle seine Münzen auf die richtigen Plätze gelegt.

Da ein Zyklus aus mindestens zwei Münzen besteht, können sechs Münzen höchstens drei Zyklen enthalten. In diesem für das Spiel schlechtesten Fall sind also höchstens drei Runden nötig, um jede Münze auf den richtigen Platz zu legen.

Links sieht man, wie die Münz-Belegung mit den Zyklen $2 \to 1 \to 2$, $4 \to 3 \to 4$ und $6 \to 5 \to 6$ in drei Runden sortiert wird.

Dies ist Informatik!

Beim Münzspiel in dieser Biberaufgabe werden die Münzen *sortiert*. Sortieren bedeutet, Daten nach einer bestimmten Vorschrift anzuordnen. Computerprogramme sortieren sehr häufig und ordnen zum Beispiel Zahlen nach dem Wert, Dateien nach dem Namen oder Daten im Arbeitsspeicher nach dem Zeitpunkt des letzten Zugriffs an. Sie tun das, weil der Zugriff auf sortierte Daten in der Regel sehr viel schneller ist als der auf unsortierte Daten. In allen obigen Beispielen ist die Grundlage für das Sortieren eine *Ordnung*, die Vergleiche zwischen Objekten erlaubt. Die Programme können also die Werte zweier Objekte miteinander vergleichen, um die Position der Objekte in der sortierten Anordnung festzulegen. Diese Art des Sortierens heisst deswegen *vergleichsbasiert*.

Im Münzspiel gibt jede Münze selbst an, wo sie hingehört, nämlich auf den Platz mit der gleichen Nummer wie die Münze. So lassen sich mit dem Münzspiel Münzen bzw. Zahlen sortieren, ohne dass Vergleiche der Werte nötig sind. Diese Art des Sortierens heisst deswegen *nicht-vergleichsbasiert*.

Weil Sortieren in Informatiksystemen so häufig passiert, ist es wichtig, dass Sortieralgorithmen möglichst wenig Ressourcen verbrauchen. Unser Münzspiel-Sortieralgorithmus zum Beispiel benötigt zur Ablage der Münzen fast nur die ohnehin nötigen Plätze auf dem Spielbrett. Abgesehen von dem zusätzlichen Platz für die Münze, die in Schritt 1 vorübergehend über ihren richtigen Platz gelegt wird, werden für alle anderen Münz-Bewegungen nur die vorhandenen Plätze benötigt. In der Informatik werden Sortieralgorithmen, die im Wesentlichen mit dem ohnehin für die zu sortierenden Objekte benötigten Speicherplatz auskommen, als «In-place-Algorithmen» bezeichnet, was bedeutet, dass sie die Objekte «an ihrem Platz» umordnen.

Ausser für den *Speicherbedarf* eines Algorithmus interessiert sich die Informatik insbesondere für seine *Laufzeit*: die Anzahl der Schritte, die ein Algorithmus abhängig von der Grösse der Eingabe machen muss. Die Laufzeit des Münzspiel-Algorithmus lässt sich anhand der Anzahl der Münzbewegungen messen. Jede Münze gehört zu genau einem Zyklus und wird deshalb nur in einer Runde bewegt,

genau einmal auf den richtigen Platz. Ausserdem wird in jeder Runde die erste Münze bewegt, über den richtigen Platz. Bei n Münzen und m Runden ist die Anzahl Münzbewegungen also $n + m$. Und weil die Anzahl der Runden höchstens $n/2$ sein kann, weil im schlechtesten Fall Zweier-Zyklen aufzulösen sind, ergibt sich eine maximale Anzahl von $n + n/2 = 3/2n$ Bewegungen. Eine solche Laufzeit, die im Wesentlichen der Eingabegrösse n entspricht, heisst in der Informatik *linear*. Nur nicht-vergleichsbasierte Sortieralgorithmen können mit linearer Laufzeit auskommen, sind aber, im Gegensatz zu vergleichsbasierten Algorithmen, nur unter bestimmten Bedingungen anwendbar.

Stichwörter und Webseiten

- Sortierverfahren: `https://de.wikipedia.org/wiki/Sortierverfahren`
- O-Notation: `https://de.wikipedia.org/wiki/Landau-Symbole`

 © Informatik-Biber 2026, SVIA

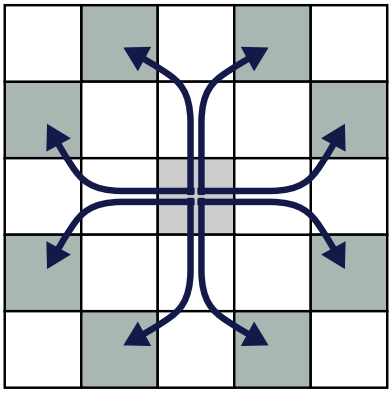
11. Sprung-Spiel



Ein Pferd springt über ein Spielbrett mit 25 Feldern.
Ein Sprung geht entweder über:

- 2 Felder horizontal und 1 vertikal oder
- 2 vertikal und 1 horizontal.

Das Bild zeigt alle möglichen Sprünge vom mittleren Feld aus.



Nun startet das Pferd auf Feld 21 und besucht mit mehreren Sprüngen ausschliesslich die nummerierten Felder. Dabei besucht es jedes nummerierte Feld genau einmal.

1		3		
	7			10
	12	13	14	
16		18		
21				25

In welcher Reihenfolge hat das Pferd die Felder besucht?

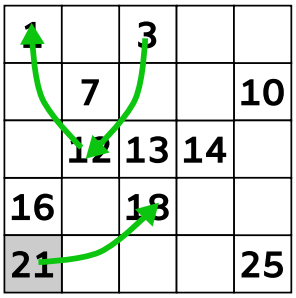
Lösung

So ist es richtig:

21	18	25	14	7	16	13	10	3	12	1

Aus den Regeln und dem Aufbau des Spielbretts lässt sich diese Sprungfolge ableiten:

- Das Pferd muss Feld 1 als letztes besuchen: Nur von Feld 12 aus kann es auf Feld 1 springen, und von dort aus kann es nur zurück zu Feld 12 - aber das ist nicht erlaubt. Somit sind die letzten zwei Felder 12, dann 1. (12→1)
- Auf 12 kann das Pferd von 3 oder 21 aus springen. Das Start-Feld 21 kann aber nicht das drittletzte Feld der Sprungfolge sein; das ist also Feld 3. (3→12→1)
- Vom Start-Feld 21 aus kann das Pferd auf die Felder 12 und 18 springen. Feld 12 ist aber schon das zweitletzte Feld in der Sprungfolge; das zweite Feld kann also nur Feld 18 sein (21→18).

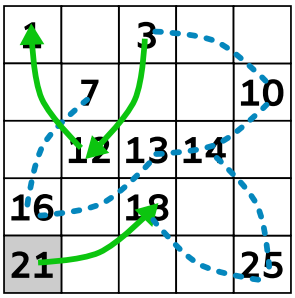


Wir kennen nun also den Beginn (21→18) und das Ende (3→12→1) der Sprungfolge (grün).

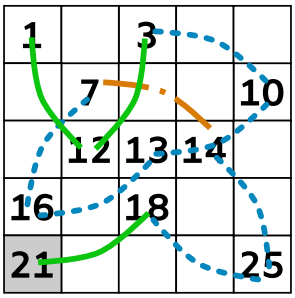
Einige weitere Überlegungen helfen die Route zu vervollständigen:

- Von den Feldern 25, 16 und 10 aus kann das Pferd jeweils zu genau zwei anderen Feldern springen.

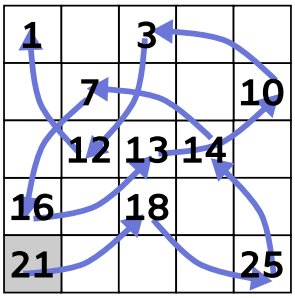
 © Informatik-Biber 2026, SVIA



Die Feldfolgen 7-16−13, 18-25−14, 13-10−3 müssen also Teile der Sprungfolge (blau) sein, nur die Richtung ist noch nicht klar.



Nur für die Felder 7 und 14 fehlt noch jeweils ein Sprung. Weil das Pferd von einem zum anderen Feld springen kann, ist auch 7−14 ein Teil der Sprungfolge (orange).

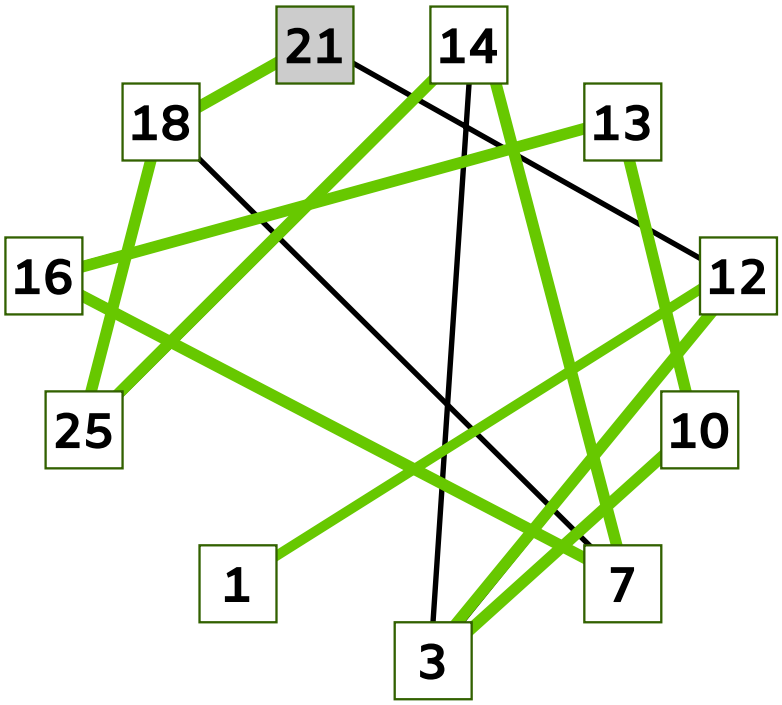


Nun sind wir fertig: Jedes nummerierte Feld wird genau ein Mal besucht. Die Sprungrichtungen innerhalb der Teilstücke ergeben sich, indem man von 21 startet und den Linien folgt.

Dies ist Informatik!

Viele Fragestellungen in der Informatik lassen sich mit Hilfe von *Graphen* modellieren. Ein Graph besteht aus Knoten, die Objekte repräsentieren, und Kanten, die Verbindungen zwischen den Objekten beschreiben. Die Objekte können alle möglichen Dinge sein, und so können beispielsweise Strassennetze, Flugverbindungen oder Kommunikationsnetze als Graphen dargestellt werden.

Auch in dieser Biberaufgabe ist ein Graph nützlich: Wenn wir die Felder als Knoten verstehen, stellen die Kanten die Möglichkeit dar, von einem Feld zu einem anderen zu springen. Hier ist der Pferd-Sprung-Graph; Kanten, die Teil der Sprungfolge sind, sind grün gezeichnet:



Eine wichtige Klasse von Graphenproblemen besteht darin, einen Weg zu finden, der bestimmte Bedingungen erfüllt. Ein bekanntes Beispiel ist die Suche nach einem *Hamilton-Pfad:* Das ist ein Weg entlang von Kanten des Graphen, der jeden Knoten genau einmal besucht. Die in dieser Biberaufgabe gesuchte Sprungfolge ist also ein Hamilton-Pfad.

Hamilton-Pfade haben besonders dann interessante Anwendungen, wenn die Kanten des Graphen Gewichte haben, etwa wenn sie für Enfernungen zwischen Orten stehen. Wer dann die kürzeste Tour durch alle Orte sucht, sucht im Orte-Entfernungen-Graph den Hamilton-Pfad (oder genauer: den Hamilton-Kreis, da eine Tour zum Ausgangsort zurückführt) mit geringstem Gesamtgewicht der im Pfad vorkommenden Kanten.

Obwohl die Frage «Gibt es in einem Graph einen Hamilton-Pfad?» einfach formuliert werden kann, gehört sie zu den klassischen schwierigen Problemen der Informatik. Für beliebige Graphen ist bis heute kein effizienter Algorithmus bekannt, der einen Hamilton-Pfad immer findet – oder nachweist, dass keiner existiert.

Stichwörter und Webseiten

- Hamilton-Pfad: `https://de.wikipedia.org/wiki/Hamiltonkreisproblem#Eigenschaften`

 © Informatik-Biber 2026, SVIA

12. Tacos für lau!

Tacos sind ein schnell zubereitetes Gericht aus Mexiko. Eine Taquería verspricht: «Wenn wir für eine Bestellung von höchstens 7 Tacos länger als 15 Minuten brauchen, ist sie kostenlos.»

Die Taquería hat drei Köche, die für die verschiedenen Taco-Sorten unterschiedlich lange brauchen (in Minuten ⏱):

Taco-Sorte	**Ana**	**Beto**	**Carlos**
Poulet	5	3	5
Fleisch	3	5	5
Vegetarisch	2	11	11

Bei einer neuen Bestellung werden die bestellten Tacos der Reihe nach den Köchen zugeteilt, und zwar nach diesen Regeln:

1. Wähle nur einen Koch, der bisher höchstens 9 Minuten braucht. Gibt es so einen Koch nicht, wähle nur nach Regel 2.
2. Wähle den Koch, der den Taco am schnellsten zubereiten kann. Bei Gleichstand den, der bisher am wenigsten Zeit braucht. Bei nochmal Gleichstand nach dem Alphabet.

Wenn alle Tacos zugeteilt sind, legen die Köche gleichzeitig los. Wenn dann ein Koch länger als 15 Minuten braucht, sind die Tacos für lau!

Finde eine Bestellung, für die die Taquería mehr als 15 Minuten braucht.

Scanne den QR-Code, um auf die interaktive Version dieser Übung zuzugreifen.

Lösung

Um eine kostenlose Bestellung zu erhalten, muss einer der drei Köche für «seine» Tacos mehr als 15 Minuten brauchen. Das kann man insbesondere mit dieser Strategie erreichen: Weise einem Koch, der bisher höchstens 9 Minuten braucht, einen Taco zu, für den er mehr als 6 Minuten braucht.

Man sieht, dass das nur bei Beto und Carlos geht, mit einem vegetarischen Taco (11 Minuten). Dazu muss Ana für ihre bisherige Zuteilung schon mehr als 9 Minuten brauchen, sonst würde dieser Taco nach Regel 2 ihr zugeteilt.

Also beschäftigen wir Ana mit z.B. 4 mal 🥩 (12 Minuten). Mit weiteren 2 mal 🥩 bekommen Beto und Carlos je 5 Minuten. Dann braucht man nur noch einmal 🥬 zu bestellen. Der wird dann nämlich Beto zugeteilt, so dass er für seine Zuteilung 16 Minuten braucht - die Tacos 🌮 sind für lau!

Es gibt 11 verschiedene Möglichkeiten, am Anfang Tacos zu bestellen, die alle Ana zugeteilt werden und für die sie mehr als 9 Minuten braucht (🍗 kommt darin nicht vor, weil Beto und Carlos bei dieser Sorte schneller sind als Ana):

- 🥩🥩🥩🥩
- 🥩🥩🥩🥬 (in 4 verschiedenen Reihenfolgen)
- 🥩🥩🥬🥬 (in 6 verschiedenen Reihenfolgen)

Danach müssen wir entweder Beto oder Carlos mit Vegi auf mindestens 16 Minuten bringen. Dafür gibt es 4 Möglichkeiten:

- 🥩🥩🥬
- 🥬🥩🥬
- 🥬🍗🥬
- 🥩🥬🥬

Es gibt also $11 \times 4 = 44$ Möglichkeiten, die Taquería zu überlasten und kostenlos zu essen. Bei insgesamt 2.187 möglichen Bestellungen ist die Chance mit ca. 2 % aber nicht allzu gross.

Dies ist Informatik!

Die Regeln der Taquería in dieser Biberaufgabe haben das Ziel, die Bestellungen schnell abzuarbeiten. Allerdings gibt es bestimmte Bestellungen, für die diese Regeln kein optimales Ergebnis liefern. Nun kann man versuchen, wie ein Angreifer diese Tatsache auszunutzen, um kostenlose Tacos zu bekommen. Die Taquería wiederum könnte dann die Regeln anpassen, um weitere Angriffe abzuwehren. Zum Beispiel könnte sie eine weitere Regel einbauen: Wenn ein Taco die Arbeitszeit des ausgewählten Kochs über 15 Minuten steigert, dann nimm einen anderen Koch. Aber hält das *alle* Angriffe ab? Leider nicht! Fallen dir Regeln ein, die absolut sicher sind?

In der Informatik ist die Strategie, für einen Algorithmus eine möglichst schwierig zu bearbeitende Eingabe zu finden, auch als *adversarial attack* bekannt. Man nimmt dabei die Rolle eines Angreifers ein und versucht einen Verteidiger zu etwas zu bringen, was er eigentlich nicht möchte. Eine besondere

 © Informatik-Biber 2026, SVIA

Rolle spielen adversarial attacks für KI-Systeme: Mit gezielt manipulierten Trainingsdaten können bei künstlichen neuronalen Netzen Fehlverhalten ausgelöst werden.

Bei Informatiksystemen bzw. den sie steuernden Programmen kann es also Regeln bzw. Algorithmen geben, die anfällig für gezielte Angriffe sind. Im harmlosesten Fall kann dies zu einem langsameren Ablauf führen, oder das Programm stürzt ab. Schlimmer wäre es, wenn Angreifer eine Anfälligkeit ausnutzen, um unberechtigt auf Daten zuzugreifen oder gar beliebige Programme auszuführen. Daher führen Sicherheitsexperten bewusst solche Angriffe selber durch, um mögliche Fehler zu identifizieren und rechtzeitig zu beheben. Das nennt man *Adversarial Testing*.

Stichwörter und Webseiten

- Lastverteilung: `https://de.wikipedia.org/wiki/Lastverteilung_(Informatik)`
- Adversarial Attack: `https://de.wikipedia.org/wiki/Adversarial_Attack`
- Adversarial Testing: `https://www.security-insider.de/was-ist-adversarial-testing-a-4d8f9ad661ac79548140fe547cfd5152/`
- Softwaretest / Sicherheitstest: `https://de.wikipedia.org/wiki/Softwaretest`, `https://de.wikipedia.org/wiki/Sicherheitstest_(Software)`

© Informatik-Biber 2026, SVIA

13. Versteckspiel

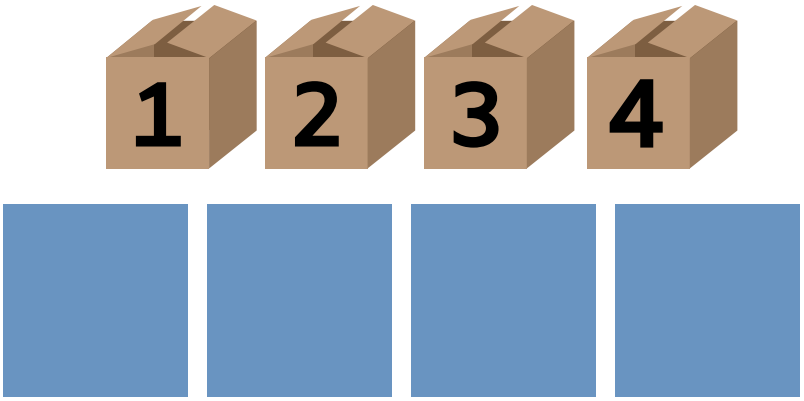
Ben versteckt sich in einer von vier nummerierten Kisten, z.B. in Kiste 2.



Seine Freundin Anja will ihn finden und schaut jeweils in genau einer Kiste nach. Findet sie Ben dort nicht, bewegt sich Ben unbemerkt in eine benachbarte Kiste (eine Kiste nach links oder rechts), bevor Anja erneut in einer Kiste nachschaut.

Anja überlegt: Wenn sie in der richtigen Reihenfolge in die Kisten schaut, muss sie höchstens 4-mal nachschauen, um Ben auf jeden Fall zu finden, egal wie er sich bewegt.

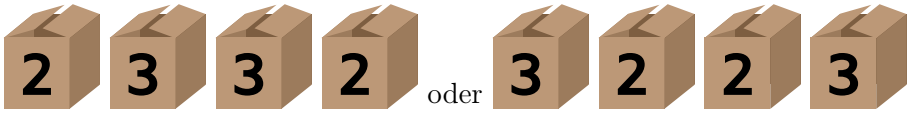
Gib eine solche Reihenfolge an!



Die Kisten können in die einzelnen Felder gezogen werden. Die Kisten können mehrfach verwendet werden. Man kann eine Kiste in ein anderes Feld schieben, dadurch werden die Kisten getauscht.

Lösung

Um Ben auf jeden Fall mit 4-mal Nachschauen zu finden, muss Anja in einer dieser beiden Reihenfolgen in die Kisten schauen:

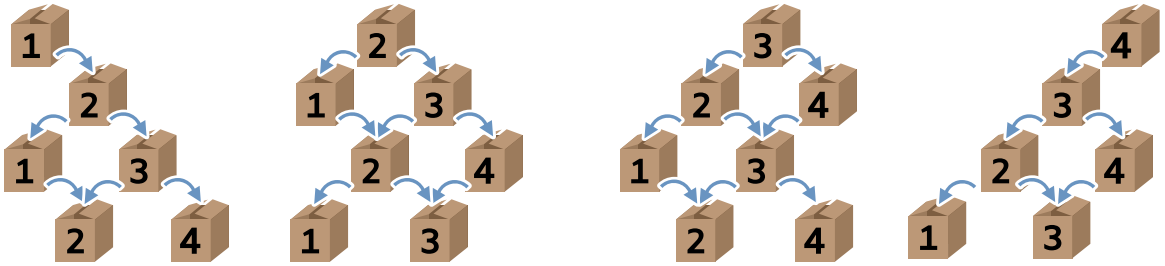


Wir zeigen, dass diese Antworten richtig sind. Dabei bezeichnen wir Bens Bewegungen als *Zug* in diesem Versteckspiel.

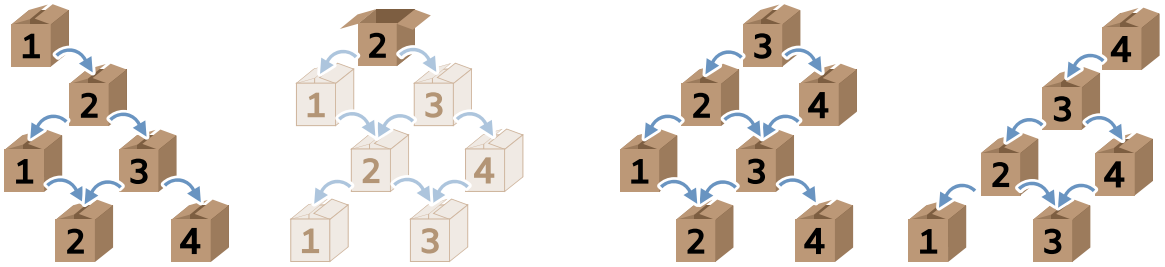
Am Anfang kann Ben seine «Start-Kiste» frei wählen. Danach hat Ben, wenn er

- in Kiste 2 oder 3 sitzt, jeweils zwei Möglichkeiten für den nächsten Zug: entweder geht er nach links oder nach rechts;
- in Kiste 1 oder 4 sitzt, jeweils eine Möglichkeit für den nächsten Zug: er geht nach rechts (von Kiste 1 aus) bzw. nach links (von Kiste 4 aus).

Ben macht seinen Zug, nachdem Anja in einer Kiste nachgeschaut und ihn nicht gefunden hat. Wenn Anja ihn mit (höchstens) 4-mal Nachschauen finden soll, kann er bis dahin höchstens drei Züge machen. Deshalb schauen wir uns alle möglichen Folgen von drei Zügen an, und zwar von jeder möglichen Start-Kiste (1–4) aus:

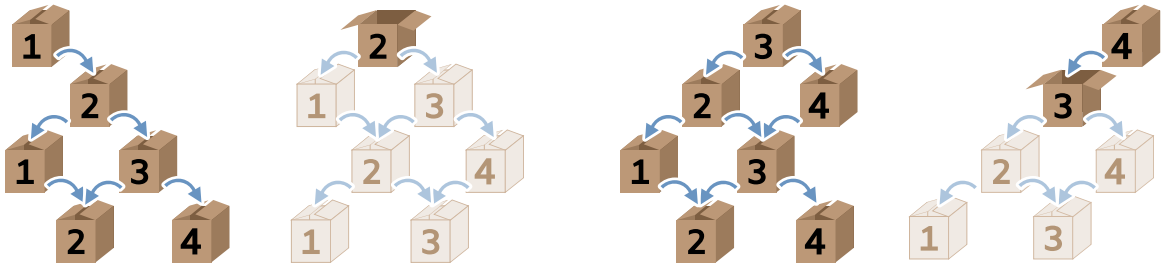


Was passiert, wenn Anja in der Reihenfolge **2-3-3-2** in die Kisten schaut? Zuerst schaut Anja in Kiste 2. Wenn sie Ben dort findet, ist Kiste 2 seine Start-Kiste, und er kann die von dort ausgehenden Züge nicht mehr machen.

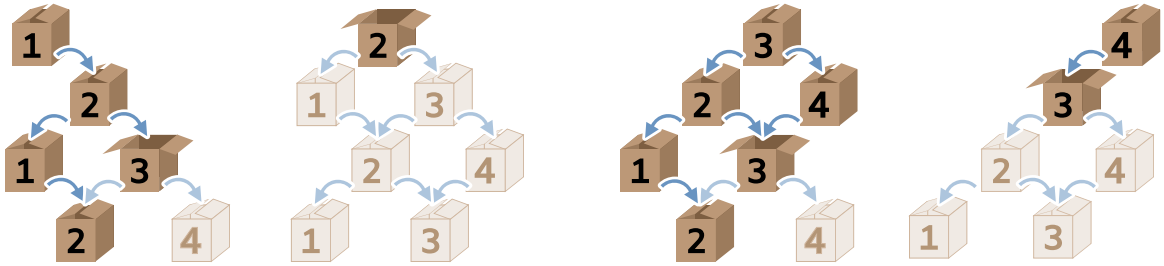


Dann schaut Anja in Kiste 3. Wenn sie Ben dort findet, ist das seine zweite Kiste. Auf der zweiten Ebene der Zugfolgen kommt Kiste 3 noch ganz rechts vor. Die von dort ausgehenden Züge kann Ben nun nicht mehr machen, und wir brauchen sie nicht weiter zu betrachten.

 © Informatik-Biber 2026, SVIA



Als Drittes schaut Anja wieder in Kiste 3. Diese kommt in den Zugfolgen noch an zwei Stellen auf der dritten Ebene vor. Die von dort ausgehenden Züge kann Ben nun nicht mehr machen.



Auf der vierten Ebene der Zugfolgen kommt jetzt nur noch Kiste 2 vor. Also kann nur Kiste 2 Bens vierte Kiste sein, falls Anja ihn bis dahin noch nicht gefunden hat. Wenn Anja jetzt in Kiste 2 schaut, muss Ben in dieser Kiste sein.

Für die Antwort **3–2–2–3** gilt die gleiche Begründung spiegelbildlich, denn sowohl die Zugfolgen von den Start-Kisten 1 und 4 aus als auch die von 2 und 3 aus sind spiegelbildlich zueinander.

Dies ist Informatik!

In der Informatik werden mögliche Abläufe, von einem oder mehreren Startpunkten aus, oft als Zustandsgraphen oder Bäume dargestellt. Das sind Datenstrukturen, die zum einen aus Knoten und zum anderen aus Kanten zwischen jeweils zwei Knoten bestehen. Jeder Knoten steht für einen möglichen Zustand. Jede Kante steht für einen möglichen nächsten Schritt von einem zu einem nächsten Zustand und hat deshalb eine Richtung.

Solche Darstellungen möglicher Abläufe kann man u.a. dann verwenden, wenn es zwei Akteure gibt, die gegensätzliche Ziele verfolgen, wie in dieser Biberaufgabe: Ben will sich vor Anja verstecken, Anja will Ben finden. Mit Hilfe der Graphen für die Zugfolgen konnten wir aus Sicht von Akteur Anja systematisch alle Möglichkeiten des Verhaltens ihres «Gegners» Ben untersuchen und so eine Strategie finden, die für Anja garantiert zum Erfolg führt. Solche Vorgehensweisen finden unter anderem in der *Spieltheorie*, bei der Planung von Algorithmen und in der Robotik Verwendung.

Stichwörter und Webseiten

- Binärbaum: `https://de.wikipedia.org/wiki/Binärbaum`
- Spieltheorie: `https://de.wikipedia.org/wiki/Spieltheorie`

 © Informatik-Biber 2026, SVIA

14. Tinto Azul

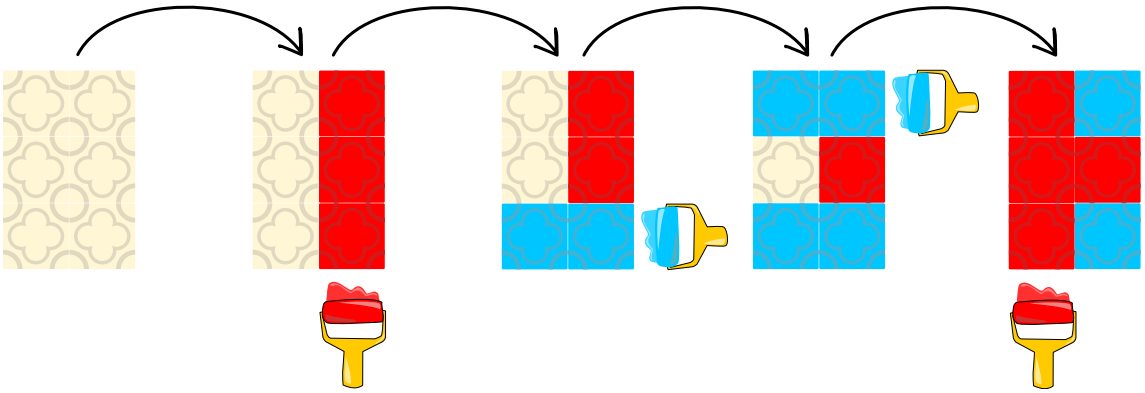
Malermeister Tinto Azul ist sehr spezialisiert: Er streicht nur Böden an, die aus quadratischen Fliesen bestehen.

Dabei macht er nur diese zwei Sachen:

- eine ganze Zeile blau anstreichen: 
- eine ganze Spalte rot anstreichen: 

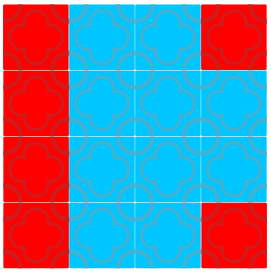
Tinto kann Farben auch wieder überstreichen. Am Ende hat er ein blau-rotes Muster auf den Boden gemalt.

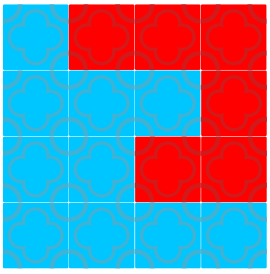
Ein Beispiel: Auf einen Boden mit 3 Zeilen und 2 Spalten kann Tinto das Muster ganz rechts malen, und zwar so:

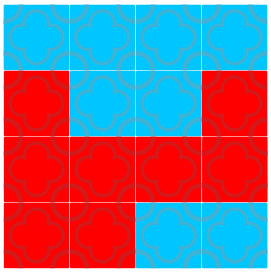


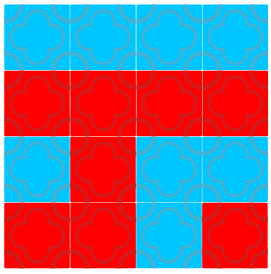
Nun soll Tinto einen Boden mit 4 Zeilen und 4 Spalten anstreichen.

Welches Muster kann er nicht malen?

A) 

B) 

C) 

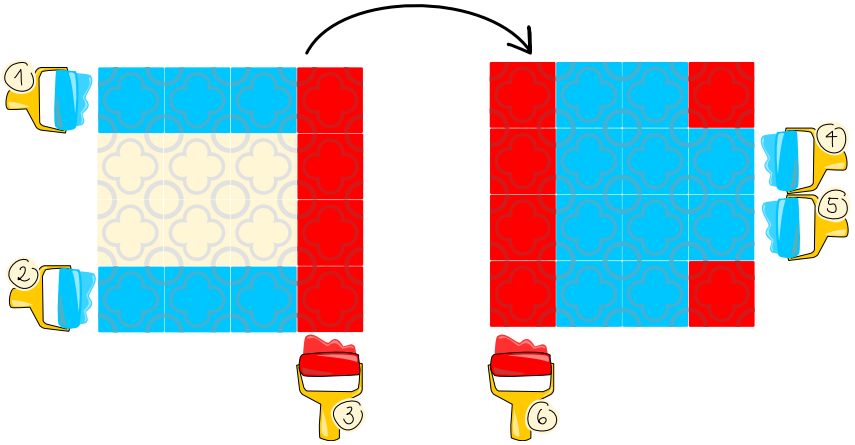
D) 

Lösung

Antwort C ist richtig.

Für die Antworten A, B und D geben wir an, wie Tinto das jeweilige Muster malen kann.

Das Muster von Antwort A kann Tinto in sechs Schritten so malen:

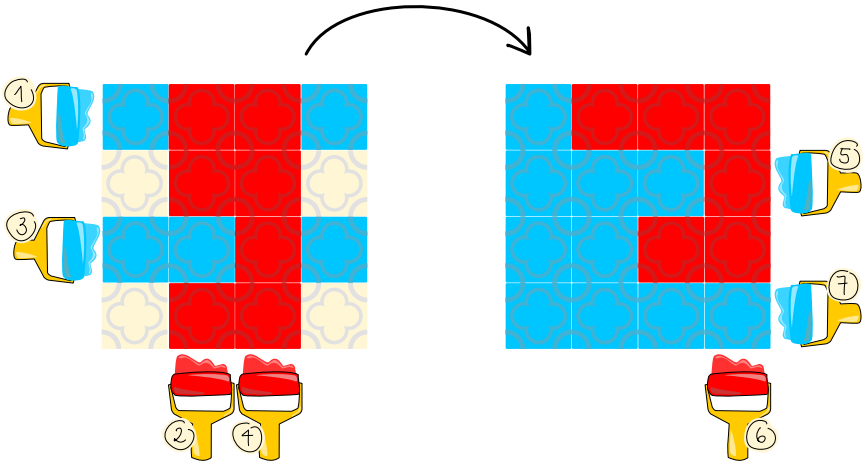


Man kann das herausfinden, indem man sozusagen «rückwärts denkt»:

- Im letzten Schritt (6) hat Tinto die linke Spalte rot angestrichen. Denn diese Spalte ist vollständig rot angestrichen und an keiner Stelle blau überstrichen.
- In zwei Schritten davor (4, 5) hat Tinto die beiden mittleren Zeilen blau angestrichen. Diese beiden Zeilen werden danach nur in der linken Spalte rot überstrichen (Schritt 6).
- Im Schritt davor (3) hat Tinto die rechte Spalte rot angestrichen. Ihre mittleren beiden Felder hat er später überstrichen (Schritte 4, 5), aber die Felder oben und unten nicht.
- Auch deshalb muss Tinto zuerst in zwei Schritten (1, 2) die obere und die untere Zeile blau angestrichen haben.

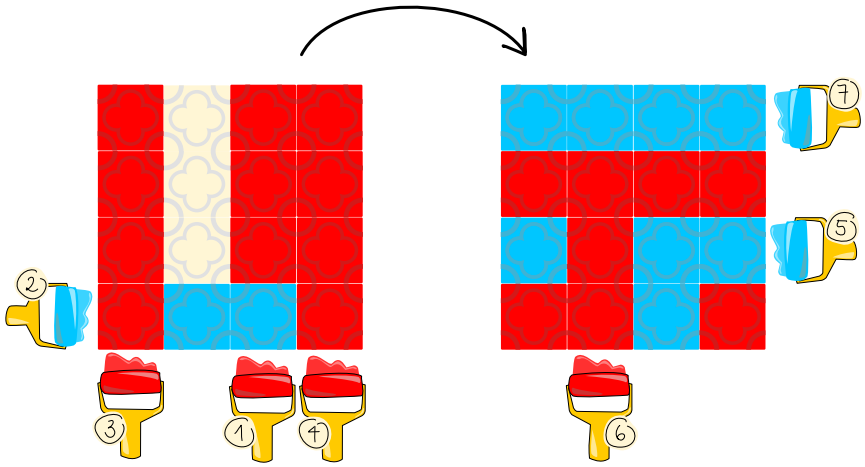
Mit ähnlichen Überlegungen finden wir heraus, wie Tinto die Muster aus den Antworten B und D malen kann:

Boden B:



 © Informatik-Biber 2026, SVIA

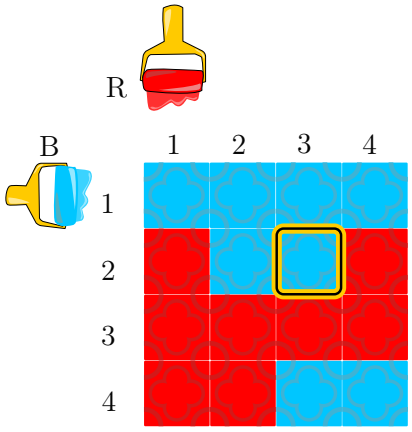
Boden D:



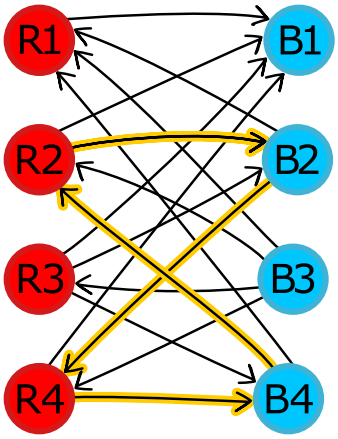
Aber warum kann Tinto das Muster in Antwort C nicht malen? Auch hier könnten wir versuchen, rückwärts zu denken. Die Argumentation gestaltet sich allerdings schwierig. Wir brauchen eine andere Perspektive und nutzen die Tatsache, dass das Muster aus einzelnen Quadraten besteht, die in 4 Zeilen und 4 Spalten angeordnet sind. Wir geben den Zeilen und Spalten jeweils Nummern von 1 bis 4. Tintos Färbeoperationen bezeichnen wir so:

- eine Zeile blau  anstreichen: B1 bis B4
- eine Spalte rot  anstreichen: R1 bis R4

Das Quadrat (x,y) in Zeile x und Spalte y kann also durch Bx und Ry eine Farbe bekommen.



An der Farbe des Quadrats (x,y) kann man sehen, welche dieser beiden Färbeoperationen Tinto zuletzt ausgeführt hat: Ist das Quadrat blau, kam Ry vor Bx, ansonsten umgekehrt. Für das blaue Quadrat (2,3) in Antwort C können wir zum Beispiel sagen, dass R3, falls überhaupt, vor B2 ausgeführt wurde.



Wenn wir für alle Quadrate diese Reihenfolge bestimmen, entsteht das Diagramm. Ein Pfeil von einer ersten zu einer zweiten Färbeoperation zeigt, dass die erste vor der zweiten ausgeführt wurde. Für das blaue Quadrat (2,3) führt also ein Pfeil von R3 zu B2.

Anhand der gelb markierten Pfeile entdecken wir einen Widerspruch: R2 kommt vor B2, B2 kommt vor R4, R4 kommt vor B4 und B4 soll, schließlich, vor R2 kommen. B4 kann aber nicht gleichzeitig vor und nach R2 ausgeführt werden. Genau deswegen kann Tinto das Muster in Antwort C nicht malen.

Dies ist Informatik!

Jedes Muster liefert uns einen Hinweis darauf, welche seiner beiden «Anstreich-Operationen» Tinto zuletzt ausgeführt hat. Die zwei Operationen sind Grundlage für das, was Informatikerinnen und Informatiker als *deterministischen Prozess* bezeichnen: Jede Anwendung einer Operation führt von einem *Zustand* (d.h. von einem Muster) auf festgelegte Weise zu einem anderen Zustand. Der *Endzustand* wird vollständig durch den Anfangszustand (hier: der ungestrichene Boden) und die Abfolge der Anstreich-Operationen bestimmt.

Wenn wir ein Muster betrachten und uns fragen, ob es durch eine Folge von Anstreich-Operationen gemalt worden sein könnte, versuchen wir aus dem Endzustand die Operations-Abfolge zu ermitteln, die zu diesem Endzustand geführt hat. Diese Art der Rückverfolgung kann bei der Suche nach Programmierfehlern (engl. *Debugging*) hilfreich sein: Man prüft in ganz ähnlicher Weise, ob und wie eine fehlerhafte Programmausgabe durch eine Abfolge der programmierten Operationen entstanden sein kann.

Stichwörter und Webseiten

- Debugging: `https://de.wikipedia.org/wiki/Debuggen`

 © Informatik-Biber 2026, SVIA

15. Petri-Ampeln

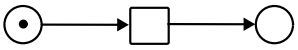
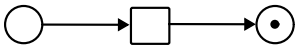
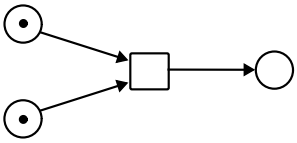
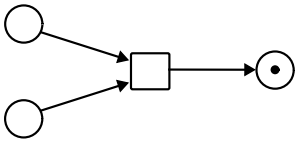
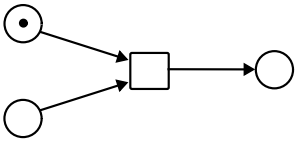
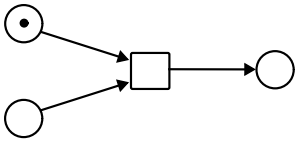
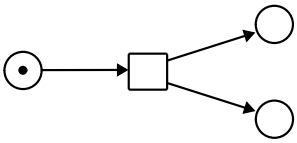
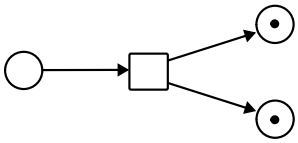
Mit Petri-Netzen kann man das Verhalten von Systemen beschreiben.

Ein einfaches Petri-Netz zeichnet man mit Kreisen und Quadraten. Ein Quadrat

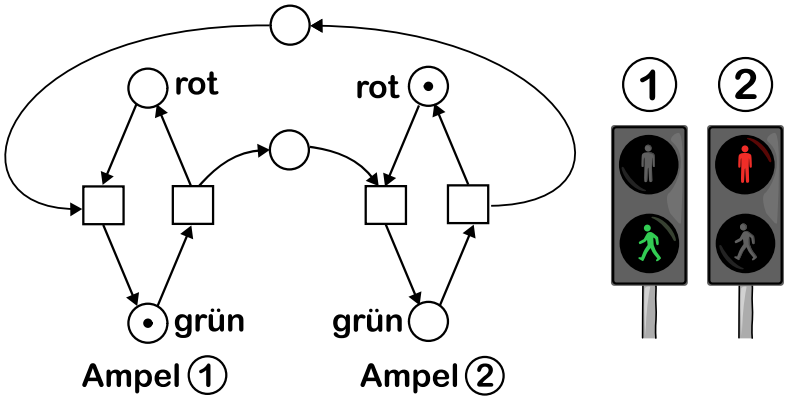
- hat *Eingabekreise* (erkennbar an einem Pfeil vom Kreis zum Quadrat) und
- *Ausgabekreise* (erkennbar an einem Pfeil vom Quadrat zum Kreis).

Ein Kreis kann eine *Marke* ⊙ enthalten. Enthält jeder Eingabekreis eines Quadrats eine Marke, dann *schaltet* das Quadrat. Anschliessend gilt: Alle Eingabekreise sind leer, und jeder Ausgabekreis enthält eine Marke.

Hier sind einige Beispiele. Nur beim dritten Beispiel schaltet das Quadrat *nicht*.

Vorher	Nachher	Quadrat geschaltet
		ja
		ja
		nein
		ja

Das folgende Petri-Netz beschreibt das Verhalten zweier Fussgängerampeln. Marken geben an, ob eine Ampel rot oder grün ist. Zu Beginn ist Ampel 1 grün und Ampel 2 rot.



Welche der folgenden Aussagen sind für das Netz und die zwei Ampeln sicher wahr?
Es können mehrere Aussagen wahr sein.

A) Es kann niemals passieren, dass Ampel 1 rot ist und gleichzeitig Ampel 2 grün ist.

B) Die Kreise in diesem Petri-Netz enthalten insgesamt niemals mehr als 3 Marken.

C) Die beiden Ampeln sind niemals gleichzeitig rot.

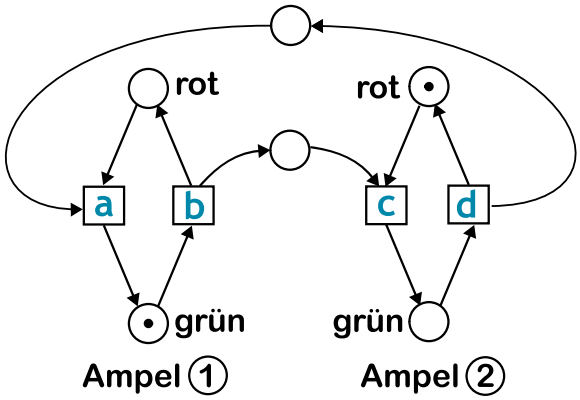
D) Die beiden Ampeln sind niemals gleichzeitig grün.

 © Informatik-Biber 2026, SVIA

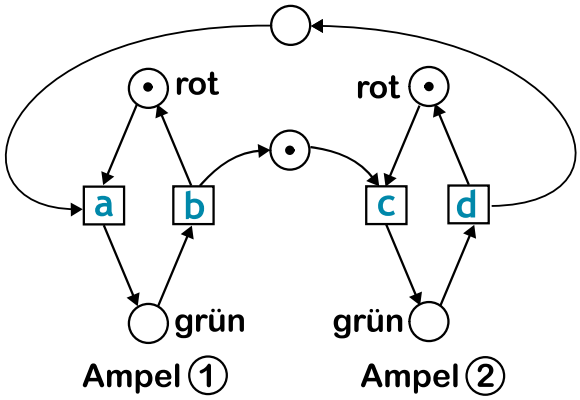
Lösung

Die Antworten B und D sind sicher wahr, die Antworten A und C sind hingegen falsch.

Im Ampel-Petri-Netz schalten nacheinander 4 Quadrate, bis das Netz wieder im Ausgangszustand ist. Wir schauen uns diese vier Schritte an. Zur leichteren Erklärung haben wir die Quadrate mit Buchstaben benannt.

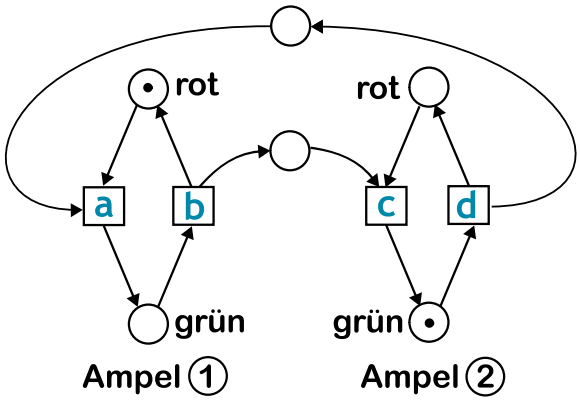


Schritt 1: Im Anfangszustand oben schaltet nur Quadrat b, denn es ist das einzige Quadrat, bei dem alle Eingabekreise eine Marke enthalten. Danach ist der Eingabekreis von b leer, und jeder seiner Ausgabekreise enthält eine Marke:



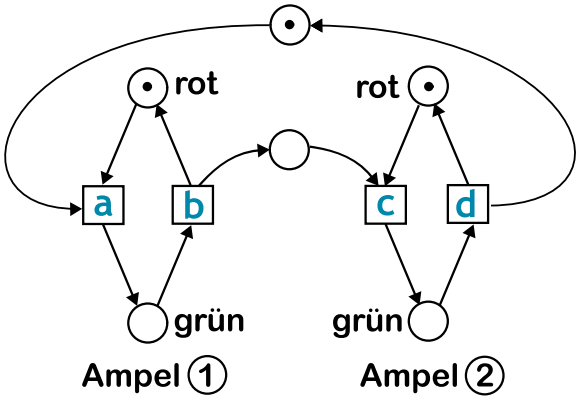
In diesem Zustand sind beide Ampeln rot. Also ist Antwort C falsch.

Schritt 2: Nur Quadrat c schaltet. Daraus ergibt sich dieser nächste Zustand:



In diesem Zustand ist Ampel 1 rot, während Ampel 2 grün ist. Also ist Antwort A falsch.

Schritt 3: Nur Quadrat d schaltet. Daraus ergibt sich dieser nächste Zustand:



Beide Ampeln sind rot. Auch hier sehen wir, dass Antwort C falsch ist.

Schritt 4: Nur Quadrat a schaltet und bringt das System wieder in seinen Ausgangszustand.

Diese vier Schritte können sich nur genau so wiederholen, und das gilt auch für die vier gezeigten Zustände. Darunter ist kein Zustand, in dem beide Ampeln gleichzeitig grün sind. Also ist Antwort D richtig. Ausserdem gibt es keinen Zustand, in dem die Kreise im Petri-Netz mehr als 3 Marken enthalten. Also ist auch Antwort B richtig.

Dies ist Informatik!

Diese Biberaufgabe verwendet eine vereinfachte Form von *Petri-Netzen*, einem formalen Modell aus der theoretischen Informatik. Petri-Netze wurden in den 1960er-Jahren von *Carl Adam Petri* eingeführt, der sie im Rahmen seiner Doktorarbeit entwickelte, um Prozesse zu untersuchen, bei denen mehrere Aktionen parallel ablaufen können.

Ein wesentlicher Vorteil von Petri-Netzen besteht darin, dass sich Nebenläufigkeit, Synchronisation und die gemeinsame Nutzung von Ressourcen anschaulich und übersichtlich darstellen lassen. Dadurch können Systemzustände und Zustandsübergänge präzise untersucht werden, während das Modell gleichzeitig leicht verständlich und intuitiv bleibt.

Petri-Netze werden in vielen praktischen Bereichen eingesetzt, beispielsweise bei Ampelsteuerungen, Kommunikationsprotokollen, Produktions- und Arbeitsabläufen, Betriebssystemen und verteilten Systemen. Diese Aufgabe zeigt, wie sich mit Petri-Netzen reale Systeme modellieren und deren Sicherheitseigenschaften überprüfen lassen – zum Beispiel die Eigenschaft, dass zwei Ampeln niemals gleichzeitig grün sind.

Stichwörter und Webseiten

Petri-Netz: `https://de.wikipedia.org/wiki/Petri-Netz`
Petri-Netz: `https://edutechwiki.unige.ch/en/Petri_net`

 © Informatik-Biber 2026, SVIA

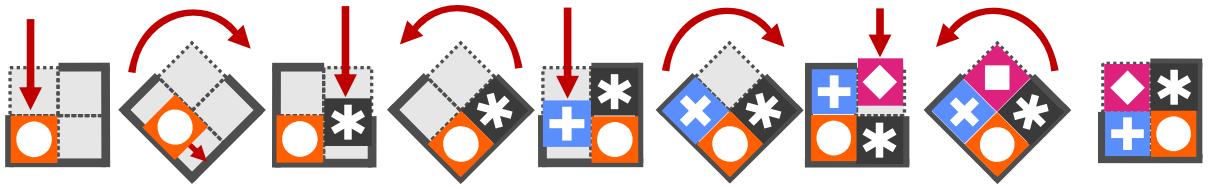
16. Schiebespiel

Das Schiebespiel besteht aus einem Brett mit Wänden und einer Menge farbiger Steine. Das Brett hat in einer Ecke eine Öffnung, durch die man die Steine nach und nach einwerfen kann.

Nach dem Einwerfen eines Steins passiert Folgendes:

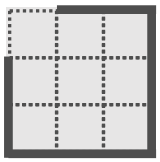
1. Der Stein fällt soweit wie möglich nach unten, bis er entweder auf eine Wand oder einen anderen Stein trifft.
2. Das Brett dreht sich um 90°:
   - nach rechts, wenn die Öffnung oben links ist, und
   - nach links, wenn die Öffnung oben rechts ist
3. Der Stein fällt erneut soweit wie möglich nach unten.

Beispiel mit einem Brett für 4 Steine



Am Ende ist das Brett voll, und die Steine bilden ein Muster.

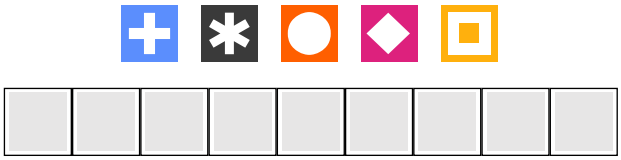
Nun spielen wir mit einem Brett für 9 Steine:



Die Steine sollen dieses Muster bilden:



In welcher Reihenfolge müssen die Steine dazu eingeworfen werden?



Lösung

So ist es richtig:



Im Beispiel mit dem kleinen Brett befindet sich der erste Stein zum Schluss in der Ecke gegenüber der Öffnung, der zweite Stein dann rechts davon und der dritte links, und der letzte Stein liegt in der Ecke bei der Öffnung.

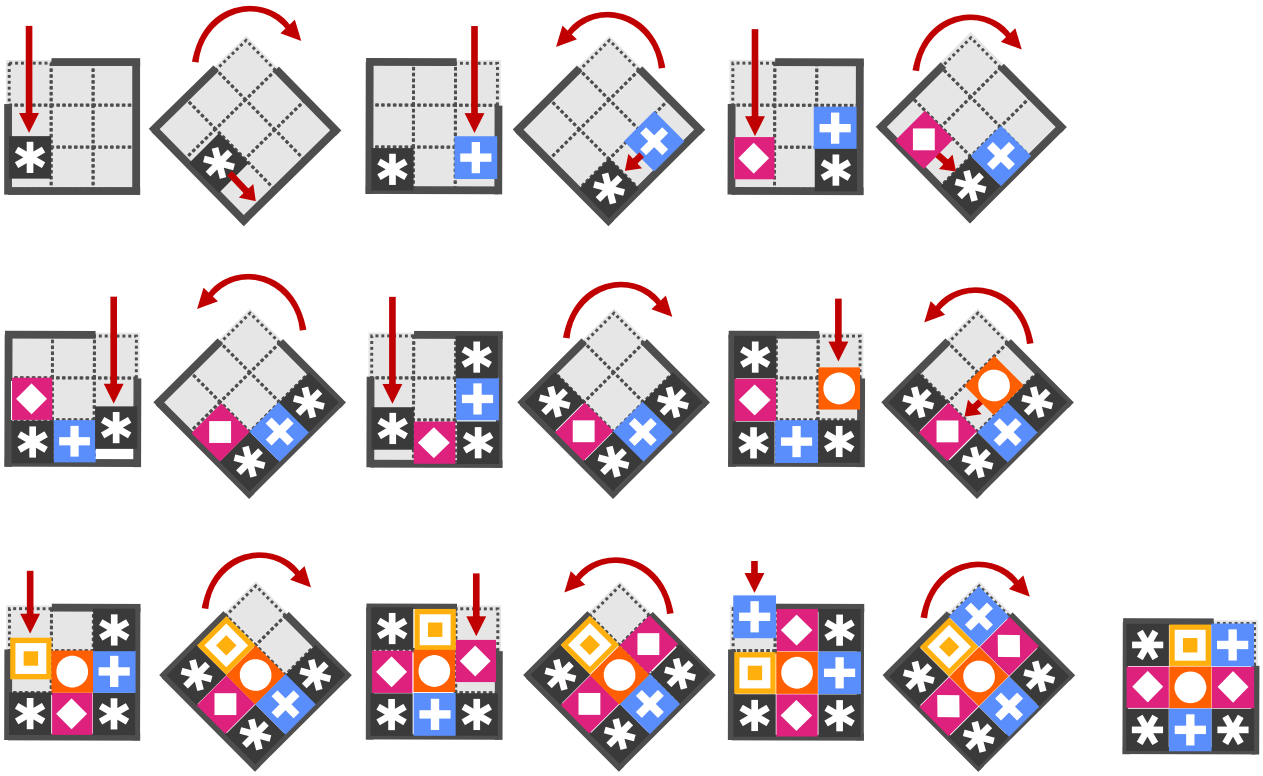
So ähnlich ist es auch beim Brett mit 9 Steinen: Beim Einwerfen fallen die Steine zunächst bis zum Rand bzw. zur nächsten belegten Position. Dadurch werden zunächst die äusseren Felder gegenüber der Öffnung gefüllt, zuallererst das Feld in der Ecke gegenüber der Öffnung.

Im Muster liegt der schwarze Stein in der Ecke gegenüber der Öffnung. Er muss also zuerst eingeworfen werden und fällt nach dem Drehen in seine Ecke. Dann folgen der blaue Stein und der magentafarbene Stein, die von rechts bzw. von links an den Stein in der Ecke heranrutschen. Die Ränder gegenüber der Öffnung werden in ihren äusseren Feldern – zuerst rechts, dann links – durch zwei schwarze Steine komplettiert.

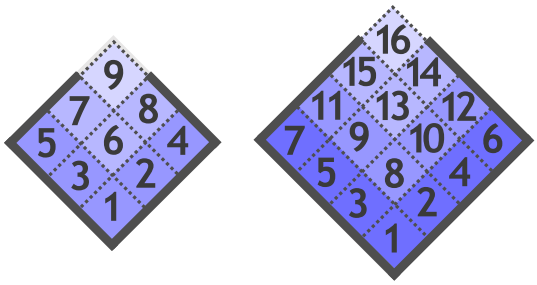
Damit ist aus dem Brett mit 9 freien Feldern ein Brett mit 4 freien Feldern geworden. Es wird analog zum Beispiel in der Aufgabenstellung befüllt. Allerdings liegt die Öffnung nach dem Einwerfen der ersten fünf Steine oben rechts. Das bedeutet: Nachdem zuerst der orange Stein eingeworfen werden muss, der bezüglich der 4 inneren Felder in der Ecke gegenüber der Öffnung liegt, rutschen die beiden nächsten Steine zuerst von links und dann von rechts an diesen Stein heran - im Gegensatz zum Beispiel, wo zweiter und dritter Stein die Felder erst rechts und dann links von der Ecke belegt haben.

 © Informatik-Biber 2026, SVIA

Hier sieht man, wie die Steine genau in der oben genannten Reihenfolge eingeworfen werden und das geforderte Muster bilden:



Es ist aber nicht nötig, das Schiebespiel Schritt für Schritt zu simulieren, um die richtige Reihenfolge der Steine zu bestimmen; sie lässt sich für Spielbretter beliebiger Grösse unmittelbar angeben. Dafür ziehen wir die Regel heran, die wir oben erkannt haben: Das Spielbrett wird, von der Öffnung aus gesehen, von aussen nach innen befüllt, sozusagen in «Schichten». Und in jeder Schicht werden die Felder von der Ecke gegenüber der Öffnung aus befüllt, wobei die Richtungen von Schicht zu Schicht wechseln. (Das liegt daran, dass jede Schicht eine ungerade Anzahl an Feldern hat.) Aus dieser Regel lässt sich die Reihenfolge, in der die Felder des Spielbretts belegt werden, unmittelbar ableiten – hier gezeigt für Bretter mit 9 bzw. 16 Steinen.



Dies ist Informatik!

Die Regeln des Schiebespiels aus dieser Biberaufgabe ergeben direkt einen Algorithmus, mit dem man das Muster ermitteln kann, das eine Eingabefolge von Steinen auf dem Brett bildet: Man simuliert die Ausführung der Regeln, also das Einwerfen der Steine, die Drehungen des Brettes und die dadurch

hervorgerufenen Positionsveränderungen des zuletzt eingeworfenen Steins. In der Erklärung der richtigen Antwort haben wir noch einen zweiten Algorithmus mit demselben Zweck gefunden: Er berechnet für jede Position in der Eingabefolge das Feld, auf das der Stein an dieser Position platziert werden muss. Die Steine einer konkreten Eingabefolge können dann direkt auf die passenden Felder gelegt werden.

Wir haben argumentiert, dass diese beiden Algorithmen die gleiche Wirkung haben. In der Informatik würde man sagen, sie sind *semantisch äquivalent*. Zwei semantisch äquivalente Algorithmen können unterschiedlich beschrieben sein und für dieselbe Eingabe zu ganz unterschiedlichen Abläufen führen. Sie liefern aber für jede mögliche Eingabe jeweils beide die gleiche Ausgabe. Semantik oder Wirkung von Algorithmen spielen eine wichtige Rolle zum Beispiel bei der Optimierung eines Programms (etwa beim Kompilieren des Programms in Maschinencode) oder bei der formalen Verifikation von Programmen. In diesen und einigen anderen Bereichen ist es wichtig, formal zu beweisen, welche Wirkung ein Programm bzw. ein Algorithmus hat – oder dass zwei verschiedene Versionen eines Programms die gleiche Wirkung haben.

Stichwörter und Webseiten

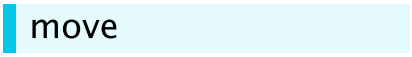
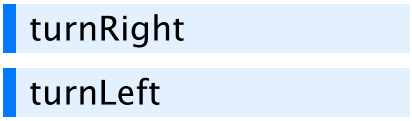
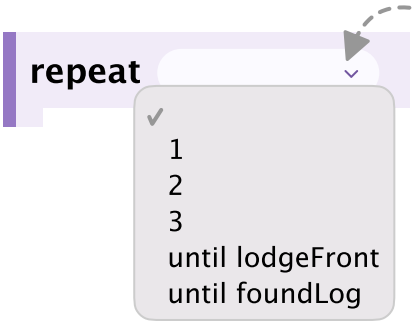
- (Formale) Semantik: `https://de.wikipedia.org/wiki/Semantik_(Logik)#Semantik_in_der_Theoretischen_Informatik`

 © Informatik-Biber 2026, SVIA

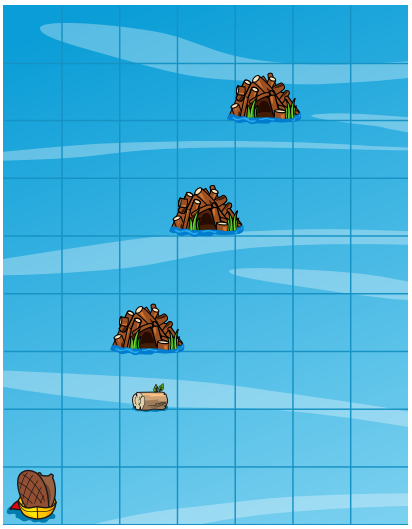
17. Quartierhilfe

Biber Belinda braucht dringend Baumstämme. Jeden Tag legt ein anderer Nachbar einen Baumstamm vor den Bau. Belinda möchte an jedem Tag mit der gleichen Wegbeschreibung den Baumstamm einsammeln. Hat Belinda den Baumstamm eingesammelt, will sie nicht mehr bei den nachfolgenden Nachbarn vorbeifahren. Kannst du Belinda helfen, diese Wegbeschreibung zu finden?

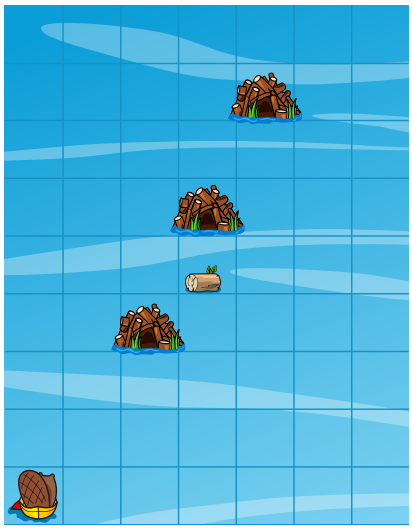
Du kannst folgende Anweisungen verwenden:

Anweisung	**Erklärung**
	Belinda bewegt sich genau ein Feld in Blickrichtung nach vorne. Landet Belinda auf einem Feld mit einem Baumstamm, sammelt sie diesen automatisch ein.
	Belinda dreht sich am Ort nach rechts / links.
	Wenn du Anweisungen in den Block ziehst, werden diese wiederholt (`repeat` = wiederhole).
	Klicke auf das helle Feld und wähle aus, wie oft die Anweisungen wiederholt werden. Bei `until` wird geprüft, ob: • Belinda direkt vor einem Biberbau steht (`lodgeFront`). • Belinda den Baumstamm eingesammelt hat (`foundLog`). Trifft dies zu, endet die Wiederholung nach der letzten Anweisung im `repeat`-Block. Wählst du nichts aus, werden die Anweisungen *dauerhaft* wiederholt.

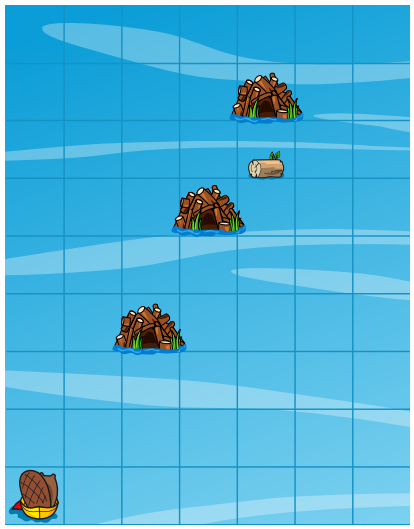
Notiere eine Wegbeschreibung, mit der Belinda an allen drei Tagen den Baumstamm einsammelt. Sobald sie ihn hat, soll sie bei keinem nachfolgenden Nachbarn vorbeifahren.



Tag 1



Tag 2



Tag 3

 © Informatik-Biber 2026, SVIA

Lösung

Eine richtige Lösung lautet wie folgt:

```
repeat until foundLog
  turnRight()
  move()
  move()
  turnLeft()
  move()
  move()
  turnLeft()
  move()
  turnRight()
```

Um an allen Tagen zum Baumstamm zu gelangen, muss man als Erstes erkennen, dass der Weg durch ein wiederkehrendes Muster beschrieben werden kann:

```
  turnRight()
  move()
  move()
  turnLeft()
  move()
  move()
  turnLeft()
  move()
  turnRight()
```

Um am dritten Tag bis zum Nachbarn mit dem Baumstamm zu gelangen, schreibt man die Anweisungen des Musters dreimal hintereinander auf. Damit sammelt sie auch am ersten und zweiten Tag den Baumstamm auf. Die Wegbeschreibung würde somit insgesamt 27 Anweisungen enthalten. Um die Beschreibung zu kürzen, kann man die Anweisungen des Musters auch wiederholt ausführen lassen. Dazu packt man sie in einen `repeat` - Block und wählt `3` Wiederholungen aus:

```
repeat 3
  turnRight()
  move()
  move()
  turnLeft()
  move()
  move()
  turnLeft()
  move()
  turnRight()
```

Damit enthält die Wegbeschreibung statt 27 nur noch zehn Befehle.

Wir haben jetzt eigentlich eine Wegbeschreibung für Belinda gefunden. Allerdings ist die Lösung mit `repeat` und einer festen Anzahl von Wiederholungen nicht ganz zufriedenstellend. Denn wenn das Biberdorf nicht aus drei, sondern z.B. aus fünf Bauten bestehen würde, müssten wir immer die Anzahl der Wiederholungen in der Wegbeschreibung anpassen.

Das Problem lösen wir, indem wir Belindas zweite Einschränkung beachten: Hat Sie einen Baumstamm eingesammelt, möchte sie nicht mehr bei den nachfolgenden Nachbarn vorbeifahren. Genau das wäre aber der Fall, wenn wir `repeat` immer mit einer festen Anzahl an Wiederholungen festlegen. Damit würde Belinda am ersten Tag, obwohl sie den Baumstamm bereits beim ersten Nachbarn eingesammelt hat, immer noch zu Nachbar zwei und drei fahren. An Tag 2 wäre es ähnlich.

Daher nutzen wir die Bedingung `until foundLog` im `repeat` - Block statt einer konkreten Zahl:

```
repeat until foundLog
  ...
```

Denn jetzt werden die Anweisungen des Musters nur so lange wiederholt, bis Belinda den Baumstamm eingesammelt hat. Bevor die Anweisungen im `repeat`- Block erneut ausgeführt werden, wird mithilfe der Bedingung `until foundLog` überprüft, ob bereits ein Baumstamm gefunden wurde. Wenn ja, werden die Anweisungen nicht nochmals ausgeführt und sie fährt nicht mehr zu den nachfolgenden Nachbarn weiter.

Beachte: Obwohl Belinda den Baumstamm bereits nach dem vierten `move` aufgesammelt hat und damit die Bedingung `until foundLog` eigentlich erfüllt ist, werden dennoch die drei nachfolgenden Befehle ausgeführt. Denn die (erneute) Überprüfung mithilfe der Bedingung `until foundLog` erfolgt immer erst dann, wenn alle Befehle im `repeat` - Block vollständig ausgeführt worden sind. Daher stoppt Belinda nicht direkt nach dem Einsammeln des Baumstamms, sondern fährt noch weiter zu ihrer nächsten Startposition, jedoch nicht mehr zum nächsten Nachbar.

```
repeat until foundLog
  turnRight()
  move()
  move()
  turnLeft()
  move()
  move() <-- hier sammelt Belinda den Baumstamm auf
  turnLeft()
  move()
  turnRight() <- hier erreicht sie ihre neue Startposition
```

Vertiefung: Wir haben den Weg über ein Muster beschrieben. Das Muster wird im `repeat` - Block in zwei Abschnitte unterteilt:

Abschnitt 1: Weg bis zum Baumstamm

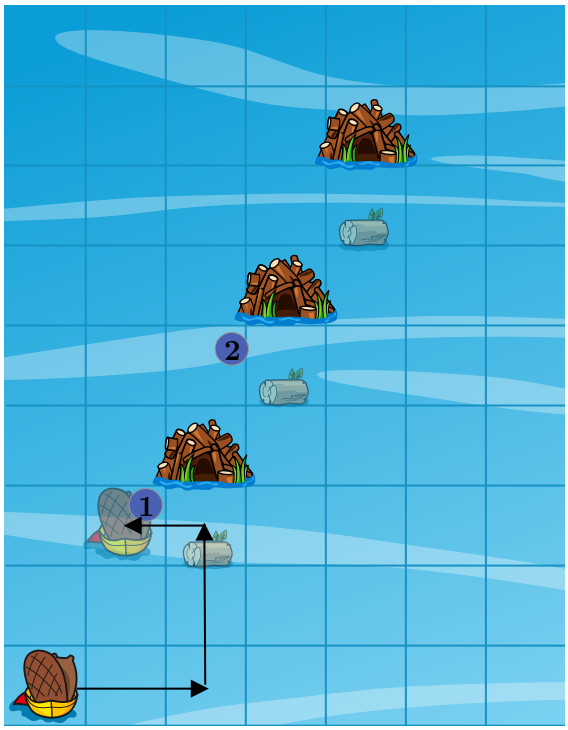
```
  turnRight()
  move()
```

```
move()
turnLeft()
move()
move()
```

Abschnitt 2: Weg weiter zur neuen Startposition

```
turnLeft()
move()
turnRight()
```

Unterteilst du dein Muster in diese beiden Abschnitte, kann Belinda einen beliebigen Weg zum Baumstamm **und** einen beliebigen Weg zur neuen Startposition (Position 1 für den zweiten Baumstamm, Position 2 für den dritten Baumstamm) fahren. Dennoch sammelt Belinda immer alle Baumstämme auf und fährt nie weiter zum nächsten Nachbarn.



Dies ist Informatik!

Eine *Invariante* (hier: *Schleifeninvariante*) ist eine Eigenschaft, die vor und nach jedem Durchlauf einer Schleife gültig ist. Sie hilft dabei, den Zustand eines Systems zu beschreiben und die Korrektheit eines Verfahrens zu überprüfen. Besonders häufig werden Invarianten bei Schleifen verwendet, um zu zeigen, dass bestimmte Bedingungen bei jedem Schleifendurchlauf erhalten bleiben. Bezogen auf unser Beispiel müssen wir unseren Biber so ausrichten, dass Startposition (vor dem Schleifendurchlauf) und Endposition (nach dem Schleifendurchlauf) identisch sind. Dabei definiert sich die jeweilige Start-/Endposition durch die Blickrichtung und durch den Standort des Bibers in Relation zum nächsten Biberbau.

Stichwörter und Webseiten

- Schleifeninvariante: `https://de.wikipedia.org/wiki/Schleifeninvariante`
- Wiederholung: `https://de.wikipedia.org/wiki/Schleife_(Programmierung)`

 © Informatik-Biber 2026, SVIA

A. Aufgabenautoren

Esraa Almajhad

Sakuya Araki

Masiar Babazadeh

Wilfried Baumann

Hans-Joachim Böckenhauer

Graeme Buckie

Leonardo Cavalcante

Cesar Cepeda

Maria Cepeda

Špela Cerar

Kris Coolsaet

Christian Datzko

Heather Duan

Oscar Durango

Nora A. Escherle

Susanne Falkenberg-Thut

Gerald Futschek

Daniel Aagrén Geert

Arnheiður Guðmundsdóttir

Jens Hartmann

Heikki Hyyro

Heidi Kaarto

Filiz Kalelioğlu

Asterios Karagiannis

Atheer Khabti

David Khachatryan

Doyong Kim

Vaidotas Kinčius

Lukas Lehner

Gunwoong Lim

Vaida Masiulionytė-Dagienė

Yoshiaki Matsuzawa

Thomas Naughton

Justina Oostendorp

Belén Palop

Praphan Pavarangkoon

Jean-Philippe Pellet

Emmanuel Plan

Zsuzsa Pluhár

Wolfgang Pohl

Liane Raji

Lorenzo Repetto

Pedro Ribeiro

Kirsten Schlüter

Margareta Schlüter

Dirk Schmerenbeck

Eljakim Schrijvers

Darija Dasovic Sebalj

Giovanni Serafini

Itsuki Shimada

Bronius Skūpas

Emma Sotomayor

Jacqueline Staub

Alieke Stijf

Monika Tomcsanyiova

Jiří Vaníček

Marcel Vlastuin

Michael Weigend

Chris Wetherell

Kyra Willekes

Alexander Wiss

© Informatik-Biber 2026, SVIA

B. Akademische Partner

Haute école pédagogique du canton de Vaud
https://www.hepl.ch/

Ausbildungs- und Beratungszentrum für Informatikunterricht der ETH Zürich
https://www.abz.inf.ethz.ch/

La Scuola universitaria professionale della Svizzera italiana (SUPSI)
https://www.supsi.ch/

Pädagogische Hochschule Luzern
https://www.phlu.ch/

Universität Trier
https://www.uni-trier.de/

C. Sponsoring



Hasler Stiftung
https://www.haslerstiftung.ch/



Amt für Kindergarten, Volksschule und Beratung, Bildungs- und Kulturdirektion, Kanton Bern
https://www.bkd.be.ch/de/start/ueber-uns/die-organisation/amt-fuer-kindergarten-volksschule-und-beratung.html



Amt für Wirtschaft, Kanton Zürich
https://www.zh.ch/de/volkswirtschaftsdirektion/amt-fuer-wirtschaft.html



Informatik Stiftung Schweiz
https://informatics-foundation.ch



cyon
https://www.cyon.ch



Senarclens Leu & Partner
https://senarclens.com/



Wealth Management IT and UBS Switzerland IT
https://www.ubs.com/



Basellandschaftliche Gebäudeversicherung
https://bgv.ch

© Informatik-Biber 2026, SVIA