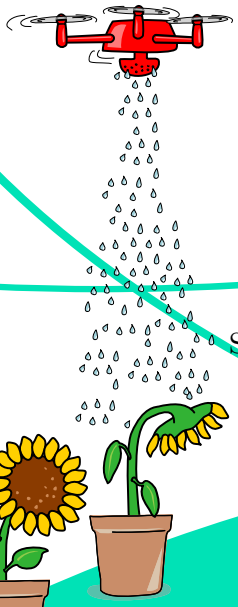


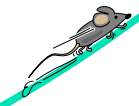


INFORMATIK-BIBER SCHWEIZ
CASTOR INFORMATIQUE SUISSE
CASTORO INFORMATICO SVIZZERA

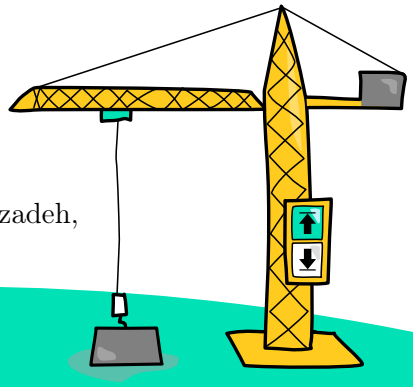
Quesiti e soluzioni 2026

9º e 10º anno scolastico



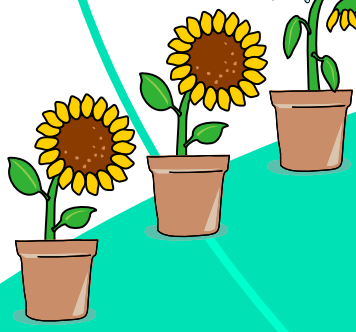


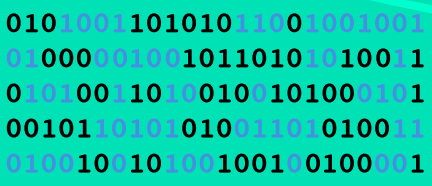
https://www.castoro-informatico.ch/



A cura di:

Susanne Falkenberg-Thut, Nora A. Escherle, Masiar Babazadeh,
Christian Giang, Jean-Philippe Pellet

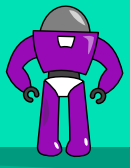




SSII

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

Hanno collaborato al Castoro Informatico 2026

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

Capi progetto: Nora A. Escherle, Janine Blattner

Un particolare ringraziamento per il lavoro sui quesiti del concorso Svizzero va a:
Juraj Hromkovič, Regula Lacher: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht
Christian Datzko, Gymnasium Liestal
Nadine Fröhlich, Università 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, Germania

La scelta dei quesiti è stata svolta in collaborazione con gli organizzatori dei concorsi in Germania, Austria e Ungheria. Ringraziamo specialmente:
Wolfgang Pohl, Karsten Schulz, Margaretha Schlüter, Kirsten Schlüter, Michael Weigend: Bundesweite Informatikwettbewerbe (BWINF), Germania
Wilfried Baumann, Liam Baumann: Österreichische Computer Gesellschaft, Austria
Gerald Futschek, Lukas Lehner: Technische Universität Wien, Austria
Zsuzsa Pluhár, Bence Gaal: ELTE Informatikai Kar, Ungheria

La versione online del concorso è stata creata su cuttle.org. Ringraziamo per la buona collaborazione:
Eljakim Schrijvers, Justina Oostendorp, Alieke Stijf, Kyra Willekes: cuttle.org, Olanda
Andrew Csizmadia: Raspberry Pi Foundation, Regno Unito

I compiti di programmazione sono stati creati e sviluppati appositamente per la piattaforma online. Desideriamo ringraziare le seguenti persone:
Jacqueline Staub: Universität Trier, Germania
Dirk Schmerenbeck: Universität Trier, Germania
Dave Oostendorp: cuttle.org, Olanda

Per il supporto durante le settimane del concorso ringraziamo:
Valérie Verdan: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht

Ringraziamo l'ETH per l'organizzazione e lo svolgimento della finale del Castoro:
Dennis Komm, Hans-Joachim Böckenhauer, Angélica Herrera Loyo, Andre Macejko, Adeline Pittet, Valérie Verdan: ETH Zürich, Ausbildungs- und Beratungszentrum für Informatikunterricht
Per la correzione dei compiti finali:
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 Whittington, Ursus Wigger, Guiheng Zheng, Mateo Zuluaga

© Castoro Informatico 2026, SSII

Per la traduzione dei compiti finali in francese:
Jean-Philippe Pellet: Haute école pédagogique du canton de Vaud
Responsabili dei workshop per i finalisti:
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

Un ringraziamento speciale va ai nostri grandi sponsor Juraj Hromkovič, Dennis Komm, Gabriel Parriaux e la Fondazione Hasler. Senza di loro, questo concorso non esisterebbe.

L'edizione dei quesiti in lingua tedesca è stata utilizzata anche in Germania, Austria e Lussemburgo. Anche la versione in lingua francese era in uso in Lussemburgo.

La traduzione francese è stata curata da Elsa Pellet, mentre quella italiana da Christian Giang.

© Castoro Informatico 2026, SSII

INFORMATIK-BIBER SCHWEIZ
CASTOR INFORMATIQUE SUISSE
CASTORO INFORMATICO SVIZZERA

Il Castoro Informatico 2026 è stato organizzato dalla Società Svizzera per l'Informatica nell'Insegnamento (SSII) e sostenuto in modo significativo e generoso dalla Fondazione Hasler. Altri partner e sponsor che hanno sostenuto finanziariamente il concorso sono l'Ufficio per la scuola materna, elementare e la consulenza (AKVB) del Cantone di Berna, l'Ufficio per l'economia AWI del Cantone di Zurigo, l'Assicurazione immobili del Cantone di Basilea Campagna, CYON AG, la Fondazione Informatica Svizzera e UBS.

I seguenti partner accademici ci sostengono nella creazione degli esercizi: il Politecnico federale di Zurigo (ETH Zürich), l'Ausbildungs- und Beratungszentrum für Informatikunterricht, l'Alta scuola pedagogica del Canton Vaud, la Scuola universitaria professionale della Svizzera italiana, l'Alta scuola pedagogica di Lucerna e l'Università di Treviri.

Questo quaderno è stato creato il 23 novembre 2026 con il sistema per la preparazione di testi LaTeX. Ringraziamo Christian Datzko per lo sviluppo del sistema di generazione dei testi che ha permesso di generare le 36 versioni di questa brochure (divise per lingua e livello scolastico). Il sistema è stato riprogrammato basandosi sul sistema precedente, sviluppato nel 2014 assieme a Ivo Blöchliger. Ringraziamo Jean-Philippe Pellet per lo sviluppo del sistema `bebras`, utilizzato dal 2020 per la conversione dei documenti sorgente dai formati Markdown e YAML.

Nota: Tutti i link sono stati verificati il 20.11.2026.

I quesiti sono distribuiti con Licenza Creative Commons Attribuzione – Non commerciale – Condividi allo stesso modo 4.0 Internazionale. Gli autori sono elencati a pagina 75.

Premessa

Il concorso del «Castoro Informatico», presente già da diversi anni in molti paesi europei, ha l'obiettivo di destare l'interesse per l'informatica nei bambini e nei ragazzi. In Svizzera il concorso è organizzato in tedesco, francese e italiano dalla Società Svizzera per l'Informatica nell'Insegnamento (SSII), con il sostegno della fondazione Hasler.

Il Castoro Informatico è il partner svizzero del Concorso «Bebras International Contest on Informatics and Computer Fluency» (`https://www.bebras.org/`), situato in Lituania.

Il concorso si è tenuto per la prima volta in Svizzera nel 2010. Nel 2012 l'offerta è stata ampliata con la categoria del «Piccolo Castoro» (3º e 4º anno scolastico).

Il Castoro Informatico incoraggia gli alunni ad approfondire la conoscenza dell'informatica: esso vuole destare interesse per la materia e contribuire a eliminare le paure che sorgono nei suoi confronti. Il concorso non richiede alcuna conoscenza informatica pregressa, se non la capacità di «navigare» in internet poiché viene svolto online. Per rispondere alle domande sono necessari sia un pensiero logico e strutturato che la fantasia. I quesiti sono pensati in modo da incoraggiare l'utilizzo dell'informatica anche al di fuori del concorso.

Nel 2026 il Castoro Informatico della Svizzera è stato proposto a cinque differenti categorie d'età, suddivise in base all'anno scolastico:

- 3º e 4º anno scolastico
- 5º e 6º anno scolastico
- 7º e 8º anno scolastico
- 9º e 10º anno scolastico
- 11º al 13º anno scolastico

Ogni categoria aveva quesiti classificati in tre livelli di difficoltà: facile, medio e difficile. Alla categoria del 3º e 4º anno scolastico sono stati assegnati 9 quesiti da risolvere, di cui 3 facili, 3 medi e 3 difficili. Alla categoria del 5º e 6º anno scolastico sono stati assegnati 12 quesiti, suddivisi in 4 facili, 4 medi e 4 difficili. Ogni altra categoria ha ricevuto invece 15 quesiti da risolvere, di cui 5 facili, 5 medi e 5 difficili.

Per ogni risposta corretta sono stati assegnati dei punti, mentre per ogni risposta sbagliata sono stati detratti. In caso di mancata risposta il punteggio è rimasto inalterato. Il numero di punti assegnati o detratti dipende dal grado di difficoltà del quesito:

	Facile	**Medio**	**Difficile**
Risposta corretta	6 punti	9 punti	12 punti
Risposta sbagliata	−2 punti	−3 punti	−4 punti

Il sistema internazionale utilizzato per l'assegnazione dei punti limita l'eventualità che il partecipante possa ottenere buoni risultati scegliendo le risposte in modo casuale.

© Castoro Informatico 2026, SSII

Ogni partecipante inizia con un punteggio pari a 45 punti (risp., 3º e 4º anno scolastico: 27 punti, 5º e 6º anno scolastico: 36 punti).

Il punteggio massimo totalizzabile era dunque pari a 180 punti (risp., 3º e 4º anno scolastico: 108 punti, 5º e 6º anno scolastico: 144 punti), mentre quello minimo era di 0 punti.

In molti quesiti le risposte possibili sono state distribuite sullo schermo con una sequenza casuale. Lo stesso quesito è stato proposto in più categorie d'età. Questi quesiti presentavano livelli di difficoltà diversi nei vari gruppi di età.

Alcuni quesiti sono indicati come «bonus» per determinate categorie di età: non contano nel totale dei punti, ma vengono utilizzati come spareggio per punteggi identici in caso di qualificazione agli eventuali turni successivi.

Per ulteriori informazioni:

Società Svizzera per l'Informatica nell'Insegnamento
SVIA-SSIE-SSII
Castoro Informatico
Masiar Babazadeh

`https://www.castoro-informatico.ch/kontaktieren/`
`https://www.castoro-informatico.ch/`

© Castoro Informatico 2026, SSII

Indice

Hanno collaborato al Castoro Informatico 2026 . . . . . **i**

Premessa . . . . . **iv**

Indice . . . . . **vi**

1. Testa o croce 2026-IN-05 **1**

2. Colora il quadrato 2026-DE-09 **3**

3. La fabbrica di auto 2026-JP-03 **7**

4. Il giardino di Gerald 2026-LT-01a **11**

5. Consigli dai contatti 2026-HU-01b **15**

6. Il metodo di Muddy 2026-DE-10 **19**

7. Il lucchetto di Benji 2026-JP-05 **23**

8. Gira che ti rigira 2026-AU-04 **27**

9. Il ramyeon perfetto 2026-KR-09 **31**

10. La diga dei castori 2026-CH-08 **35**

11. Fiori dimenticati 2026-JP-02b **41**

12. Biglie musicali 2026-DE-05 **45**

13. Il gioco delle monete 2026-AT-02 **49**

14. Il salto del cavallo 2026-NL-01 **53**

15. Tacos gratis! 2026-MX-02b **57**

16. Nascondino 2026-US-03b **61**

17. Tinto Azul 2026-PT-01a **65**

18. Un aiuto dai vicini 2026-DE-90 **69**

A. Autori dei quesiti **75**

B. Partner accademici **77**

C. Sponsoring **78**

© Castoro Informatico 2026, SSII

1. Testa o croce

Ogni moneta dei castori ha due facce: «testa»  e «croce» .

Alex ha davanti a sé 7 monete:



Ora decide di girare esattamente due monete per un certo numero di volte.

In quale disposizione NON si troveranno MAI le monete di Alex?

A) 

B) 

C) 

D) 

Soluzione

La risposta corretta è B.

Ogni volta che Alex capovolge esattamente due monete, su di esse compaiono facce diverse rispetto a prima. Si possono verificare tre casi:

1. Entrambe le monete ora mostrano croce, mentre prima mostravano testa,
2. entrambe ora mostrano testa, mentre prima mostravano croce, oppure
3. una ora mostra testa invece di croce e l'altra ora mostra croce invece di testa.

Il *numero di teste* (ovvero la quantità di monete che mostrano testa) varia di 2 (nei casi 1 e 2) oppure resta invariato, cambiando di 0 (nel caso 3). Di conseguenza, se il numero iniziale di teste è pari, rimarrà sempre pari; se invece è dispari, rimarrà sempre dispari.

All'inizio le monete di Alex mostrano 2 teste, un numero dunque pari. Di conseguenza, capovolgendole a due a due, Alex potrà ottenere esclusivamente configurazioni in cui il numero di teste è pari, come nelle risposte A (4 teste), C (4 teste) e D (2 teste). Le monete non potranno mai trovarsi in una disposizione con un numero dispari di teste. Soltanto nella risposta B il numero di teste è dispari, ovvero pari a 3.

Questa è l'informatica!

I numeri interi possono essere pari o dispari; questa proprietà è nota come parità. Come visto nella spiegazione precedente, la parità del numero di teste nella fila di Alex non può subire variazioni finché si girano sempre e solo due monete alla volta.

In informatica, una proprietà che rimane immutata in seguito all'applicazione di determinate operazioni prende il nome di invariante. Individuare l'invariante di questo problema (la parità del numero di teste) permette di individuare la soluzione corretta con estrema rapidità, senza dover procedere per tentativi.

Le invarianti trovano frequente impiego nell'ambito informatico, ad esempio per:

- per verificare se gli algoritmi funzionano correttamente;
- per dimostrare che certi stati non possono essere raggiunti; oppure
- per analizzare sistemi con molti stati possibili.

Parole chiave e siti web

- Invariante: `https://it.wikipedia.org/wiki/Invariante_(informatica)`
- Parità: `https://it.wikipedia.org/wiki/Bit_di_parità`

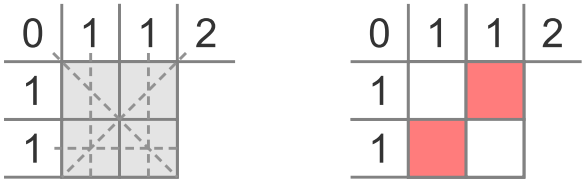
 © Castoro Informatico 2026, SSII

2. Colora il quadrato

In una griglia quadrata, le caselle devono essere colorate di rosso o di bianco. All'inizio tutte le caselle sono grigie.

I numeri sui bordi indicano quante caselle devono essere **rosse** in ogni riga, in ogni colonna e in ognuna delle due diagonali. Tutte le altre caselle devono diventare bianche.

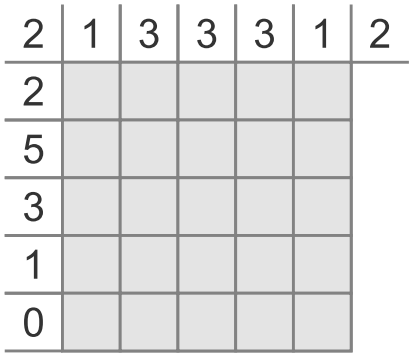
Ecco un piccolo esempio con un quadrato 2×2:



In ogni riga e in ogni colonna deve esserci esattamente una casella rossa. Sulla diagonale che va dall'angolo in alto a sinistra a quello in basso a destra non sono permesse caselle rosse, sull'altra diagonale devono esserci esattamente 2 caselle rosse.

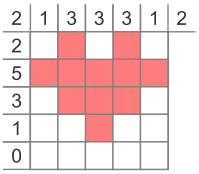
Questa colorazione è l'unica soluzione per questo quadrato.

Qual è la soluzione per questo quadrato 5×5?

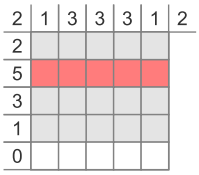


Soluzione

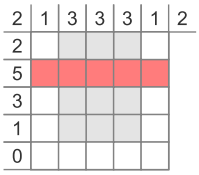
La risposta corretta è la seguente:



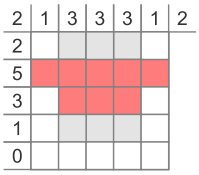
Non c'è nessun altro modo di colorare le caselle rispettando le indicazioni. Lo mostra il procedimento descritto qui sotto per trovare questa soluzione:



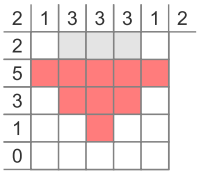
Passo 1: La riga con l'indicazione **5** deve essere colorata tutta di rosso, quella con l'indicazione **0** tutta di bianco.



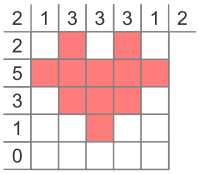
Passo 2: Le caselle ancora grigie nella prima e nella quinta colonna devono essere bianche: in queste colonne c'è infatti già una casella rossa.



Passo 3: Le altre caselle grigie della riga centrale devono essere colorate di rosso: solo così questa riga può avere 3 caselle rosse.



Passo 4: Nella quarta riga, le caselle 2 e 4 devono essere bianche a causa delle indicazioni per le diagonali; quindi la casella 3 deve essere rossa.



Passo 5: Infine, nella prima riga le caselle 2, 3 e 4 devono necessariamente essere colorate in sequenza **rosso-bianco-rosso**.

Senza le indicazioni per le diagonali ci sarebbero più soluzioni.

Questa è l'informatica!

Ogni possibile colorazione in rosso e bianco del quadrato 5×5 rappresenta una potenziale soluzione del rompicapo. Tutte insieme costituiscono il cosiddetto *spazio delle soluzioni*. Esiste un algoritmo molto semplice per trovarla: testare ciascuna candidata verificando se rispetta tutte le condizioni. Tuttavia, per un quadrato 5×5 le combinazioni possibili sono 2^{25} (pari a 33'554'432), e occorrerebbe troppo tempo per generarle e verificarle tutte. Per non parlare di un quadrato 6×6, con oltre 68 miliardi di candidate, o di un quadrato 7×7, che ne conta oltre 562'000 miliardi!

 © Castoro Informatico 2026, SSII

Per questo motivo in informatica si impiegano metodi di ricerca più efficienti. Quando una soluzione deve soddisfare determinati vincoli, questi ultimi vengono sfruttati per costruirla passo dopo passo. Spesso si ricorre al backtracking: se una scelta effettuata in corso d'opera (ad esempio l'assegnazione di un colore) conduce a un vicolo cieco in cui non è più possibile trovare una soluzione, si torna sui propri passi per provare un'altra opzione.

Il vantaggio di questo approccio è che si applica a qualsiasi problema che richieda di esplorare uno spazio delle soluzioni, indipendentemente dalle dimensioni o dal numero di soluzioni. L'efficacia è massima quando, per la maggior parte delle decisioni, i vincoli non lasciano margini di scelta (ad esempio imponendo un unico colore possibile). Nel procedimento illustrato sopra ciò si è verificato in ogni passaggio, permettendo di giungere alla soluzione senza dover mai tornare indietro.

Parole chiave e siti web

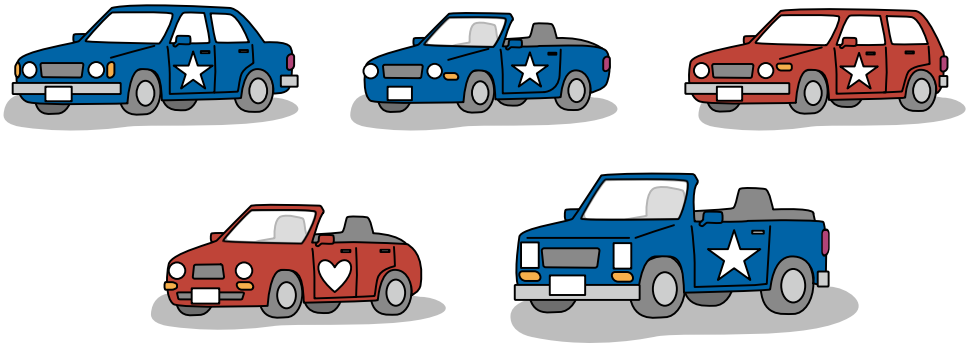
- Algoritmi: `https://it.wikipedia.org/wiki/Algoritmo`
- Ricerca nello spazio delle soluzioni: `https://en.wikipedia.org/wiki/Feasible_region`
- Backtracking: `https://it.wikipedia.org/wiki/Backtracking`



© Castoro Informatico 2026, SSII

3. La fabbrica di auto

In una fabbrica vengono costruiti questi cinque modelli di auto:



Un robot riconosce queste caratteristiche nelle auto appena costruite:

- Colore dell'auto: blu o rosso
- Logo: stella o cuore
- Porte: due (solo davanti) o quattro (davanti e dietro)
- Tetto: aperto o chiuso
- Fari: rotondi o squadrati

Per ogni nuova auto, il robot deve stabilire con certezza a quale modello appartiene.
Binia, esperta di informatica, dice: per farlo, il robot deve osservare solo tre caratteristiche.

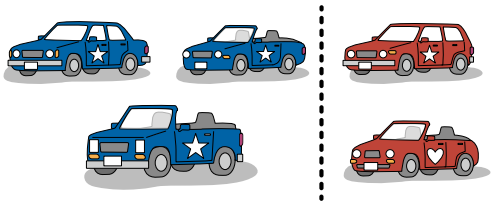
Quali sono queste tre caratteristiche? Cerchiale.

Colore dell'auto, Logo, Porte, Tetto, Fari

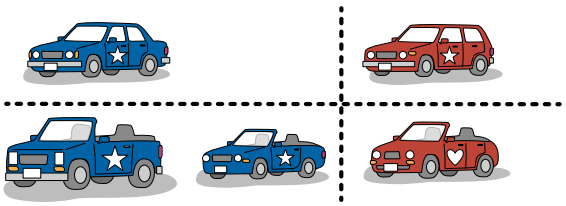
Soluzione

Il robot può stabilire con certezza il modello di ogni nuova auto se osserva il colore dell'auto, il tetto e i fari.

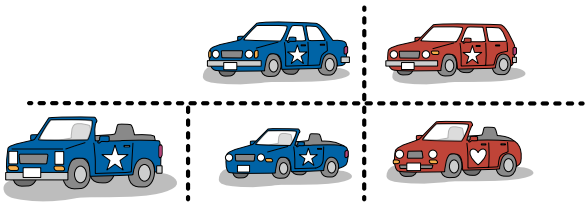
Con ogni caratteristica il robot può dividere i modelli in due gruppi. Con la giusta combinazione di tre caratteristiche può dividere i modelli in gruppi che contengono un solo modello ciascuno; così può assegnare una nuova auto a quel modello. Per trovare le tre caratteristiche giuste si può procedere così:



Si comincia con una caratteristica, per esempio il colore dell'auto, e si dividono i modelli di auto in due gruppi: auto blu e auto rosse.



Poi si cerca una caratteristica che permetta di porre un'ulteriore suddivisione. Possiamo procedere per esempio con il tetto, perché con il colore dell'auto e il tetto si ottengono quattro gruppi: due modelli blu con il tetto aperto, un modello blu con il tetto chiuso, un modello rosso con il tetto aperto, un modello rosso con il tetto chiuso.



Ora bisogna dividere solo il gruppo con due modelli. Per farlo è adatta la caratteristica dei fari, poiché uno dei due modelli blu con il tetto aperto ha i fari rotondi e l'altro squadrati.

C'è una sola risposta corretta. La tabella seguente mostra che solo la combinazione di caratteristiche colore dell'auto, tetto e fari è diversa per ogni modello. Con questa combinazione di caratteristiche si possono distinguere tutti i modelli di auto tra loro - e assegnare con certezza ogni nuova auto a un modello.

 © Castoro Informatico 2026, SSII

Numero	Modello	Colore	Logo	Porte	Tetto	Fari
1		blu	stella	davanti e dietro	chiuso	rotondi
2		blu	stella	solo davanti	aperto	rotondi
3		rosso	stella	davanti e dietro	chiuso	rotondi
4		rosso	cuore	davanti e dietro	aperto	rotondi
5		blu	stella	solo davanti	aperto	squadrati

Con ogni altra combinazione di caratteristiche, almeno due modelli di auto non si possono distinguere tra loro:

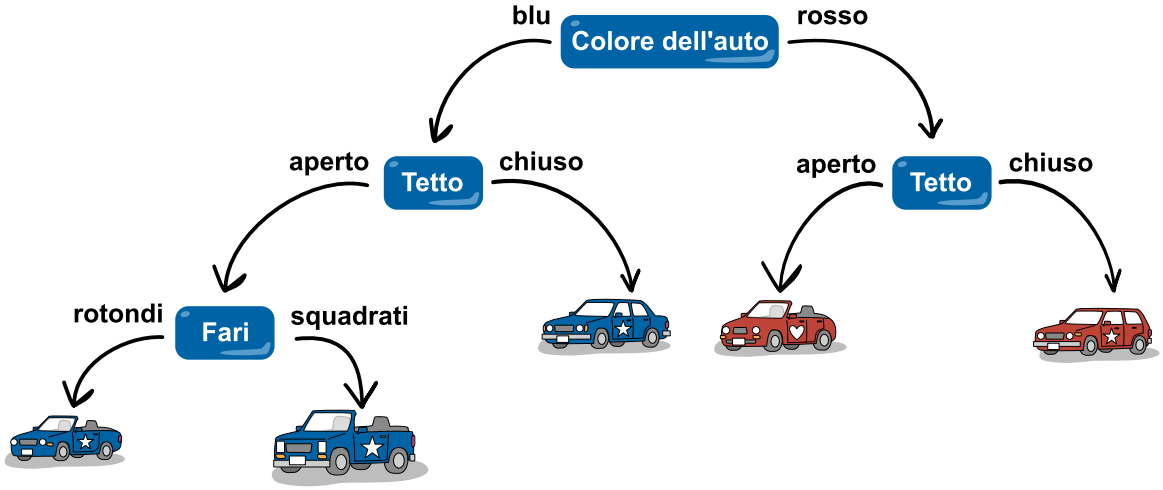
Combinazione di caratteristiche	Modelli di auto con le stesse caratteristiche
Colore, logo, porte	2 e 5
Colore, logo, tetto	2 e 5
Colore, logo, fari	1 e 2
Colore, porte, tetto	2 e 5
Colore, porte, fari	3 e 4
Logo, porte, tetto	1 e 3, e anche 2 e 5
Logo, porte, fari	1 e 3
Logo, tetto, fari	1 e 3
Porte, tetto, fari	1 e 3

Questa è l'informatica!

In questo compito un robot deve assegnare con certezza degli oggetti (le auto) a diversi gruppi (cioè i modelli di auto) in base a determinate caratteristiche (come il colore). Per un'assegnazione di questo tipo – gli esperti parlano anche di *classificazione* – il programma del robot può usare un *albero di decisione*. Un albero di decisione stabilisce quali caratteristiche il robot deve controllare, in quale ordine, e a quali assegnazioni portano i risultati dei controlli.

Scrivere un programma che fa assegnazioni in base a un albero di decisione non è troppo difficile. È più difficile costruire l'albero di decisione, cioè scegliere con astuzia le caratteristiche su cui decidere e il loro ordine. In teoria si potrebbero testare tutti i possibili alberi di decisione per individuare quello corretto, tuttavia, poiché il numero di alberi possibili è solitamente elevatissimo, questo approccio risulta estremamente dispendioso (se non impraticabile). Al contrario, è possibile procedere in modo euristico, sfruttando le osservazioni pratiche sugli oggetti da classificare. Per esempio, si sceglie «colore» come prima caratteristica, perché in base al colore si possono formare due gruppi di auto, o di modelli di auto, più o meno della stessa grandezza. Come seconda caratteristica non si sceglie il «logo», perché tutte le auto blu hanno la stella come logo, quindi controllare questa caratteristica non aiuta a fare l'assegnazione.

Ecco l'albero di decisione per il robot di questo compito:



Generare un albero di decisione a partire da esempi è un procedimento che si può persino automatizzare. Già negli anni Ottanta, nell'ambito della ricerca sui *sistemi basati sulla conoscenza* e sull'*apprendimento automatico*, gli informatici hanno studiato come ricavare alberi di decisione da casi di esempio e dalla frequenza delle caratteristiche osservate.

Parole chiave e siti web

- Albero di decisione: `https://it.wikipedia.org/wiki/Albero_di_decisione`
- Sistemi basati sulla conoscenza: `https://it.wikipedia.org/wiki/Sistema_basato_sulla_conoscenza`
- Apprendimento automatico: `https://it.wikipedia.org/wiki/Apprendimento_automatico`

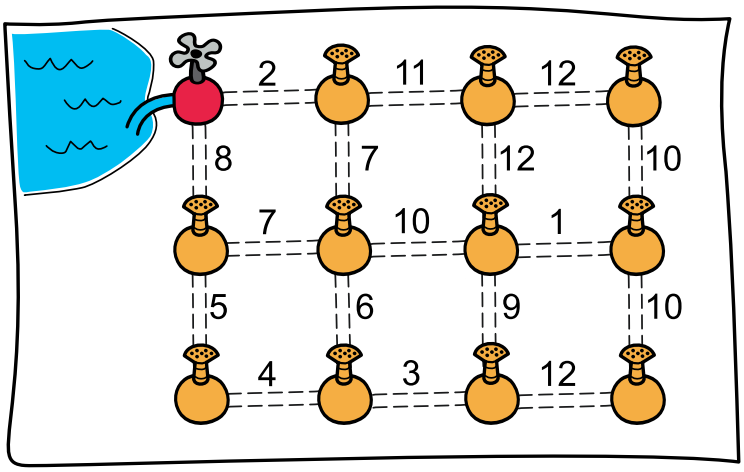
 © Castoro Informatico 2026, SSII

4. Il giardino di Gerald

Il giardiniere Gerald ha installato un nuovo impianto di irrigazione nel suo giardino fiorito.

Per farlo, Gerald ha utilizzato lo schema sottostante, in cui sono illustrati:

- la pompa 
- tutti gli irrigatori  e
- tutti i tratti ⚌⚌⚌⚌⚌ in cui è possibile posare i tubi.



Nello schizzo è indicato anche quanti metri di tubo servono in ogni tratto.

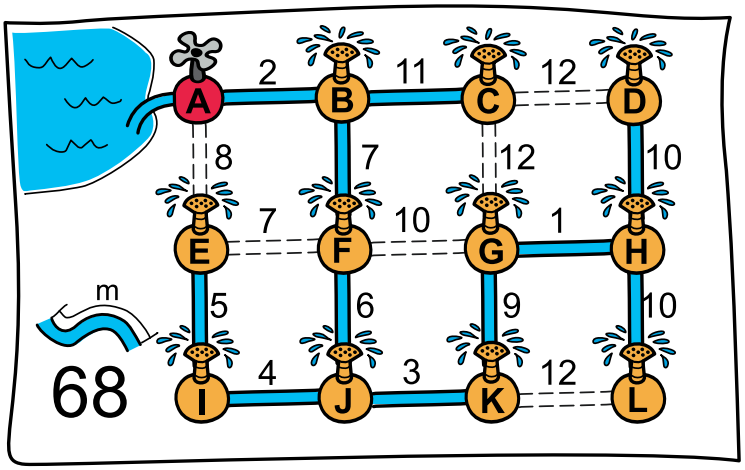
Gerald non ha posato i tubi in ogni tratto disponibile, ma solo il quantitativo necessario affinché

- tutti gli irrigatori siano raggiunti dall'acqua della pompa **e**
- impiegando il minor numero possibile di metri di tubo in totale.

In quali tratti Gerald ha posato i tubi?

Soluzione

La risposta corretta è la seguente:



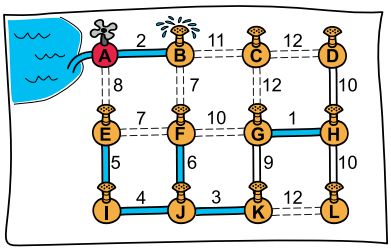
Per semplificare la spiegazione, abbiamo etichettato gli irrigatori con delle lettere e distinto i tratti in base alla lunghezza del tubo richiesto. Parleremo inoltre di *ciclo* quando, partendo da un irrigatore (o dalla pompa) e seguendo i tubi installati, è possibile tornare al punto di partenza senza percorrere alcuno tratto più di una volta.

Per trovare la risposta corretta si può usare la seguente strategia:

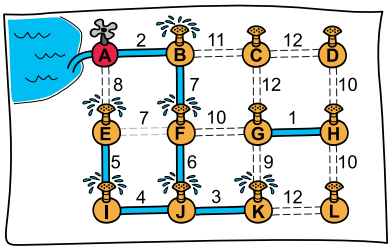
- Esamina i tratti in ordine di lunghezza, partendo dal più corto.
- Posa un tubo in un tratto se e solo se così *non* si forma un ciclo.

Perché è opportuno evitare i cicli? Un ciclo di tubi richiede più materiale del necessario: rimuovendo un tubo da un ciclo, infatti, tutti gli irrigatori continuano comunque a ricevere l'acqua. Per esempio, se nel giardino di Gerald abbiamo già posato i tubi di lunghezza 4, 5 e 6 e ora posiamo anche il tubo di 7 metri tra E e F, si forma il ciclo E, F, J, I, E, senza che nessun irrigatore in più riceva l'acqua. Il tubo tra E e F si può quindi risparmiare. Siccome la strategia comincia dai tratti (cioè dai tubi) più corti, in ogni possibile ciclo si risparmia il tubo più lungo. Così, in totale, servono meno metri di tubo possibile.

La strategia descritta sopra porta sempre a una posa dei tubi che usa meno metri di tubo possibile e collega tutti gli irrigatori alla pompa. Per il giardino di Gerald la strategia funziona così:

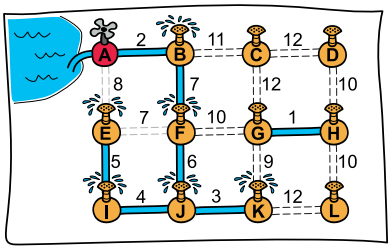


In tutti i tratti di lunghezza da 1 a 6 viene posato un tubo, perché così non si forma nessun ciclo.

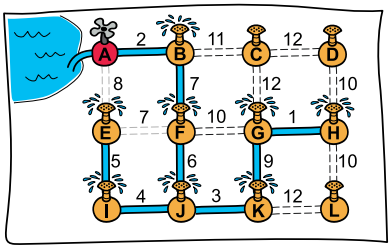


Nei tratti di lunghezza 7

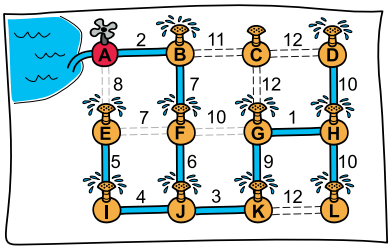
- tra B e F viene posato un tubo, ma
- tra E e F non viene posato nessun tubo, perché si formerebbe il ciclo E, F, J, I, E.



Tra A ed E (lunghezza 8) non viene posato nessun tubo, perché si formerebbe il ciclo A, B, F, J, I, E, A.

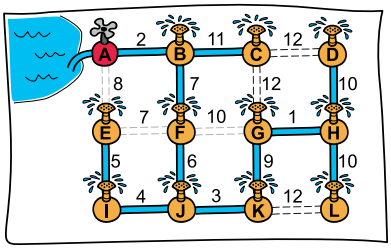


Tra G e K (lunghezza 9) viene posato un tubo.



Nei tratti di lunghezza 10

- viene posato un tubo tra D e H e uno tra H e L, ma
- tra F e G non viene posato nessun tubo, perché si formerebbe il ciclo F, G, K, J, F.



Tra B e C (lunghezza 11) viene posato un tubo.

Nei tratti di lunghezza 12 non viene posato nessun tubo, perché ogni volta si formerebbe un ciclo; sicuramente riesci a scoprire tu quale.

Questa è l'informatica!

In informatica, la pompa, gli irrigatori e i tratti di collegamento formano complessivamente un *grafo*. Un grafo è una struttura matematica costituita da oggetti detti *nodi* (o vertici) e da collegamenti tra coppie di essi, detti *archi*. Nel nostro problema, la pompa e gli irrigatori rappresentano i nodi, mentre i tratti per i tubi costituiscono gli archi. A ciascun arco è associato un valore numerico corrispondente ai metri di tubo necessari, denominato *peso* dell'arco.»

I grafi sono ampiamente utilizzati per modellare reti di ogni genere, come reti stradali, ferroviarie, elettriche o di telecomunicazioni. Poiché la posa di cavi o binari comporta dei costi elevati, si ricerca spesso un sottografo in cui tutti i nodi siano connessi tra loro utilizzando il minor numero possibile di archi e riducendo al minimo il costo complessivo (ovvero la somma dei pesi). In informatica un sottografo di questo tipo si chiama *albero di copertura minimo*. La strategia descritta sopra per trovare un albero di copertura minimo è nota come *algoritmo di Kruskal*.

Questo algoritmo appartiene alla categoria dei cosiddetti algoritmi greedy (o golosi). Durante la costruzione della soluzione, tali algoritmi compiono a ogni passo la scelta localmente ottimale (in questo caso: selezionare sempre il tratto più corto disponibile), senza valutare gli effetti a lungo termine sulle fasi successive. Nel caso dell'albero di copertura minimo, questa strategia locale garantisce di raggiungere la soluzione globalmente ottimale. Nella pratica gli algoritmi «golosi» fanno risparmiare molto denaro e molte risorse, perché trovano soluzioni in fretta e sono facili da implementare. Tuttavia trovano soluzioni ottimali solo per alcuni problemi. Prima di voler risolvere un problema in modo «goloso», bisognerebbe quindi dimostrare che fa parte di questi problemi.

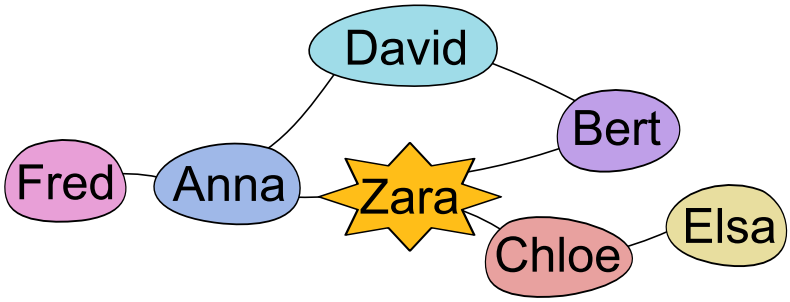
Parole chiave e siti web

- Algoritmo di Kruskal: `https://it.wikipedia.org/wiki/Algoritmo_di_Kruskal`

 © Castoro Informatico 2026, SSII

5. Consigli dai contatti

A Zara piace guardare i video su ZickZock. Per proporre nuovi video, ZickZock usa i contatti dei suoi utenti. L'immagine mostra i contatti di Zara: Anna, Bert e Chloe sono contatti *diretti*, mentre David, Elsa e Fred sono contatti *indiretti*.



ZickZock propone a una persona tutti i video che i suoi contatti (diretti e indiretti) hanno visto e che lei non ha ancora visto, ma soltanto se:

- un contatto diretto ha visto almeno due video uguali a quelli visti dalla persona;
- un contatto indiretto ha visto almeno tre video uguali a quelli visti dalla persona.

La tabella mostra quali video hanno visto Zara e i suoi contatti (✔).

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

Quali nuovi video propone ZickZock a Zara?

Soluzione

ZickZock propone a Zara i video  e .

Si trova la risposta corretta analizzando la tabella dei video visti esattamente come fa ZickZock:

Persona	**Contatto**								
Zara		✔	✔	✔	✔				
Anna	diretto	✔				✔	✔	✔	
Bert	**diretto**			✔	✔			✔	
Chloe	diretto	✔					✔	✔	✔
David	indiretto		✔			✔	✔		
Elsa	**indiretto**	✔	✔	✔			✔		
Fred	indiretto			✔			✔	✔	✔

Nella prima riga vediamo quali video ha visto Zara. Ora confrontiamo questi video con quelli visti dai suoi contatti.

- Per prima cosa consideriamo i contatti diretti Anna, Bert e Chloe. Bert è l'unico contatto diretto che ha visto almeno due video uguali a quelli di Zara (✔). Perciò ZickZock propone l'unico video che Bert ha visto e Zara non ancora (★). ZickZock non propone invece i video di Anna e di Chloe, poiché non hanno visto almeno due video uguali a quelli di Zara.
- Dopodiché consideriamo i contatti indiretti David, Elsa e Fred. Elsa è l'unico contatto indiretto che ha visto almeno tre video uguali a quelli di Zara. Perciò ZickZock propone l'unico video che Elsa ha visto e Zara non ancora. ZickZock non propone invece i video di David e di Fred, poiché non hanno visto almeno tre video uguali a quelli di Zara.

Questa è l'informatica!

ZickZock è un *sistema di raccomandazione*. Forse conosci già qualcosa di simile grazie ad app come Netflix, YouTube o Spotify, che ti propongono video, canzoni o altri contenuti che potrebbero piacerti.

L'app ZickZock di questo compito usa una forma semplice di un metodo che in informatica si chiama *filtraggio collaborativo*, in cui i suggerimenti nascono dal confronto con altre persone. L'algoritmo di suggerimento dell'app cerca altri utenti a cui piacciono le stesse cose che piacciono a te, o cose simili. Se questi utenti hanno visto un video o ascoltato una canzone che tu non conosci ancora, l'app può proporti proprio quel video o quella canzone.

In questo compito non sono stati considerati tutti gli altri utenti, ma soltanto i contatti. Anche i social network come Facebook o LinkedIn usano i contatti o altre informazioni «sociali» per i loro suggerimenti: possono per esempio tenere conto di chi è collegato con chi, di quali gruppi o temi interessano a una persona o di quali post guardano persone simili. In questo modo possono suggerire nuovi contatti, gruppi, post o contenuti.

 © Castoro Informatico 2026, SSII

I sistemi di raccomandazione possono però usare anche altri metodi. Nelle *raccomandazioni basate sui contenuti*, l'algoritmo dell'app tiene conto soprattutto dei contenuti che ti hanno interessato finora. Se in un'app di notizie leggi spesso articoli sullo spazio, l'app potrebbe proporti altri articoli su razzi, pianeti o nuove missioni spaziali.

Questi suggerimenti sono utili, perché ti aiutano a trovare più facilmente cose che corrispondono ai tuoi interessi. Tuttavia, possono anche contribuire a creare una *bolla di filtraggio*: l'app ti mostra soprattutto cose simili a quelle che ti piacciono già, e così rischi di perderti cose importanti o sorprendenti di altri ambiti.

Parole chiave e siti web

- Sistema di raccomandazione: `https://it.wikipedia.org/wiki/Sistema_di_raccomandazione`
- Filtraggio collaborativo: `https://it.wikipedia.org/wiki/Collaborative_filtering`
- Bolla di filtraggio: `https://it.wikipedia.org/wiki/Bolla_di_filtraggio`

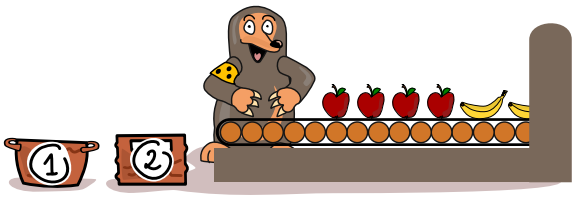


© Castoro Informatico 2026, SSII

6. Il metodo di Muddy

Su un nastro trasportatore arriva della frutta in quest'ordine: prima le mele, poi le banane e infine le pere.

La talpa Muddy vuole sapere se sul nastro trasportatore arriva lo stesso numero di frutti per ciascun tipo. Muddy sa distinguere le mele, le banane e le pere. Purtroppo non sa contare; ha però a disposizione due cesti, che sa distinguere l'uno dall'altro.



Con i frutti e i cesti può fare le seguenti azioni:



mettere un frutto in uno dei cesti



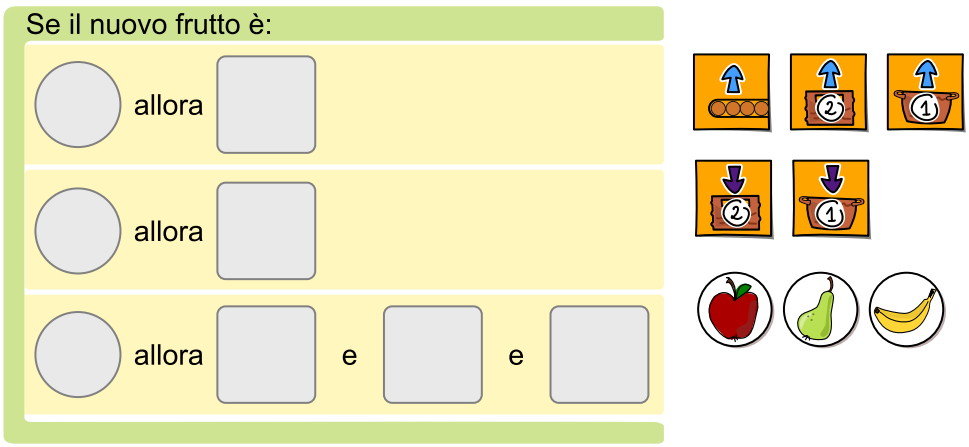
prendere un frutto qualsiasi da uno dei cesti (Muddy non può vedere quale) e metterlo da parte



prendere un frutto dal nastro e metterlo da parte

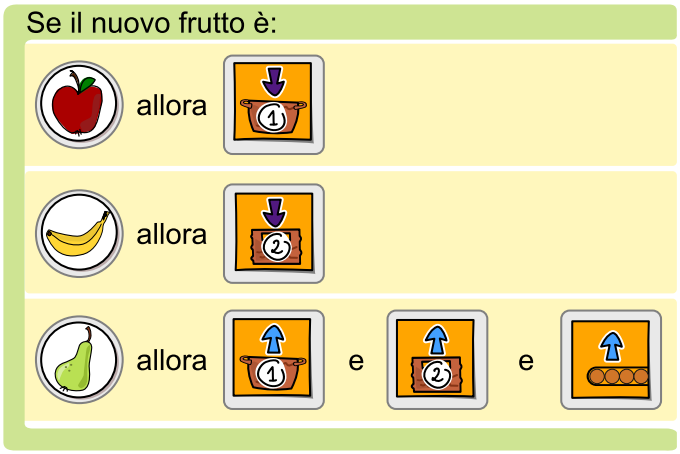
Con queste azioni Muddy ha ideato un metodo per gestire i frutti che arrivano sul nastro trasportatore. Applica questo metodo a ogni nuovo frutto che gli arriva e continua finché il nastro trasportatore non è vuoto, oppure finché non prova a prendere un frutto da un cesto vuoto. Se a quel punto sia il nastro trasportatore sia i due cesti sono vuoti, allora sul nastro è arrivato lo stesso numero di mele, banane e pere.

Come funziona il metodo di Muddy?



Soluzione

Muddy ha ideato questo metodo: mette tutte le mele nel cesto 1 e tutte le banane nel cesto 2, o viceversa. Dopodiché, per ogni pera sul nastro trasportatore prende esattamente una mela e una banana dai cesti e infine mette da parte anche la pera.



Solo se sul nastro arriva lo stesso numero di mele, banane e pere, i due cesti e il nastro trasportatore si svuotano nello stesso momento. Guardando più da vicino, ci sono diversi casi in cui Muddy smette di applicare il suo metodo:

Caso 1: almeno un cesto è vuoto, ma sul nastro c'è ancora almeno una pera. Allora, ...

- se il cesto delle mele è vuoto, sul nastro sono arrivate più pere che mele;
- se il cesto delle banane è vuoto, sul nastro sono arrivate più pere che banane.

Caso 2: il nastro trasportatore è vuoto, ma in almeno un cesto ci sono ancora dei frutti. Allora, ...

- se nel cesto ci sono ancora delle mele, sul nastro sono arrivate meno pere che mele;
- se nel cesto ci sono ancora delle banane, sul nastro sono arrivate meno pere che banane.

Caso 3: il nastro trasportatore e i due cesti sono vuoti. Allora sul nastro è arrivato lo stesso numero di frutti per ciascun tipo.

Questa è l'informatica!

Muddy usa due cesti per confrontare il numero dei frutti. Più precisamente, usa i cesti per «memorizzare» le mele e le banane finché non arrivano le pere. Poiché tiene le mele e le banane separate, per ogni pera sul nastro può prendere, senza guardare, un frutto da ciascun cesto ed essere sicuro di aver preso in tutto una mela e una banana.

In informatica esiste una struttura simile a quella di questo compito: è composta da un nastro di input (come il nastro trasportatore di Muddy) e da due *pile* (cioè memorie, come i cesti di Muddy) e si chiama *automa a due pile*. A differenza dei cesti di Muddy, però, nelle pile regna l'ordine: ciò che è stato messo per ultimo nella pila viene tolto per primo.

In generale, gli *automi* hanno un ruolo importante nell'informatica teorica. Un automa legge dei simboli di input: a seconda del simbolo letto e del suo stato attuale, esegue delle azioni e passa a uno stato successivo. L'input viene accettato se la parola è stata letta completamente ed è stato raggiunto uno stato finale. L'insieme di tutte le parole accettate costituisce il linguaggio che l'automa riconosce. Un automa definisce quindi un *linguaggio formale*, che in alternativa può essere descritto anche da una *grammatica formale*.

Nell'informatica teorica esiste infatti un legame fondamentale tra i *linguaggi formali* (insiemi di parole), le *grammatiche formali* (gli insiemi di regole che generano questi linguaggi) e gli *automi* (i modelli di macchina che riconoscono, o accettano, questi linguaggi). Determinati tipi di automi corrispondono a determinati tipi di grammatiche, e quindi di linguaggi. Per esempio, un linguaggio accettato da un automa a pila con una sola pila viene chiamato, nella *gerarchia di Chomsky* dei linguaggi formali, linguaggio libero dal contesto. I tipici linguaggi informatici, come i linguaggi di programmazione o di markup (per es. HTML), sono liberi dal contesto e possono essere elaborati rapidamente con algoritmi efficienti.

Con una sola pila, o un solo cesto, Muddy potrebbe confrontare due tipi di frutta; con due pile, tre. E così via? No: in realtà è già la seconda pila a fare la differenza decisiva, e ulteriori pile non portano alcun vantaggio. In informatica è infatti noto che un automa a due pile è già in grado di simulare una macchina di Turing. La macchina di Turing si trova in cima alla gerarchia di Chomsky e può riconoscere qualsiasi linguaggio formale che sia calcolabile. La macchina di Turing, e quindi anche l'automa a due pile, è potente quanto qualsiasi computer moderno.

Parole chiave e siti web

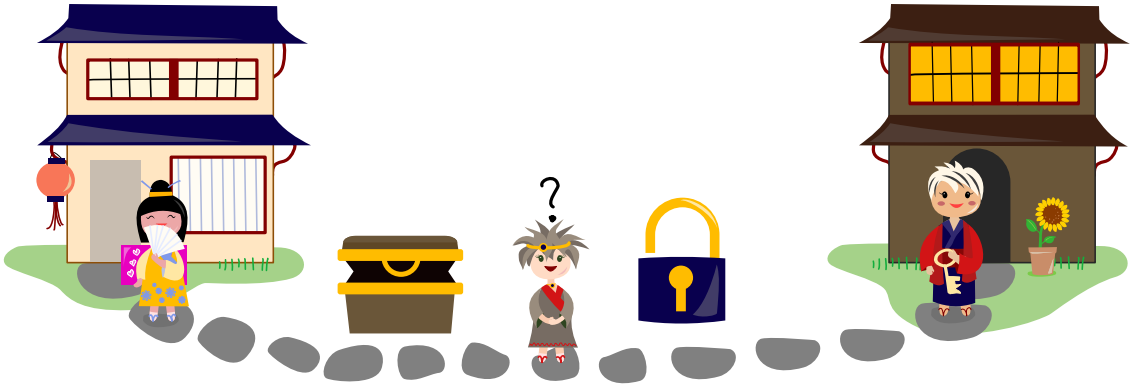
- linguaggio libero dal contesto (CFL – context-free language): `https://it.wikipedia.org/wiki/Linguaggio_libero_dal_contesto`
- automa a due pile: `https://en.wikipedia.org/wiki/Two-Stack_Push_Down_Automaton` (`https://it.wikipedia.org/wiki/Automa_a_pila`)



© Castoro Informatico 2026, SSII

7. Il lucchetto di Benji


Aika vuole dare qualcosa a Benji  senza che nessun altro sappia che cosa sia. Per questo lo mette in una cassetta. Aika e Benji preferiscono restare a casa, ma Mana va regolarmente a trovarli entrambi e può trasportare la cassetta per loro.



Affinché nemmeno Mana sappia che cosa Aika vuole dare a Benji, Aika chiude la cassetta con uno speciale lucchetto di Benji:

- Aika può chiudere il lucchetto senza chiave: basta farlo scattare.
- Solo Benji può aprire il lucchetto, poiché ha l'unica chiave.
- Aika sa con certezza che il lucchetto è di Benji, poiché lui può contrassegnare un lucchetto in un modo che nessun altro è in grado di imitare.

Aika e Benji hanno pensato a una sequenza di passi con cui Mana può trasportare la cassetta avanti e indietro tra Benji e Aika, senza che Mana possa in alcun modo scoprire che cosa Aika vuole dare a Benji. Purtroppo i passi si sono mescolati.

Questi sono i passi. Qual è l'ordine corretto?

- *Mana porta la cassetta ad Aika.*
- *Aika mette qualcosa nella cassetta e la chiude con il lucchetto che si trovava nella cassetta.*
- *Benji mette nella cassetta il lucchetto contrassegnato da lui e tiene la chiave.*
- *Benji apre il lucchetto con la chiave.*
- *Benji prende il lucchetto aperto e lo contrassegna.*
- *Aika dà la cassetta a Mana.*
- *Benji dà la cassetta a Mana.*
- *Mana porta la cassetta a Benji.*

Soluzione

Aika vuole mandare qualcosa a Benji nella cassetta, senza che Mana possa in alcun modo scoprire che cosa sia. Per farlo, Benji deve prima dare ad Aika, senza la chiave, il suo lucchetto speciale contrassegnato da lui (o *firmato*):

1. Benji prende il lucchetto aperto e lo contrassegna.
2. Benji mette nella cassetta il lucchetto contrassegnato da lui e tiene la chiave.
3. Benji dà la cassetta a Mana.
4. Mana porta la cassetta ad Aika.

Ora Aika ha una cassetta e un lucchetto. Dal contrassegno riconosce che si tratta del lucchetto di Benji e che quindi solo lui può aprirlo. Perciò Aika può mettere tranquillamente qualcosa nella cassetta, chiuderla con il lucchetto di Benji e rimandarla a Benji, che potrà riaprirla con la sua chiave:

5. Aika mette qualcosa nella cassetta e la chiude con il lucchetto che si trovava nella cassetta.
6. Aika dà la cassetta a Mana.
7. Mana porta la cassetta a Benji.
8. Benji apre il lucchetto con la chiave.

Solo Benji può aprire il lucchetto, e quindi la cassetta, poiché solo lui ha la chiave.

È fondamentale che il lucchetto sia contrassegnato e che nessuno tranne Benji sia in grado di imitare il contrassegno. Senza questo, Mana potrebbe prendere un altro lucchetto di cui ha la chiave e applicarvi il contrassegno di Benji. Durante il tragitto da Benji ad Aika, Mana scambierebbe poi il lucchetto di Benji con il suo. Aika riconoscerebbe il contrassegno di Benji senza accorgersi di quello che ha fatto Mana, e chiuderebbe quindi la cassetta con quel lucchetto. Più tardi, durante il tragitto da Aika a Benji, Mana potrebbe aprire il lucchetto con la sua chiave e scoprire così che cosa Aika vuole dare a Benji. Se Mana avesse addirittura un lucchetto che anche Benji potesse aprire con la sua chiave, potrebbe richiudere la cassetta, e nemmeno Benji si accorgerebbe di quello che ha fatto Mana.

Questa è l'informatica!

Il lucchetto di questo compito, che chiunque può chiudere senza chiave ma che solo una persona con la chiave può aprire, funziona in modo simile alla *crittografia asimmetrica*. Nella crittografia asimmetrica ci sono due chiavi: una *chiave pubblica* per *cifrare* un messaggio e una *chiave privata*, senza la quale un messaggio non può essere *decifrato*, se non con uno sforzo enorme e insostenibile. La chiave pubblica può finire nelle mani di chiunque e spesso è addirittura consultabile liberamente su Internet. La chiave privata invece non deve averla nessuno tranne il destinatario, poiché solo lui deve poter decifrare il messaggio.

La crittografia asimmetrica è sicura – a patto che lo *scambio delle chiavi* possa essere protetto. Soprattutto quando due persone non si incontrano (o non possono incontrarsi) di persona, deve essere chiaro che una chiave pubblica appartiene davvero a una determinata persona. Questo può avvenire, per esempio, dimostrando la propria identità a terze parti affidabili (le cosiddette *autorità di*

 © Castoro Informatico 2026, SSII

certificazione, che di solito sono preimpostate nel sistema operativo o nel browser), le quali confermano poi che una chiave pubblica appartiene a una determinata persona. Se questo non funzionasse, un malintenzionato potrebbe sostituire la chiave pubblica del destinatario con la propria, intercettare il messaggio e leggerlo o modificarlo.

Le certificazioni, o meglio i certificati, vengono usati molto spesso nei sistemi informatici, senza che ce ne rendiamo conto. Se per esempio si visita `https://concorso.castoro-informatico.ch/`, la trasmissione della pagina web è cifrata (lo si riconosce dal protocollo «https»). Guardando meglio, si scopre che il dominio ha un certificato, confermato come valido per questo dominio da un'autorità di certificazione (in questo caso «Let's Encrypt»).

Un altro aspetto da non sottovalutare è la gestione sicura delle chiavi private. Capita regolarmente di leggere notizie di chiavi private diventate di dominio pubblico, per esempio a causa di *social engineering*, di intrusioni nei sistemi informatici o semplicemente di errori di elaborazione. Può bastare usare un browser che sincronizza le password in un cloud, oppure che un malintenzionato installi una propria autorità di certificazione fraudolenta nel sistema operativo o nel browser. In alcuni Paesi gli enti statali si arrogano il diritto di violare in questo modo comunicazioni altrimenti sicure. Se una comunicazione ritenuta sicura non lo è più, le conseguenze per le persone coinvolte possono essere molto spiacevoli.

Parole chiave e siti web

- Crittografia asimmetrica: `https://it.wikipedia.org/wiki/Crittografia_asimmetrica`
- Crittografia end-to-end: `https://it.wikipedia.org/wiki/Crittografia_end-to-end`
- `https://it.wikipedia.org/wiki/Transport_Layer_Security`
- `https://it.wikipedia.org/wiki/Chat_Control`

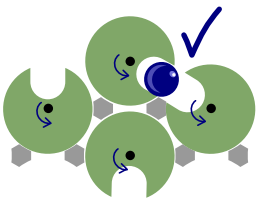


© Castoro Informatico 2026, SSII

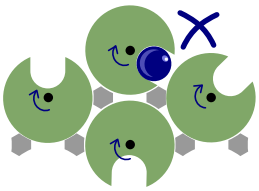
8. Gira che ti rigira

Una pallina deve attraversare una disposizione di ruote identiche , ciascuna delle quali ha un incavo in cui la pallina entra perfettamente.

Un motore fa girare lentamente le ruote, tutte nello stesso senso e alla stessa velocità costante. Con un interruttore è possibile cambiare in qualsiasi momento il senso di rotazione delle ruote.



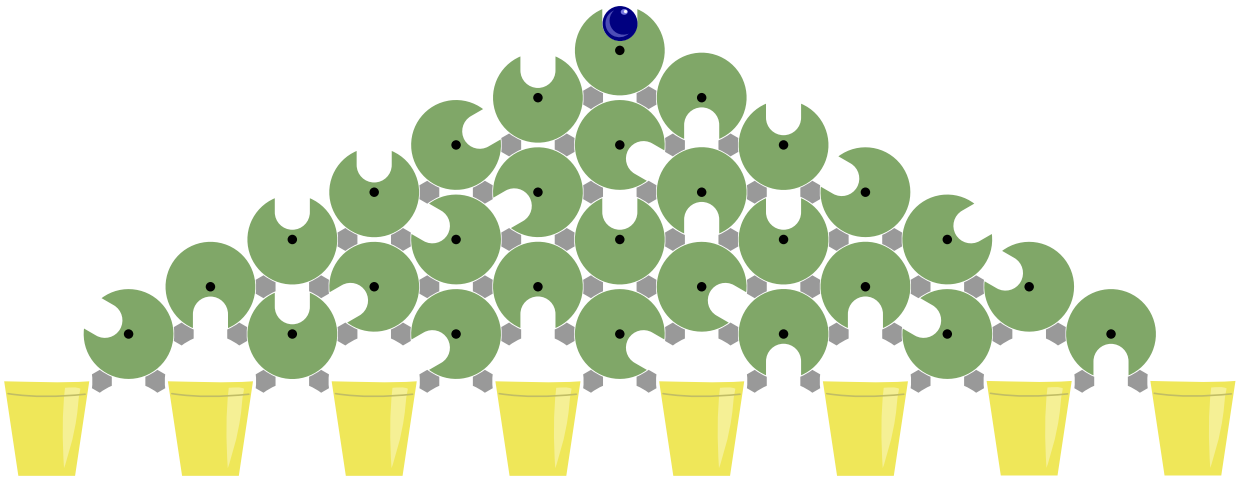
Durante la rotazione, gli incavi di due ruote vicine possono trovarsi uno di fronte all'altro: in quel momento la pallina rotola verso il basso.



In questa disposizione due incavi non si trovano mai uno di fronte all'altro, perciò la pallina resta nella ruota superiore.

I fermi tra le ruote impediscono che la pallina rotoli nello spazio tra due ruote o che cada fuori dalla disposizione.

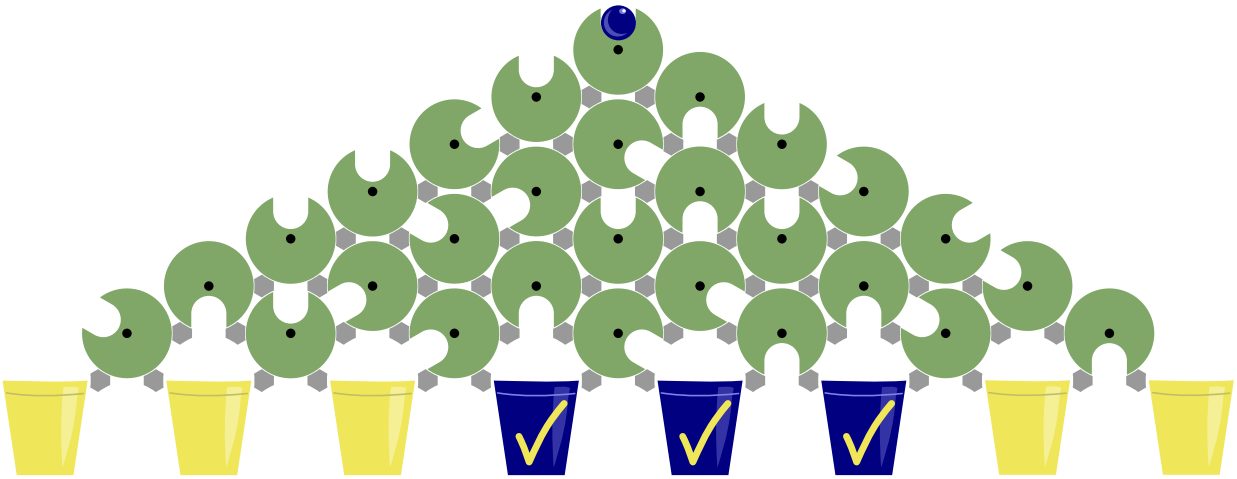
Qui vedi una disposizione di ruote più grande. La pallina viene messa nella ruota più in alto, mentre sotto le ruote si trovano alcuni secchi.



In quali secchi può finire la pallina?

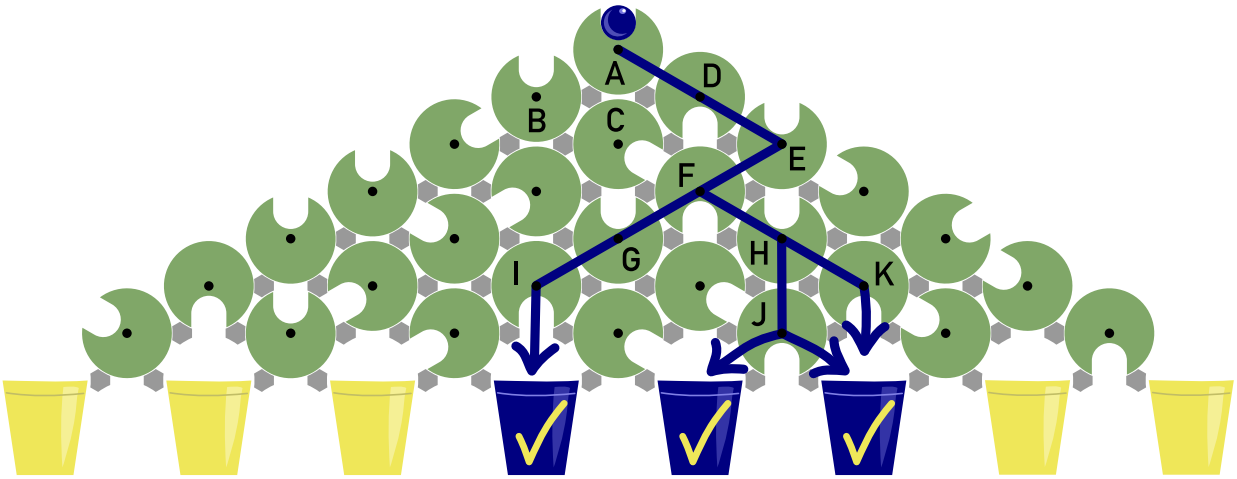
Soluzione

La risposta corretta è la seguente:



Per trovare la risposta corretta potremmo provare a simulare mentalmente le rotazioni di tutte le ruote, ma sarebbe molto difficile. Ci aiutiamo perciò con un'osservazione: affinché la pallina possa rotolare da una ruota all'altra, in quel momento gli incavi delle due ruote devono essere rivolti in direzioni opposte. Poiché tutte le ruote girano nello stesso senso e alla stessa velocità, possiamo constatare che, se due incavi sono rivolti in direzioni opposte in un certo momento, lo sono anche in qualsiasi altro momento.

Ciò significa che, nella disposizione iniziale, possiamo verificare per ogni coppia di ruote vicine se i loro incavi sono rivolti in direzioni opposte. Se è così, grazie all'osservazione precedente sappiamo che la pallina può rotolare dalla ruota superiore a quella inferiore. In caso contrario, gli incavi non potranno mai trovarsi uno di fronte all'altro e la pallina non rotolerà mai dall'alto verso il basso. Cominciamo la verifica dalla ruota più in alto, in cui viene messa la pallina, e in questo modo possiamo determinare tutti i percorsi che la pallina può seguire attraverso la disposizione.



- La pallina viene messa nella ruota A. Delle tre ruote vicine (B, C e D), solo la ruota D ha l'incavo rivolto nella direzione opposta rispetto all'incavo della ruota A. Dalla ruota A la pallina può quindi rotolare soltanto nella ruota D.

 © Castoro Informatico 2026, SSII

- Allo stesso modo, dalla ruota D la pallina può rotolare soltanto nella ruota E.
- Dalla ruota E la pallina può rotolare soltanto nella ruota F.
- Dalla ruota F ci sono più possibilità: sia l'incavo della ruota G sia quello della ruota H sono rivolti nella direzione opposta rispetto all'incavo della ruota F. A seconda del senso di rotazione, la pallina rotola nella ruota G oppure nella ruota H.
- Dalla ruota G la pallina può rotolare soltanto nella ruota I, da cui finisce nel secchio di sinistra fra i tre segnati con la spunta.
- Dalla ruota H la pallina può rotolare nella ruota K oppure nella ruota J.
- Dalla ruota K finisce nel secchio di destra fra i tre, mentre dalla ruota J finisce nel secchio centrale o in quello di destra.

Questa è l'informatica!

In questo compito dovevi analizzare il comportamento di una disposizione di ruote per scoprire in quali secchi la pallina *può* finire. Chi ha progettato questa disposizione, o l'intero sistema di ruote, potrebbe verificare allo stesso modo che la pallina finisca esattamente nei secchi in cui *deve* finire. In informatica, la verifica del fatto che un sistema (informatico) possieda determinate proprietà o soddisfi determinati requisiti si chiama *verifica del sistema*. Quanto più gravi possono essere le conseguenze di un malfunzionamento, tanto più importante è la verifica: per esempio, in un software che visualizza le radiografie bisognerebbe verificare che l'immagine non venga mai mostrata specchiata da sinistra a destra, poiché ciò potrebbe avere conseguenze fatali per le pazienti e i pazienti.

Spesso, però, è difficile dimostrare come si comporta un sistema complesso. Come primo passo si potrebbe provare a *testare* il sistema. Tuttavia, lo si può fare sempre e solo in un numero limitato di situazioni diverse, mentre spesso le situazioni possibili sono infinite: pensa per esempio ai diversi testi che si possono scrivere a un chatbot, alle condizioni meteorologiche nella guida autonoma o alle possibili sequenze di rotazioni in questo compito. In questi casi i test non bastano per essere sicuri che il sistema soddisfi sempre i requisiti.

Un altro approccio è la *verifica formale*, con cui si dimostra formalmente come si comporta il sistema in determinate condizioni. Per le ruote di questo compito abbiamo mostrato che la pallina può rotolare da una ruota all'altra solo se i loro incavi sono rivolti in direzioni opposte. Quando un sistema cambia dinamicamente, ma una certa proprietà rimane sempre la stessa, si parla di un *invariante*. Gli invarianti sono uno strumento importante per capire i sistemi complessi, non solo in informatica.

Parole chiave e siti web

- Verifica: `https://en.wikipedia.org/wiki/Verification_and_validation`, `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`
- Verifica dei programmi (verifica formale): `https://it.wikipedia.org/wiki/Verifica_formale`,

https://www.studysmarter.de/studium/ingenieurwissenschaften/automatisierung-in-der-informationstechnologie/programmverifikation/, https://wirtschaftslexikon.gabler.de/definition/programmverifikation-46218

- Invarianti: https://loeh.app.uni-regensburg.de/seminars/invarianten.pdf, https://it.wikipedia.org/wiki/Invariante_(informatica), https://en.wikipedia.org/wiki/Invariant_(mathematics)

9. Il ramyeon perfetto

Lo chef Dong gestisce un ristorante di ramyeon, una zuppa coreana con i noodles. Alla ricerca del gusto perfetto, prova diverse impostazioni: per la temperatura dell'acqua e il tempo di cottura, ma anche per le quantità di uova, cipollotti e aglio.



Ogni volta che Dong prepara una nuova ricetta, i clienti abituali gli danno una valutazione su una scala da 0 a 100. Dong sa che per ciascuna impostazione esiste un unico valore ottimale, indipendentemente dalle altre impostazioni. Sa inoltre che, quanto più un'impostazione si allontana dal suo valore ottimale, tanto peggiore diventa la valutazione. Ha già provato alcune ricette e ha annotato i risultati in una tabella.

Utilizza il codice QR per accedere alla versione interattiva di questo esercizio.

Soluzione

Una valutazione perfetta di 100 si ottiene con una temperatura dell'acqua di 100 °C, 1 uovo, 20 g di cipollotti, 1 cucchiaino di aglio e un tempo di cottura di 270 s.

Sappiamo che il valore ottimale di ciascuna impostazione non dipende dalle altre impostazioni. Possiamo sfruttarlo determinando il valore ottimale di un'impostazione alla volta. I test svolti finora mostrano già come funziona per alcune impostazioni.

Con le ricette 2 e 3, confrontate con la ricetta 1, si trova per esempio la temperatura ottimale dell'acqua. Dei tre valori provati, 100 °C dà la valutazione migliore. A partire da lì la valutazione scende: con 90 °C è peggiore che con 100 °C, e con 80 °C è ancora peggiore. Poiché sappiamo che la valutazione peggiora quanto più il valore di un'impostazione si allontana da quello ottimale, è chiaro, senza bisogno di altri test, che temperature ancora più basse porterebbero a valutazioni ancora peggiori. La temperatura dell'acqua non può superare i 100 °C, quindi non è possibile ottenere una valutazione migliore con temperature più alte. Perciò 100 °C è il valore ottimale, e lo manteniamo per i test successivi.

Con le ricette da 3 a 7 si trova la quantità ottimale di cipollotti. La valutazione migliora aumentando la quantità da 10 g a 15 g e poi a 20 g, ma peggiora con un ulteriore aumento a 25 g. La quantità ottimale è quindi di 20 g.

Allo stesso modo, con altri test, si possono determinare i valori ottimali delle altre impostazioni. Per le uova la valutazione peggiora sia con meno di un uovo sia con più di un uovo; il numero ottimale è quindi 1. Per l'aglio la quantità ottimale è di 1 cucchiaino, mentre il tempo di cottura ottimale è di 270 secondi.

Portando ciascuna impostazione al suo valore ottimale, si ottiene una valutazione di 100.

Questa è l'informatica!

In questo compito bisogna scegliere le impostazioni di una ricetta in modo tale che il ramyeon sia il più buono possibile, cioè ottenga la valutazione più alta possibile. Si tratta di un esempio di *problema di ottimizzazione*: si cercano valori per diversi parametri (qui le impostazioni) che minimizzino o massimizzino una funzione obiettivo (qui la valutazione). Per risolvere problemi di questo tipo, l'informatica dispone di diversi *algoritmi di ottimizzazione*.

Gli algoritmi di ottimizzazione sono molto importanti per l'*apprendimento automatico* (machine learning). In questo ambito si «addestrano» modelli con moltissimi parametri (a volte migliaia di miliardi!), ottimizzandoli in modo da ottenere i risultati migliori possibili sui cosiddetti dati di addestramento. Uno degli algoritmi più usati a questo scopo è la *discesa del gradiente*, che sfrutta la pendenza della funzione obiettivo per correggere molti parametri contemporaneamente nella direzione giusta.

Quale algoritmo di ottimizzazione sia il più adatto dipende dalle caratteristiche del problema. In questo compito sappiamo che i parametri influiscono sulla valutazione in modo indipendente l'uno

dall'altro. Questo semplifica molto l'ottimizzazione, ma nei problemi reali di solito non è così: per esempio, è logico che con una temperatura dell'acqua più bassa serva un tempo di cottura più lungo. Un algoritmo adatto al caso di parametri indipendenti è la *discesa per coordinate* (in inglese *coordinate descent*). Con questo metodo si modifica un solo parametro alla volta e si osserva l'effetto sul risultato. In questo modo i singoli parametri vengono regolati uno dopo l'altro, per trovare passo dopo passo una soluzione sempre migliore.

Parole chiave e siti web

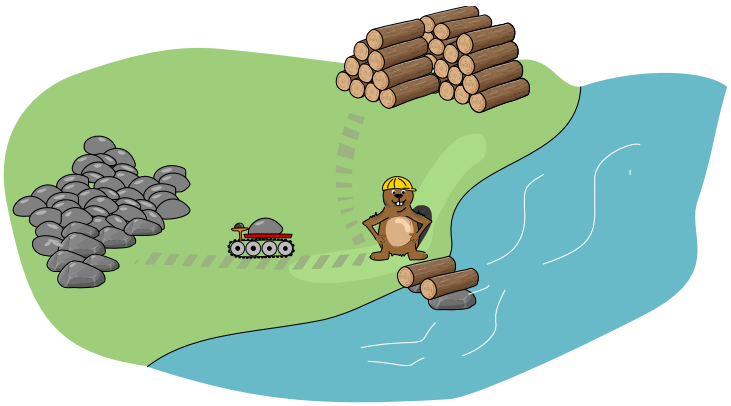
- Problema di ottimizzazione:
  `https://it.wikipedia.org/wiki/Problema_di_ottimizzazione`
- Algoritmi di ottimizzazione:
  `https://it.wikipedia.org/wiki/Ottimizzazione_(matematica)`
- Discesa del gradiente: `https://it.wikipedia.org/wiki/Discesa_del_gradiente`
- Discesa per coordinate (coordinate descent):
  `https://en.wikipedia.org/wiki/Coordinate_descent`



© Castoro Informatico 2026, SSII

10. La diga dei castori

I castori vogliono costruire una diga aggiungendo sempre prima una pietra e poi un pezzo di tronco – finché ci sono ancora pietre *e* tronchi a disposizione. Un robot li aiuta nella costruzione.



Un programma per il robot è una sequenza di istruzioni numerate. L'esecuzione di un programma inizia sempre dall'istruzione 1. Le istruzioni disponibili sono le seguenti:



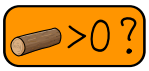
Aggiungi una pietra alla diga e vai all'istruzione successiva.



Aggiungi un tronco alla diga e vai all'istruzione successiva.



Se ci sono ancora pietre, vai all'istruzione successiva; altrimenti saltala e vai a quella dopo.



Se ci sono ancora tronchi, vai all'istruzione successiva; altrimenti saltala e vai a quella dopo.

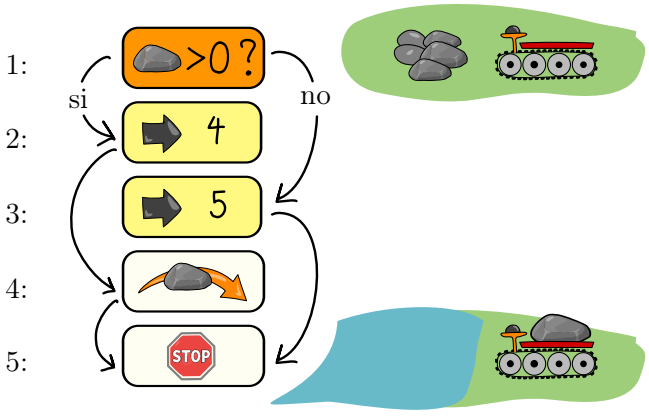


Vai all'istruzione numero X.



Interrompi l'esecuzione del programma.

Ecco un programma di esempio: con questo programma il robot aggiunge esattamente una pietra, se ce ne sono ancora.



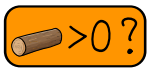
I castori vogliono scrivere un programma con cui il robot costruisca la diga esattamente come desiderano. Hanno quasi finito: mancano solo i numeri nelle istruzioni «Vai a . . . ».

Aiuta i castori e inserisci correttamente tutti i numeri di istruzione mancanti.

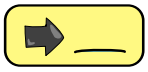
1: 

2: 

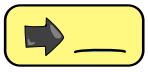
3: 

4: 

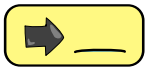
5: 

6: 

7: 

8: 

9: 

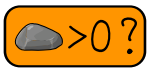
10: 

11: 

 © Castoro Informatico 2026, SSII

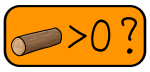
Soluzione

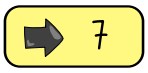
La risposta corretta è la seguente:

1: 

2: 

3: 

4: 

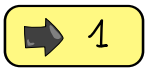
5: 

6: 

7: 

8: 

9: 

10: 

11: 

L'istruzione 1 verifica se ci sono ancora pietre.

In caso affermativo, il robot esegue l'istruzione successiva:

2: 

... e salta all'istruzione 4.

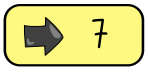
In caso contrario, esegue l'istruzione 3:

3: 

... salta all'istruzione 11 e si ferma. Se non ci sono più pietre, quindi, il programma si ferma.

Se invece ci sono ancora pietre, l'istruzione 4 verifica se ci sono ancora anche dei tronchi.

In caso affermativo, il robot esegue l'istruzione successiva, cioè la 5:

5: 

... salta all'istruzione 7 e aggiunge così una pietra.

In caso contrario, esegue l'istruzione 6:

6: 

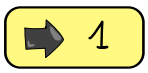
... salta all'istruzione 11 e si ferma. Se non ci sono più tronchi, quindi, il programma si ferma, proprio come succede con l'istruzione 3 per le pietre.

L'istruzione 8 è interessante. Quando il robot la esegue, ha appena aggiunto una pietra con l'istruzione 7 e «sa» che ci sono sia pietre sia tronchi. La prossima cosa da fare è aggiungere un tronco, cioè eseguire l'istruzione 9: il programma deve quindi semplicemente passare all'istruzione successiva. Per questo motivo l'istruzione 8 è in realtà inutile, poiché il programma esegue comunque sempre l'istruzione successiva.

Ora il programma esegue quindi l'istruzione 9 e aggiunge un tronco:

9: 

... passa all'istruzione 10:

10: 

... e salta all'istruzione 1, ricominciando quindi dall'inizio.

Nel complesso, il programma fa sì che il robot aggiunga alla diga prima una pietra e poi un tronco, finché ci sono ancora pietre e tronchi - proprio come vogliono i castori.

Lo stesso risultato si ottiene se le istruzioni 3 e 6 non portano entrambe direttamente all'istruzione 11 (e quindi all'arresto del programma), ma una sola delle due fa una deviazione: o l'istruzione 3 porta all'istruzione 6 (e da lì all'istruzione 11), oppure l'istruzione 6 porta all'istruzione 3 (e da lì all'istruzione 11). Esistono quindi altre due risposte corrette.

Questa è l'informatica!

I programmi che i processori dei computer possono eseguire direttamente sono, come i programmi di questo compito, semplici sequenze di istruzioni: dopo aver eseguito un'istruzione, di norma il processore prosegue con quella successiva. Nel codice macchina del processore ogni istruzione è descritta in binario, e i primi bit (l'*opcode*) indicano di quale istruzione si tratta. Un programma in codice macchina è quindi una lunga sequenza di zeri e di uni.

Agli inizi dell'era dei computer, negli anni Quaranta, i programmi dovevano ancora essere scritti direttamente in codice macchina. Questo lavoro causava moltissimi errori, e i programmi erano difficili da leggere. Per semplificarlo, all'inizio degli anni Cinquanta venne sviluppato il linguaggio assembly, che sostituiva gli opcode binari con abbreviazioni leggibili, i cosiddetti *mnemonici*, come MOV per spostare (move), ADD per sommare (add) o JMP per saltare (jump). Un programma speciale, l'assembler, traduceva poi, tra le altre cose, queste abbreviazioni in codice macchina. L'assembly

è considerato il primo linguaggio di programmazione «di livello superiore»: costruiva un ponte tra l'essere umano e la macchina, senza che il programmatore perdesse il controllo diretto sull'hardware.

Una differenza fondamentale tra l'assembly e i linguaggi di programmazione moderni è che in assembly non esistono né istruzioni condizionali («if-then-else») né istruzioni di ripetizione («for», «while», ...). Condizioni e ripetizioni vengono invece realizzate con l'aiuto di istruzioni di salto (jump), che dicono al processore: «Continua con l'istruzione X invece che con quella successiva.»

Nello sviluppo software moderno, tuttavia, le istruzioni di salto (che, quando esistono, di solito si chiamano «goto») sono considerate il nemico giurato della buona programmazione. Alla fine degli anni Sessanta, l'informatico Edsger W. Dijkstra, con il suo famoso articolo «Go To Statement Considered Harmful», pose le basi della programmazione strutturata.

Alla base di un codice «pulito» e comprensibile nei linguaggi di alto livello c'è la rinuncia completa ai salti incontrollati. Al loro posto si usano strutture di controllo chiare (come cicli e funzioni), che hanno un inizio e una fine ben definiti. In questo modo il codice diventa leggibile, più facile da mantenere e molto meno soggetto a errori.

Parole chiave e siti web

- Linguaggio assembly: `https://it.wikipedia.org/wiki/Linguaggio_assembly`
- Linguaggio macchina: `https://it.wikipedia.org/wiki/Linguaggio_macchina`
- Istruzione di salto (GOTO): `https://it.wikipedia.org/wiki/GOTO`
- Programmazione strutturata: `https://it.wikipedia.org/wiki/Programmazione_strutturata`

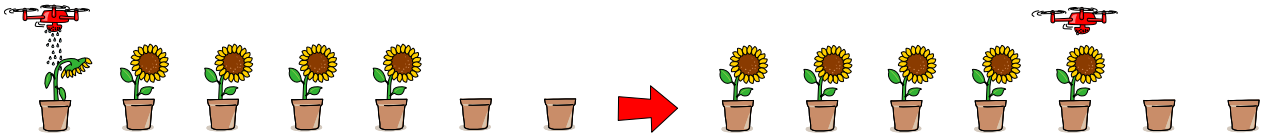


© Castoro Informatico 2026, SSII

11. Fiori dimenticati

Archie ha un drone .

Archie vuole che il drone voli sopra una fila di vasi e annaffi tutti i fiori assetati  che ci sono dentro.



Archie ha scritto il seguente programma per il drone.

Passo 1:

Parti e vola fino al vaso più a sinistra.

Passo 2:

Se nel vaso c'è un fiore assetato, annaffialo.
Altrimenti vola al vaso successivo a destra

Passo 3:

Se nel vaso non c'è nessun fiore, atterra.
Altrimenti vola al vaso successivo a destra e continua con il passo 2.

Il drone esegue i passi del programma uno dopo l'altro, finché non atterra.

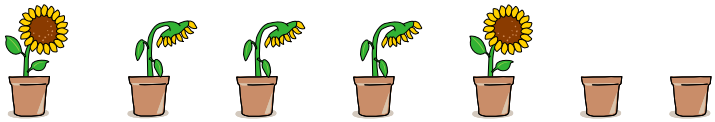
Però il drone non fa sempre quello che vuole Archie.

Indica una fila di vasi in cui un fiore assetato non *viene annaffiato.*



Soluzione

Questa è una risposta corretta:



Il programma di Archie ha un problema. Sia il passo 2 sia il passo 3 eseguono una di due istruzioni possibili, a seconda di cosa contiene il vaso sotto il drone:

Opzioni nel passo 2:

- Se il vaso contiene un fiore assetato → *annaffiare il fiore*
- Se il vaso non contiene un fiore assetato → *volare al vaso successivo a destra*

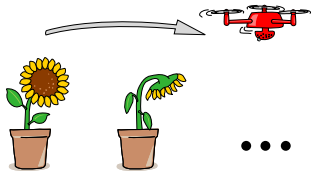
Optioni nel passo 3:

- Se il vaso non contiene nessun fiore → *atterrare*
- Se il vaso contiene un fiore → *volare al vaso successivo a destra*

Questo significa: quando un vaso contiene un fiore assetato, il drone esegue il controllo del passo 2 (innaffiando), e poi nel passo 3 vola al vaso successivo a destra, dove può eseguire di nuovo il passo 2. Complessivamente, il drone «salta» in avanti di **un** solo vaso alla volta.

Se invece un vaso contiene un fiore non assetato (o è vuoto) e il vaso alla sua destra contiene un fiore qualsiasi, il drone vola verso destra sia nel passo 2 che nel passo 3. Di conseguenza, il passo 2 viene saltato per il vaso sorvolato (il cui fiore assetato non viene annaffiato) e viene eseguito solo per il vaso successivo. In questo caso il drone salta di **due** vasi in un colpo solo.

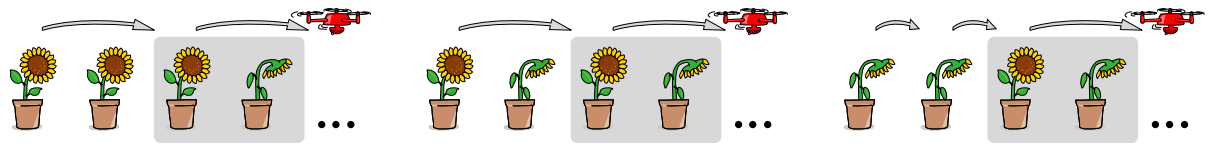
Poiché i vasi vuoti compaiono solo alla fine, possiamo trascurarli e osservare che: un fiore assetato viene saltato e *non* viene annaffiato se alla sua sinistra si trova un fiore non assetato che a sua volta non viene saltato.



Pertanto, è possibile costruire una fila in cui un fiore assetato non viene annaffiato disponendo innanzitutto un fiore non assetato subito a sinistra di quello assetato. Più a sinistra, gli altri fiori (assetati o meno) vanno posizionati in modo tale che, grazie ai salti di 1 o 2 vasi, il drone raggiunga proprio quel fiore non assetato.

Il modo più semplice è posizionare la combinazione  direttamente all'inizio della fila, come nell'esempio mostrato sopra: in questo modo il secondo fiore della serie non viene annaffiato. Tuttavia, qualsiasi altra fila creata seguendo questo principio è una risposta corretta. Qui sotto, ad esempio, sono mostrate tutte e tre le possibilità per fare in modo che il quarto fiore della fila non venga annaffiato:

 © Castoro Informatico 2026, SSII



Questa è l'informatica!

Il drone di questo compito esegue un *algoritmo*. Un algoritmo è una sequenza di istruzioni che risolve un problema concreto. I sistemi informatici come computer, smartphone o droni sono controllati da algoritmi. Di solito gli algoritmi vengono inventati da persone e descritti in linguaggi di programmazione comprensibili per i sistemi informatici. In informatica il programma ottenuto in questo modo si chiama anche *implementazione* dell'algoritmo su cui si basa.

Gli algoritmi, o anche le loro implementazioni, possono contenere errori quando qualcosa durante l'esecuzione non corrisponde alle intenzioni di chi li ha sviluppati. In informatica questi errori vengono spesso chiamati *bug*; per questo trovarli e correggerli si chiama *debugging*. In questo compito è l'algoritmo stesso a contenere un errore: in certi casi un vaso con un fiore assetato non viene annaffiato.

Per verificare se un programma o il suo algoritmo presentano difetti, le sviluppatrici e gli sviluppatori eseguono dei test fornendo differenti dati di input, noti appunto come *casi di test*. Ciascuna delle risposte corrette a questo problema rappresenta un caso di test in grado di portare alla luce l'errore nel codice di Archie.

Parole chiave e siti web

Algoritmo, Debugging, Caso di test

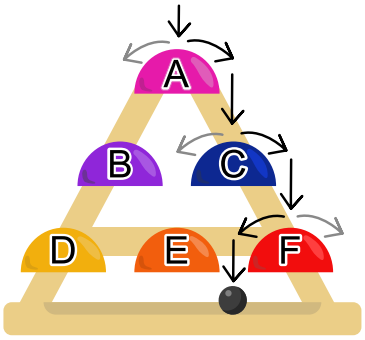
- Algoritmo: `https://it.wikipedia.org/wiki/Algoritmo`
- Debugging: `https://it.wikipedia.org/wiki/Debugging`
- Collaudo del software (test): `https://it.wikipedia.org/wiki/Collaudo_del_software`



© Castoro Informatico 2026, SSII

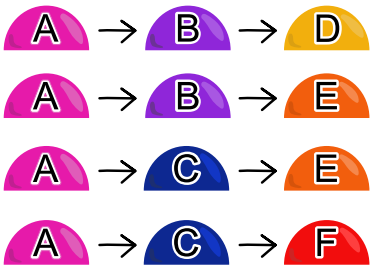
12. Biglie musicali

Marta ha un kit per costruire piste per biglie musicali. Ci sono sei campanelle, da A a F, e ognuna suona in modo diverso quando una biglia ci cade sopra. Le campanelle si possono fissare su una struttura, in sei posizioni. A destra c'è una pista per biglie già pronta.



Una biglia scende lungo la pista dall'alto verso il basso. Lungo il suo percorso cade sulla campanella più in alto, su una campanella della seconda fila e su una della terza fila. Così si sente una melodia di tre note. Su ogni campanella la biglia rotola a caso verso sinistra o verso destra, quindi ci sono diversi percorsi della biglia lungo le campanelle.

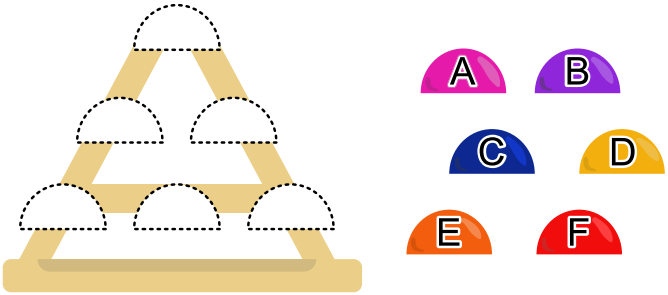
Nella pista illustrata sopra si possono percorrere i seguenti tragitti:



Marta ama molto certe melodie, mentre altre non le piacciono per niente. Perciò vuole costruire una pista in cui siano presenti ...

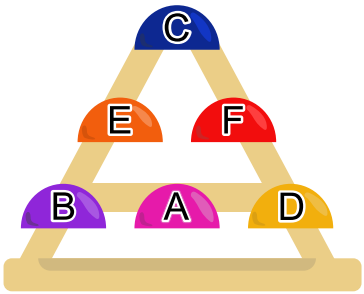
... almeno i seguenti percorsi, ...	**... ma in nessun caso i seguenti percorsi:**
un percorso qualsiasi con le tre campanelle 	tutti i percorsi con le due campanelle 
un percorso qualsiasi con le due campanelle 	il percorso 
il percorso 	

Aiuta Marta a fissare le campanelle nelle posizioni giuste.

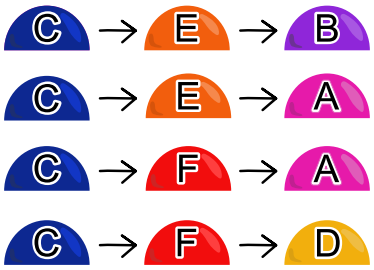


Soluzione

Se le campanelle sono fissate così, . . .



ci sono questi percorsi della biglia:



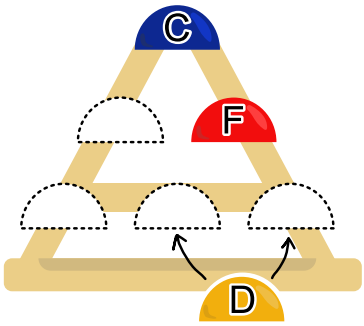
In questo modo sono presenti tutti i percorsi desiderati, ma nessuno di quelli vietati:

- Il percorso C → E → A è un percorso con le tre campanelle A, C, E; il percorso C → E → B è un percorso con le due campanelle B ed E; e c'è anche il percorso C → F → D.
- La biglia non può seguire né un percorso con le note B e F, né il percorso B → A → D.

Come si può determinare questa disposizione? Dobbiamo fare in modo che la biglia possa seguire tutti i percorsi desiderati, evitando quelli vietati. Per riuscirci, consideriamo un percorso desiderato alla volta e successivamente escludiamo quelli vietati. Per farlo, inseriamo nella pista per biglie un percorso desiderato dopo l'altro e poi evitiamo i percorsi vietati.

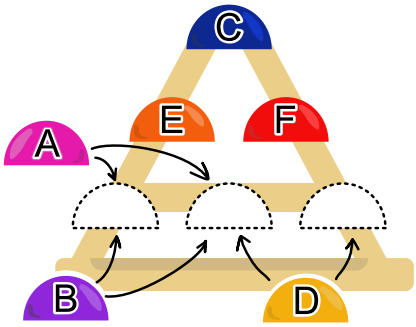
Passo 1: inserire il percorso C → F → D

La campanella C deve essere posizionata in alto, F nella seconda fila e D nella terza. Scegliamo di collocare F a destra; di conseguenza, D dovrà trovare posto al centro o a destra nella terza fila.



Passo 2: inserire i percorsi con A, C, E **e con** B, E

C è già posizionata in alto. Sotto di essa, E deve far parte sia di un percorso con A che di uno con B; perciò E deve trovarsi sopra A e B. Di conseguenza, E andrà nella seconda fila a sinistra, mentre A e B nella terza fila a sinistra o al centro.



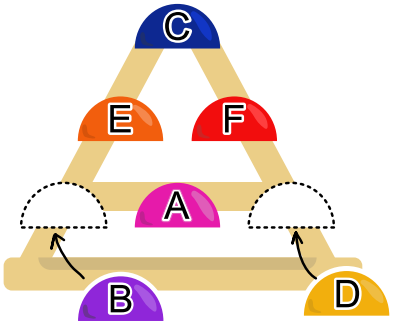
 © Castoro Informatico 2026, SSII

Passo 3: evitare il percorso B → A → D

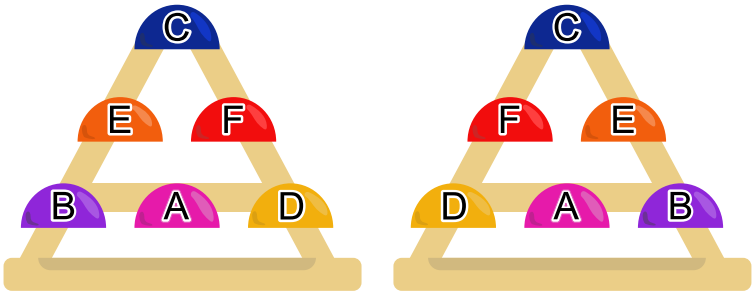
Dato che C è già posizionata in cima, non vi è alcun percorso che possa iniziare con B.

Passo 4: evitare tutti i percorsi con B **e** F

Visto che C è già in cima e F nella seconda fila, dobbiamo solo impedire alla biglia di passare da F a B. Pertanto, nella terza fila posizioneremo B a sinistra, A al centro e D a destra.



In totale ci sono 2 soluzioni corrette, poiché anche la disposizione speculare delle campanelle genera gli stessi 4 percorsi.



Questa è l'informatica!

Hai posizionato e combinato le campanelle in una pista in modo che la biglia segua esclusivamente percorsi desiderati e nessun percorso vietato. Questa configurazione soddisfa le condizioni (i vincoli o constraints) imposte da Marta. In informatica, i cosiddetti *problemi di soddisfacimento di vincoli* (in inglese *constraint satisfaction problems*, CSP) sono estremamente comuni, ad esempio nella pianificazione dell'orario scolastico.

Di norma si tratta di selezionare singoli elementi affinché la loro combinazione soddisfi un insieme di requisiti, giungendo a una soluzione valida. Un approccio ingenuo per risolvere i CSP consisterebbe nel testare le combinazioni una dopo l'altra fino a trovarne una corretta; tuttavia, se le combinazioni possibili sono molte, il carico di lavoro diventa proibitivo persino per un computer.

Una strategia ben più efficiente è il backtracking (ovvero «tornare sui propri passi»): le soluzioni vengono costruite in modo incrementale, passo dopo passo, annullando l'ultima mossa non appena si incontra una contraddizione. In questo modo si scartano immediatamente numerose opzioni non valide. Questo metodo si applica direttamente anche a questo problema: si sceglie una campanella alla volta per ciascuna posizione, verificando che tutte le condizioni siano rispettate; qualora si arrivi a un vicolo cieco, si torna sui propri passi rimpiazzando la scelta precedente.

Parole chiave e siti web

- Problemi di soddisfacimento di vincoli (CSP): `https://it.wikipedia.org/wiki/Problema_di_soddisfacimento_di_vincoli`
- Backtracking: `https://it.wikipedia.org/wiki/Backtracking`

 © Castoro Informatico 2026, SSII

13. Il gioco delle monete

In un gioco ci sono

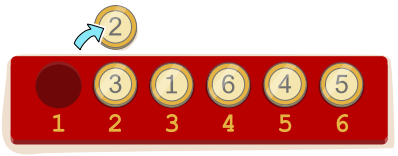
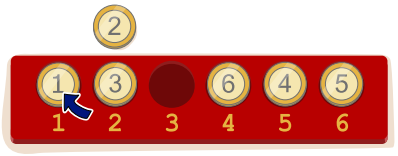
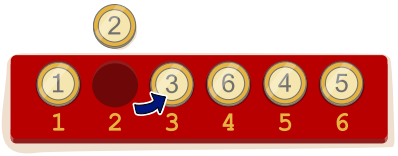
- sei monete, numerate da 1 a 6, e
- una tavola da gioco con i posti numerati da 1 a 6.

All'inizio del gioco le monete vengono disposte a caso sui posti, per esempio così:



Lo scopo del gioco è mettere tutte le monete al posto giusto nel minor numero possibile di turni: la moneta 1 sul posto 1, la moneta 2 sul posto 2 e così via.

Un turno è composto dai seguenti passi:

	Passo	**Esempio**
1	Prendi la prima moneta da sinistra che si trova al posto sbagliato e spostala in alto, sopra il suo posto giusto. In questo modo si libera un posto.	
2	Metti sul posto libero la moneta giusta (liberando così un altro posto) e continua in questo modo finché non si libera il posto sotto la moneta spostata in alto.	 
3	Ora metti la moneta che si trova in alto sul suo posto giusto, che è libero. Il turno è così concluso e tutti i posti sono di nuovo occupati.	

Si giocano altri round finché tutte le monete non sono al proprio posto.

Ora vogliamo ricominciare il gioco da capo. Le monete vengono di nuovo disposte a caso sui posti, ma in modo tale che nessuna moneta si trovi al posto giusto.

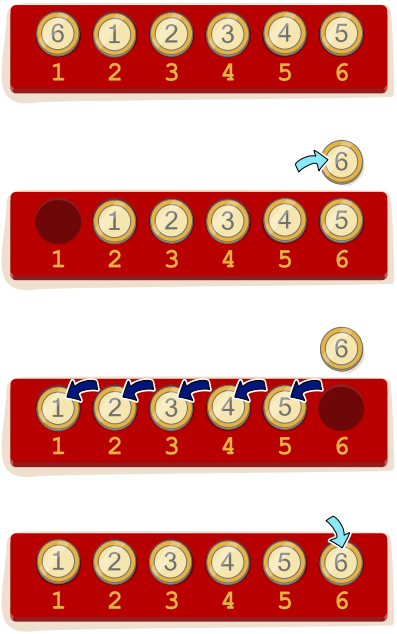
*Quanti turni servono **almeno**, in questo caso, per mettere ogni moneta al posto giusto?*
*E quanti ne servono **al massimo**?*

A) almeno 1 turno, al massimo 3 turni
B) almeno 1 turno, al massimo 6 turni
C) almeno 2 turni, al massimo 3 turni
D) almeno 2 turni, al massimo 6 turni

Soluzione

La risposta corretta è A.

Per prima cosa ci chiediamo quanti turni servano *almeno* per mettere ogni moneta al posto giusto.



Poiché all'inizio nessuna moneta si trova al posto giusto, serve sicuramente almeno un turno. Ma è possibile cavarsela con un solo turno?

A sinistra si vede che è effettivamente possibile: tutte le monete si trovano esattamente un posto a destra rispetto al loro posto giusto, tranne la moneta 6, che si trova sul posto 1, tutto a sinistra.

- Nel passo 1 del turno, la moneta 6 viene spostata in alto, sopra il posto 6.
- Nel passo 2 tutte le altre monete possono essere spostate di un posto verso sinistra, cioè sul loro posto giusto. A quel punto il posto 6 è libero, quindi
- nel passo 3 anche la moneta 6 può essere messa al posto giusto.

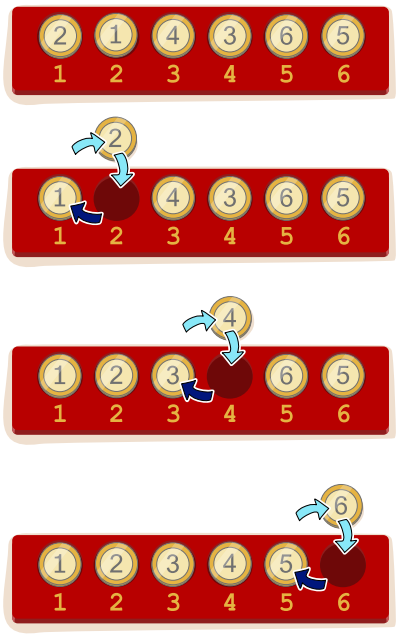
E quanti turni servono *al massimo*?

Nell'esempio qui sopra, risolto in un solo turno, tutte e sei le monete formano un gruppo particolare: ogni moneta si trova su un posto che è quello giusto per un'altra moneta dello stesso gruppo.

Le monete di un gruppo di questo tipo formano un *ciclo*; nell'esempio qui sopra: $6 \to 1$ (significa: la moneta 6 si trova sul posto della moneta 1) $\to 2 \to 3 \to 4 \to 5 \to 6$.

Nell'esempio del testo del compito ci sono due cicli: $2 \to 1 \to 3 \to 2$ e $6 \to 4 \to 5 \to 6$.

 © Castoro Informatico 2026, SSII



In ogni turno del gioco viene risolto esattamente un ciclo, cioè tutte le sue monete vengono messe al posto giusto.

Poiché un ciclo è composto da almeno due monete, sei monete possono formare al massimo tre cicli. In questo caso, il peggiore per il gioco, servono quindi al massimo tre turni per mettere ogni moneta al posto giusto.

A sinistra si vede come la disposizione delle monete con i cicli $2 \to 1 \to 2$, $4 \to 3 \to 4$ e $6 \to 5 \to 6$ viene ordinata in tre turni.

Questa è l'informatica!

Nel gioco delle monete di questo compito, le monete vengono *ordinate*. Ordinare significa disporre dei dati secondo una determinata regola. I programmi per computer ordinano dati molto spesso: per esempio, mettono in ordine numeri in base al loro valore, file in base al nome oppure dati nella memoria di lavoro in base al momento dell'ultimo accesso. Lo fanno perché, di solito, accedere a dati ordinati è molto più veloce che accedere a dati non ordinati. In tutti questi esempi, l'ordinamento si basa su una *relazione d'ordine*, che permette di confrontare gli oggetti tra loro. I programmi possono quindi confrontare i valori di due oggetti per stabilire la loro posizione nella disposizione ordinata. Per questo motivo, questo tipo di ordinamento si dice *basato sul confronto*.

Nel gioco delle monete, invece, è ogni moneta stessa a indicare dove deve andare, cioè sul posto con il suo stesso numero. In questo modo, con il gioco delle monete si possono ordinare monete, o numeri, senza dover confrontare i loro valori. Per questo motivo, questo tipo di ordinamento si dice *non basato sul confronto*.

Poiché nei sistemi informatici si ordinano dati molto spesso, è importante che gli algoritmi di ordinamento consumino meno risorse possibile. Il nostro algoritmo di ordinamento del gioco delle monete, per esempio, per sistemare le monete ha bisogno quasi soltanto dei posti già presenti sulla tavola da gioco. A parte lo spazio in più per la moneta che nel passo 1 viene messa temporaneamente sopra il suo posto giusto, per tutti gli altri spostamenti bastano i posti esistenti. In informatica, gli algoritmi di ordinamento che se la cavano essenzialmente con lo spazio di memoria già necessario per gli oggetti da ordinare si chiamano «algoritmi in loco» (in inglese: *in-place*), perché riordinano gli oggetti «sul posto».

Oltre all'*occupazione di memoria* di un algoritmo, l'informatica si interessa in particolare al suo *tempo di esecuzione*, cioè al numero di passi che l'algoritmo deve compiere in funzione della dimensione dell'input. Il tempo di esecuzione dell'algoritmo del gioco delle monete si può misurare in base al

numero di spostamenti delle monete. Ogni moneta appartiene a un solo ciclo e quindi viene spostata in un solo turno, e una sola volta, sul suo posto giusto. Inoltre, in ogni turno la prima moneta viene spostata anche sopra il suo posto giusto. Con n monete e m turni, il numero di spostamenti è quindi $n + m$. Poiché nel caso peggiore bisogna risolvere cicli di due monete, il numero di turni può essere al massimo $n/2$; si ottiene quindi un numero massimo di $n + n/2 = 3/2n$ spostamenti. In informatica, un tempo di esecuzione di questo tipo, che corrisponde essenzialmente alla dimensione n dell'input, si dice *lineare*. Solo gli algoritmi di ordinamento non basati sul confronto possono avere un tempo di esecuzione lineare; a differenza di quelli basati sul confronto, però, si possono usare solo a determinate condizioni.

Parole chiave e siti web

- Algoritmo di ordinamento: `https://it.wikipedia.org/wiki/Algoritmo_di_ordinamento`
- Algoritmo in loco: `https://it.wikipedia.org/wiki/Algoritmo_in_loco`
- Notazione O-grande: `https://it.wikipedia.org/wiki/O-grande`

 © Castoro Informatico 2026, SSII

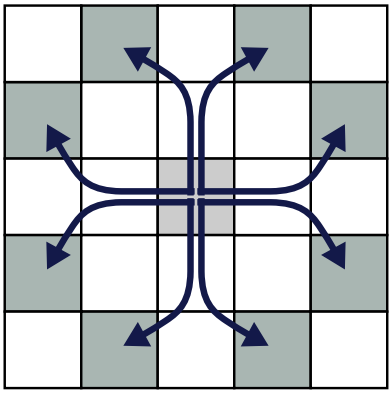
14. Il salto del cavallo



Un cavallo salta su un tabellone di gioco con 25 caselle.
Per visitare una casella, con ogni salto si sposta di:

- 2 caselle in orizzontale e 1 in verticale, oppure
- 2 in verticale e 1 in orizzontale.

L'immagine mostra tutti i salti possibili a partire dalla casella centrale.



Ora il cavallo parte dalla casella 21 e, con una serie di salti, visita esclusivamente le caselle numerate. Ciascuna casella numerata viene visitata esattamente una volta.

1		3		
	7			10
	12	13	14	
16		18		
21				25

In quale ordine il cavallo ha visitato le caselle?

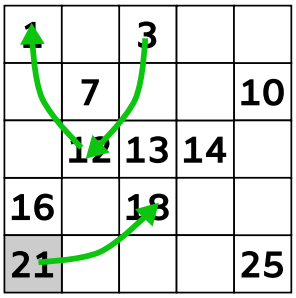
Soluzione

La risposta corretta è la seguente:

21	18	25	14	7	16	13	10	3	12	1

Dalle regole e dalla struttura del tabellone si può ricavare questa sequenza di salti:

- Il cavallo deve visitare la casella 1 per ultima: può saltare sulla casella 1 soltanto dalla casella 12, e da lì potrebbe solo tornare sulla 12, cosa che però non è permessa. Le ultime due caselle sono quindi la 12 e poi la 1. (12→1)
- Sulla casella 12 il cavallo può arrivare dalla 3 o dalla 21. Tuttavia, la casella di partenza 21 non può essere la terzultima della sequenza: deve quindi trattarsi della casella 3. (3→12→1)
- Dalla casella di partenza 21 il cavallo può saltare sulle caselle 12 e 18. Poiché la 12 è già la penultima casella della sequenza, la seconda casella può essere soltanto la 18 (21→18).

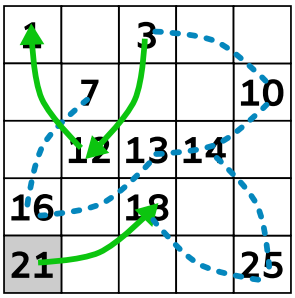


Conosciamo quindi l'inizio (21→18) e la fine (3→12→1) della sequenza di salti (in verde).

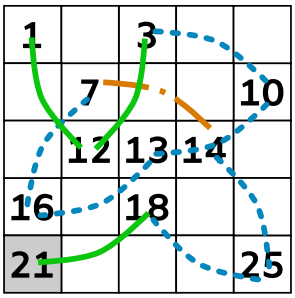
Alcune altre considerazioni ci aiutano a completare il percorso:

- Da ciascuna delle caselle 25, 16 e 10 il cavallo può saltare esattamente su due altre caselle.

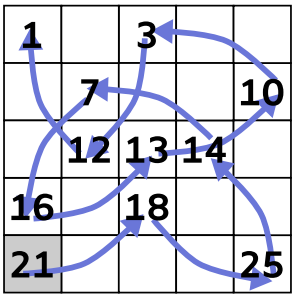
 © Castoro Informatico 2026, SSII



Le sequenze di caselle 7-16−13, 18-25−14 e 13-10−3 devono quindi far parte della sequenza di salti (in blu); resta solo da stabilire in quale direzione vengono percorse.



Solo alle caselle 7 e 14 manca ancora un salto ciascuna. Poiché il cavallo può saltare dall'una all'altra, anche 7−14 fa parte della sequenza di salti (in arancione).

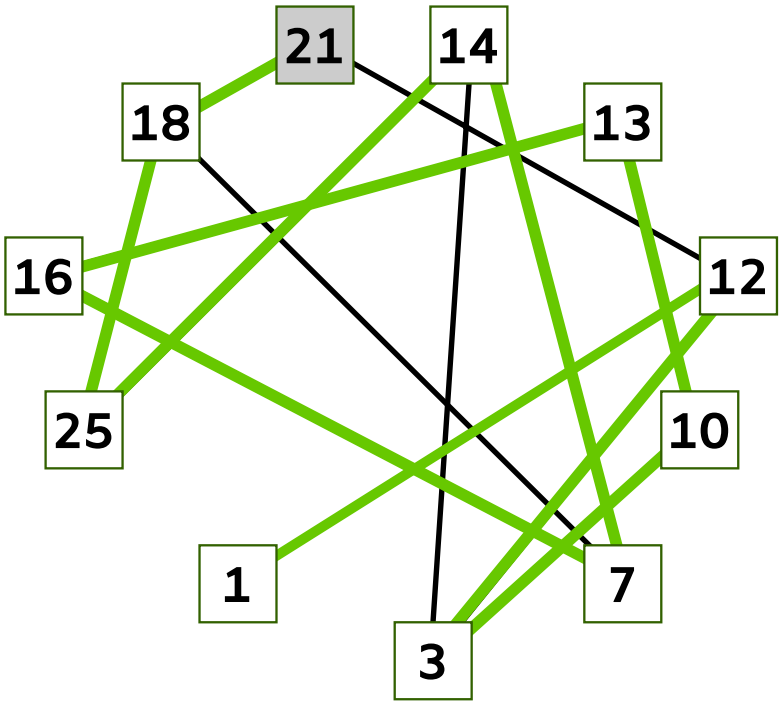


Abbiamo finito: ogni casella numerata viene visitata esattamente una volta. Le direzioni dei salti all'interno dei singoli tratti si ottengono partendo dalla 21 e seguendo le linee.

Questa è l'informatica!

Molti problemi dell'informatica si possono modellare con l'aiuto dei *grafi*. Un grafo è composto da nodi, che rappresentano degli oggetti, e da archi, che descrivono i collegamenti tra questi oggetti. Gli oggetti possono essere di qualsiasi tipo: per esempio, reti stradali, collegamenti aerei o reti di comunicazione possono essere rappresentati come grafi.

Anche in questo compito un grafo è utile: se consideriamo le caselle come nodi, gli archi rappresentano la possibilità di saltare da una casella a un'altra. Ecco il grafo dei salti del cavallo; gli archi che fanno parte della sequenza di salti sono disegnati in verde:



Un'importante classe di problemi sui grafi consiste nel trovare un cammino che soddisfi determinate condizioni. Un esempio noto è la ricerca di un *cammino hamiltoniano:* si tratta di un cammino lungo gli archi del grafo che visita ogni nodo esattamente una volta. La sequenza di salti cercata in questo compito è quindi un cammino hamiltoniano.

I cammini hamiltoniani trovano applicazioni particolarmente interessanti quando gli archi del grafo hanno dei pesi, per esempio quando rappresentano le distanze tra diverse località. Chi cerca il giro più breve che tocchi tutte le località cerca allora, nel grafo delle località e delle distanze, il cammino hamiltoniano (o più precisamente il ciclo hamiltoniano, dato che il giro riporta al punto di partenza) in cui la somma dei pesi degli archi sia la più piccola possibile.

Anche se la domanda «In questo grafo esiste un cammino hamiltoniano?» è semplice da formulare, si tratta di uno dei classici problemi difficili dell'informatica. Per grafi qualsiasi, ancora oggi non si conosce un algoritmo efficiente che trovi sempre un cammino hamiltoniano – o che dimostri che non ne esiste nessuno.

Parole chiave e siti web

- Cammino hamiltoniano: `https://it.wikipedia.org/wiki/Cammino_hamiltoniano`

 © Castoro Informatico 2026, SSII

15. Tacos gratis!

I tacos sono un piatto messicano che si prepara in fretta. Una taquería promette: «Se per un'ordinazione di al massimo 7 tacos impieghiamo più di 15 minuti, l'ordinazione è gratis.»

La taquería ha tre cuochi, che per i diversi tipi di taco impiegano tempi diversi (in minuti):

Tipo di taco	**Ana**	**Beto**	**Carlos**
Pollo	5	3	5
Carne	3	5	5
Vegetariano	2	11	11

Quando arriva una nuova ordinazione, i tacos ordinati vengono assegnati ai cuochi uno dopo l'altro, secondo queste regole:

1. Seleziona un cuoco che finora impieghi al massimo 9 minuti. Se non vi sono cuochi che soddisfano questa condizione, procedi unicamente in base alla regola 2.
2. Scegli il cuoco che prepara il taco più velocemente. In caso di parità, scegli quello che ha bisogno di meno tempo per i tacos che ha già assegnati. Se più cuochi hanno bisogno dello stesso numero di minuti, scegli in ordine alfabetico.

Quando tutti i tacos sono stati assegnati, i cuochi iniziano a lavorare tutti insieme. Se poi un cuoco impiega più di 15 minuti, i tacos sono gratis!

Trova un'ordinazione per la quale la taquería impiega più di 15 minuti.

Utilizza il codice QR per accedere alla versione interattiva di questo esercizio.

Soluzione

Per ottenere un'ordinazione gratis, uno dei tre cuochi deve impiegare più di 15 minuti per i «suoi» tacos. Una strategia per ottenerlo potrebbe essere assegnare a un cuoco che finora lavora al massimo 9 minuti un taco per il quale impiega più di 6 minuti.

Questo è possibile solo con Beto e Carlos, con un taco vegetariano (11 minuti). Per farlo, Ana deve già impiegare più di 9 minuti per i tacos che le sono stati assegnati finora, altrimenti, secondo la regola 2, questo taco verrebbe assegnato a lei.

Impegniamo quindi Ana, ad esempio ordinando 4 volte la carne 🥩 (12 minuti). Con altre 2 porzioni di carne 🥩, Beto e Carlos riceveranno 5 minuti di lavoro ciascuno. A quel punto basterà aggiungere un ulteriore taco vegetariano 🥬: quest'ultimo verrà assegnato a Beto, che totalizzerà così 16 minuti di lavoro, facendo scattare la promozione dei 🌮 gratis!

Ci sono 11 modi diversi di ordinare all'inizio dei tacos che vengono assegnati tutti ad Ana e per i quali lei impiega più di 9 minuti (🍗 non compare, perché per questo tipo di taco Beto e Carlos sono più veloci di Ana):

- 🥩🥩🥩🥩
- 🥩🥩🥩🥬 (in 4 sequenze diverse)
- 🥩🥩🥬🥬 (in 6 sequenze diverse)

Dopodiché dobbiamo portare Beto oppure Carlos ad almeno 16 minuti con un taco vegetariano. Per farlo ci sono 4 possibilità:

- 🥩🥩🥬
- 🥬🥩🥬
- 🥬🍗🥬
- 🥩🥬🥬

Esistono dunque $11 \times 4 = 44$ combinazioni possibili per sovraccaricare la taquería e ottenere i tacos gratis. Su un totale di 2.187 ordinazioni possibili, tuttavia, la probabilità è pari a circa il 2%: non particolarmente elevata.

Questa è l'informatica!

Le regole della taquería in questo compito hanno lo scopo di preparare rapidamente le ordinazioni. Tuttavia, per alcune ordinazioni queste regole non danno un risultato ottimale. Si può allora provare, come farebbe un «attaccante», a sfruttare questa situazione per ottenere tacos gratis. La taquería potrebbe a sua volta modificare le regole per difendersi da simili azioni, ad esempio introducendo un vincolo supplementare: se l'assegnazione di un taco fa superare i 15 minuti di lavoro a un cuoco, si sceglie un cuoco differente. Basterà questo a sventare *tutti* gli attacchi? Purtroppo no! Ti vengono in mente criteri di assegnazione totalmente sicuri?

In informatica, la strategia di cercare per un algoritmo un input il più difficile possibile da elaborare è nota anche come *adversarial attack* (attacco avversario). Si assume il ruolo di un attaccante e si cerca di spingere un difensore a fare qualcosa che in realtà non vorrebbe fare. Gli adversarial attack hanno un ruolo particolare nei sistemi di intelligenza artificiale (IA): con dati di addestramento manipolati di proposito, si possono provocare comportamenti sbagliati nelle reti neurali artificiali.

Nei sistemi informatici, o meglio nei programmi che li controllano, possono quindi esserci regole o algoritmi vulnerabili ad attacchi mirati. Nel caso più innocuo, questo può rallentare il funzionamento o far bloccare il programma. Sarebbe peggio se degli attaccanti sfruttassero una vulnerabilità per accedere a dati senza autorizzazione o addirittura per eseguire programmi a loro scelta. Per questo motivo gli esperti di sicurezza eseguono di proposito, loro stessi, attacchi di questo tipo, per individuare possibili errori e correggerli in tempo. Questo si chiama *adversarial testing*.

Parole chiave e siti web

- Bilanciamento del carico: `https://it.wikipedia.org/wiki/Bilanciamento_del_carico`
- Adversarial attack (attacco avversario): `https://it.wikipedia.org/wiki/Adversarial_machine_learning`
- Adversarial testing: `https://www.security-insider.de/was-ist-adversarial-testing-a-4d8f9ad661ac79548140fe547cfd5152/`
- Collaudo del software / test di sicurezza: `https://it.wikipedia.org/wiki/Collaudo_del_software`, `https://en.wikipedia.org/wiki/Security_testing`



© Castoro Informatico 2026, SSII

16. Nascondino

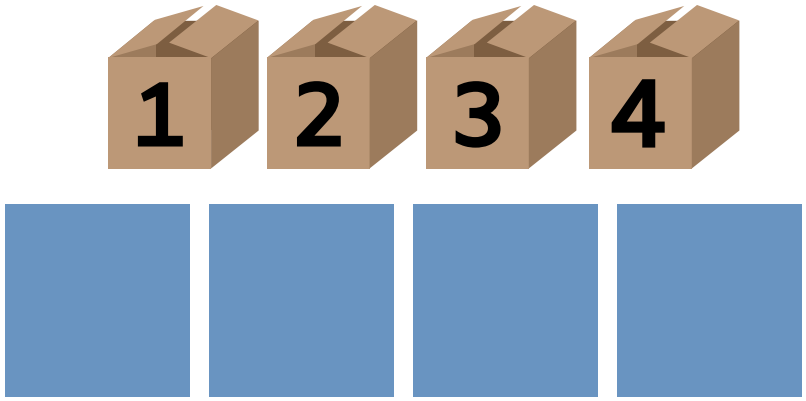
Ben si nasconde in una di quattro scatole numerate, per esempio nella scatola 2.



La sua amica Anja vuole trovarlo e a ogni turno guarda dentro una scatola. Se non lo trova, mentre Anja si sposta per controllare un'altra scatola, Ben si sposta inosservato in una scatola adiacente (a destra o a sinistra).

Anja riflette: se guarda nelle scatole nell'ordine giusto, le bastano al massimo 4 tentativi per trovare sicuramente Ben, in qualunque modo lui si sposti.

Indica un ordine di questo tipo!



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.

Soluzione

Per trovare sicuramente Ben con 4 tentativi, Anja deve guardare nelle scatole in uno di questi due ordini:

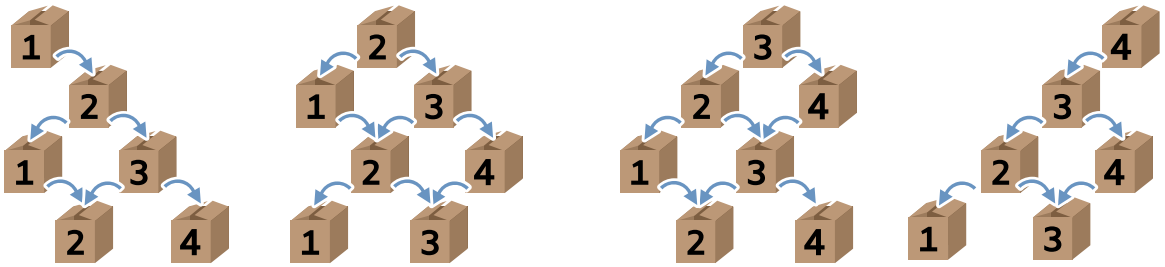


Ora mostriamo che queste risposte sono corrette. Chiamiamo *mossa* ogni spostamento di Ben in questo gioco del nascondino.

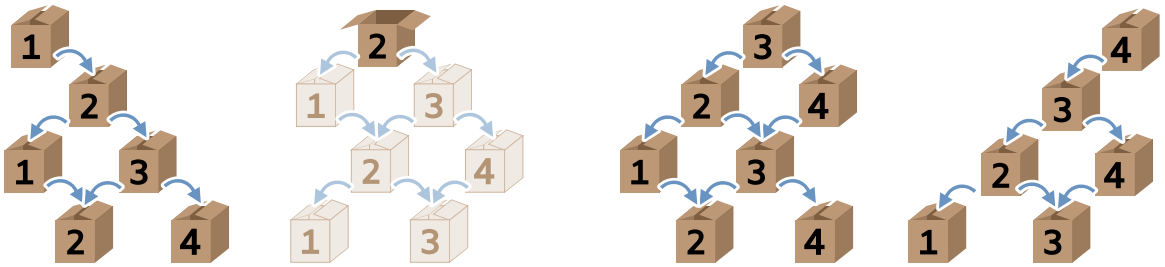
All'inizio Ben può scegliere liberamente la sua «scatola di partenza». Poi, se Ben

- si trova nella scatola 2 o 3, ha due possibilità per la mossa successiva: va a sinistra oppure a destra;
- si trova nella scatola 1 o 4, ha una sola possibilità per la mossa successiva: va a destra (dalla scatola 1) oppure a sinistra (dalla scatola 4).

Ben fa la sua mossa dopo che Anja ha guardato in una scatola senza trovarlo. Se Anja deve trovarlo con (al massimo) 4 tentativi, fino ad allora Ben può fare al massimo tre mosse. Per questo osserviamo tutte le possibili sequenze di tre mosse, partendo da ogni possibile scatola di partenza (1–4):

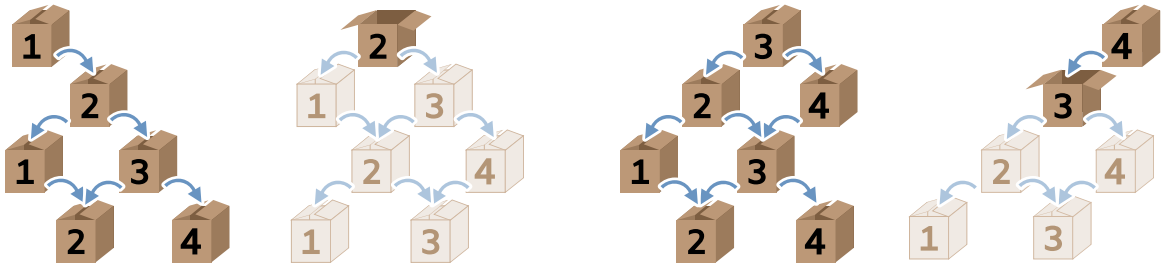


Cosa succede se Anja guarda nelle scatole nell'ordine **2-3-3-2**? Prima Anja guarda nella scatola 2. Se trova Ben lì, la scatola 2 è la sua scatola di partenza, e Ben non può più fare le mosse che partono da lì.

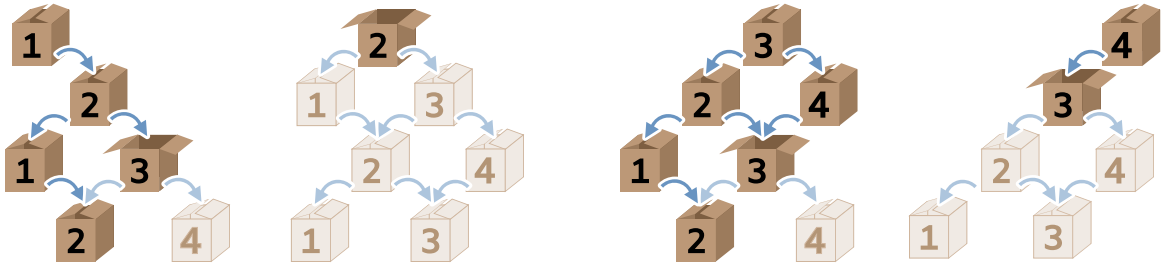


Poi Anja guarda nella scatola 3. Se trova Ben lì, questa è la sua seconda scatola. Al secondo livello delle sequenze di mosse, la scatola 3 compare ancora solo tutta a destra. Ora Ben non può più fare le mosse che partono da lì, e non dobbiamo più considerarle.

 © Castoro Informatico 2026, SSII



Come terzo tentativo, Anja guarda di nuovo nella scatola 3. Questa compare ancora in due punti al terzo livello delle sequenze di mosse. Ora Ben non può più fare le mosse che partono da lì.



Al quarto livello delle sequenze di mosse ora compare solo la scatola 2. Quindi solo la scatola 2 può essere la quarta scatola di Ben, se Anja non lo ha ancora trovato. Se ora Anja guarda nella scatola 2, Ben deve per forza essere lì.

Per la risposta **3–2–2–3** vale lo stesso ragionamento, ma a specchio: infatti le sequenze di mosse che partono dalle scatole 1 e 4 sono speculari tra loro, e lo stesso vale per quelle che partono dalle scatole 2 e 3.

Questa è l'informatica!

In informatica, le possibili sequenze di eventi che partono da uno o più punti di partenza vengono spesso rappresentate come grafi degli stati o come alberi. Si tratta di strutture dati composte da nodi e da archi, ognuno dei quali collega due nodi. Ogni nodo rappresenta un possibile stato. Ogni arco rappresenta un possibile passo da uno stato a quello successivo e per questo ha una direzione.

Queste rappresentazioni si possono usare, tra l'altro, quando ci sono due giocatori con obiettivi opposti, come in questo compito: Ben vuole nascondersi da Anja, Anja vuole trovare Ben. Con l'aiuto dei grafi delle sequenze di mosse, dal punto di vista di Anja abbiamo potuto esaminare in modo sistematico tutti i possibili comportamenti del suo «avversario» Ben e trovare così una strategia che porta Anja sicuramente al successo. Questi metodi vengono usati, tra l'altro, nella *teoria dei giochi*, nella progettazione di algoritmi e nella robotica.

Parole chiave e siti web

- Albero binario: `https://it.wikipedia.org/wiki/Albero_binario`
- Teoria_dei_giochi: `https://it.wikipedia.org/wiki/Teoria_dei_giochi`



© Castoro Informatico 2026, SSII

17. Tinto Azul

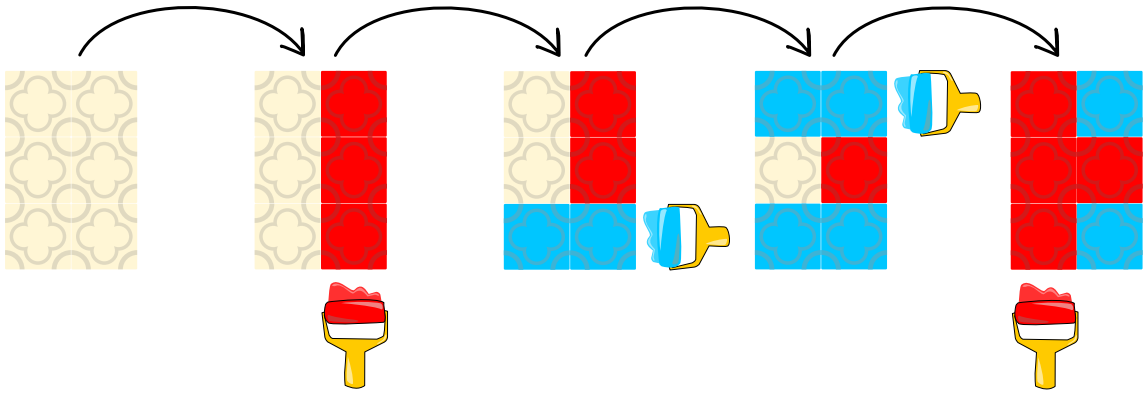
L'imbianchino Tinto Azul è molto specializzato: dipinge soltanto pavimenti composti da piastrelle quadrate .

Per farlo, esegue soltanto queste due azioni:

- dipingere di blu un'intera riga: 
- dipingere di rosso un'intera colonna: 

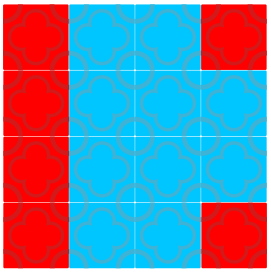
Tinto può anche ridipingere sopra i colori già stesi. Alla fine, sul pavimento si ottiene un motivo blu e rosso.

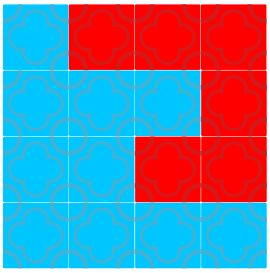
Un esempio: su un pavimento con tre righe e due colonne, Tinto può dipingere il motivo all'estrema destra in questo modo:

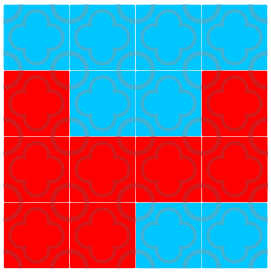


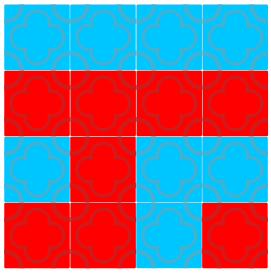
Ora Tinto deve dipingere un pavimento e ha quattro idee. Una di esse è impossibile da creare.

Quale motivo è impossibile da riprodurre?

A) 

B) 

C) 

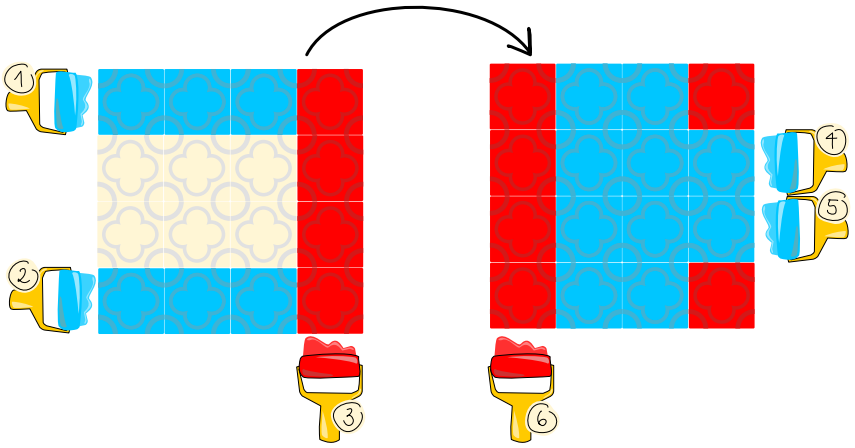
D) 

Soluzione

La risposta corretta è C.

Dapprima spieghiamo in che modo Tinto è in grado di dipingere i motivi nelle risposte A, B e D.

I passaggi necessari per pitturare il motivo nella risposta A sono sei. Tinto procede come segue:

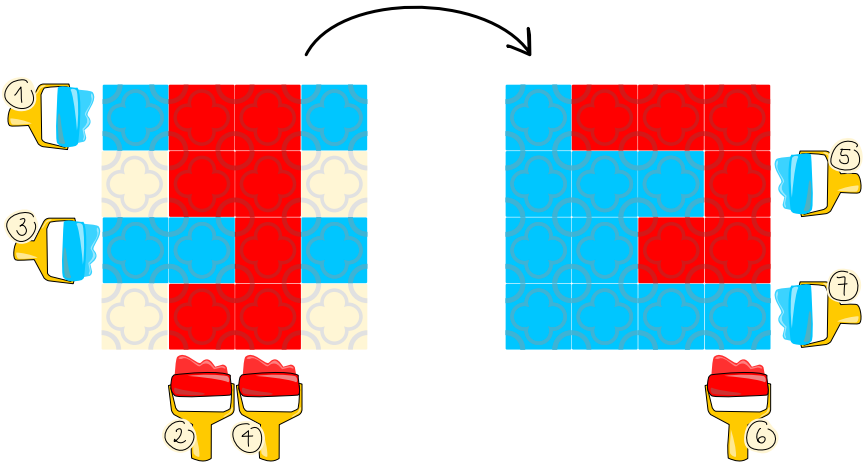


Per ricostruire i singoli passaggi ragioniamo, per così dire, «a ritroso»:

- Nell'ultimo passaggio (6) Tinto ha dipinto di rosso la colonna di sinistra. Questa colonna è l'unica completamente rossa. Nessuna delle sue piastrella verrà ricoperta, in seguito, di blu.
- Nei due passaggi precedenti (4, 5) Tinto ha dipinto di blu le due righe centrali. Più in là (nel passaggio 6), la prima piastrella di ciascuna riga verrà ricoperta di rosso.
- Nel passaggio precedente (3), Tinto ha colorato di rosso la colonna di destra. Sappiamo infatti che Tinto in seguito (passaggi 4 e 5) ricoprirà di blu le sue due piastrelle centrali, ma non le rispettive piastrelle in alto e in basso.
- Per la medesima ragione, Tinto ha colorato di blu, sull'arco di due passaggi (1 e 2), la riga superiore e quella inferiore.

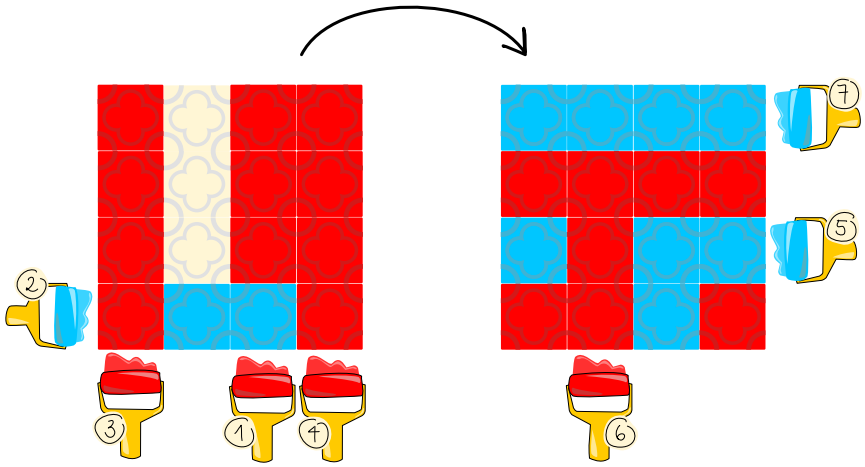
Ragionamenti simili ci permettono di capire come Tinto abbia dipinto i motivi nelle risposte B e D:

Pavimento B:



 © Castoro Informatico 2026, SSII

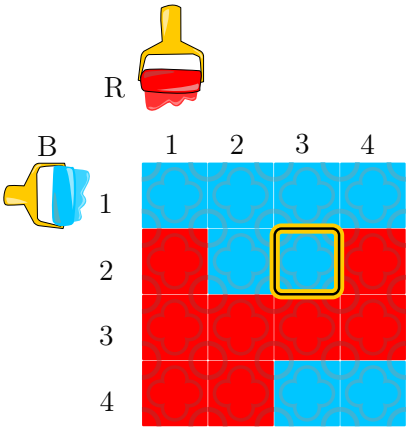
Pavimento D:



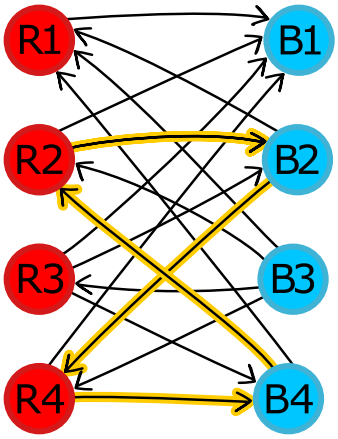
Per quale ragione Tinto non è in grado di dipingere il motivo della risposta C? Anche in questo caso potremmo procedere a ritroso. L'argomentazione è tuttavia complicata. Cambiamo pertanto l'approccio e facciamo leva sul fatto che il pavimento è composto da singoli quadrati disposti in quattro righe e quattro colonne. Assegniamo ad ogni riga e ad ogni colonna un numero da 1 a 4 e descriviamo le due azioni come segue:

- Tinto dipinge una riga di blu : B1 – B4
- Tinto dipinge una colonna di rosso : R1 – R4

Il quadrato (x,y) nella riga x e nella colonna y viene pitturata con le azioni Bx o Ry.



Il colore attuale del quadrato (x,y) ci consente di determinare quale delle due azioni sia stata eseguita per ultima: se il quadrato è blu, l'azione Ry viene eseguita, se del caso, prima di Bx, altrimenti vale il contrario.



Determiniamo quest'ordine per tutti i quadrati e otteniamo il diagramma soprastante. Una freccia da una prima a una seconda azione indica che la prima azione è stata eseguita prima della seconda. Per il quadrato blu (2,3) c'è quindi una freccia che va da R3 a B2.

Le frecce evidenziate in giallo ci permettono di scoprire una sottile contraddizione: R2 viene eseguita prima di B2, B2 viene eseguita prima di R4, R4 viene eseguita prima di B4, e, da ultimo, B4 dovrebbe venir prima di R2. B4 non può venire eseguita prima di R2 e, nel contempo, dopo R2. Proprio per questo motivo Tinto non sarà mai in grado di dipingere il motivo della risposta C.

Questa è l'informatica!

Ogni motivo colorato ci fornisce un indizio su quale delle sue due azioni Tinto abbia eseguito per ultima. Le due azioni sono la base di ciò che le informatiche e gli informatici chiamano un *procedimento deterministico*: l'esecuzione di ogni singola operazione porta da uno *stato* (cioè da un motivo dipinto sul pavimento) ad un altro stato in modo unico e prestabilito. Lo *stato finale* è determinato completamente dallo stato iniziale (in questo task partiamo da un pavimento privo di colore) e dalla sequenza delle operazioni.

Se osserviamo un motivo e ci chiediamo se sia possibile o meno dipingerlo con le azioni a nostra disposizione, cerchiamo di ricostruire, partendo dallo stato finale, la sequenza delle azioni che ha portato, passo dopo passo, a tale stato. Questo tipo di ricostruzione a ritroso può aiutarci a scoprire errori di programmazione: durante il processo di *debugging* verifichiamo in modo molto simile se, e come, un output errato di un programma possa essere stato prodotto dalla sequenza delle operazioni che abbiamo programmato.

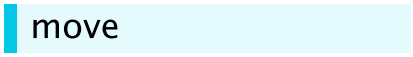
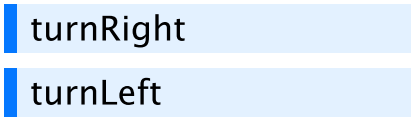
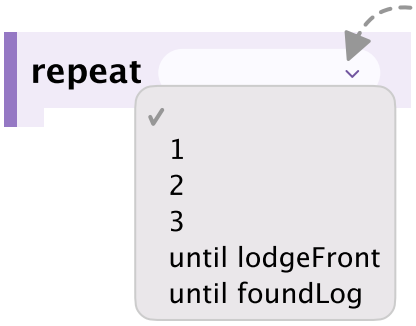
Parole chiave e siti web

- Debugging: `https://it.wikipedia.org/wiki/Debugging`

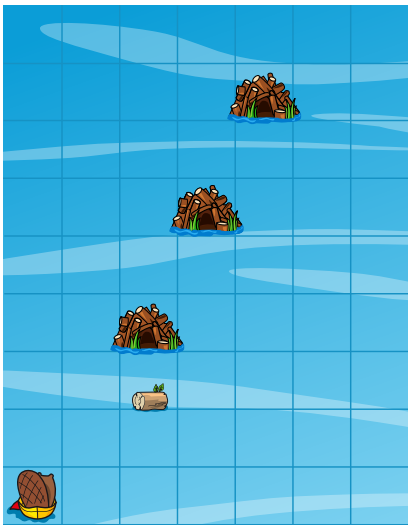
18. Un aiuto dai vicini

La castora Belinda ha urgente bisogno di tronchi. Ogni giorno un vicino diverso lascia un tronco davanti alla propria tana. Belinda vuole raccogliere il tronco ogni giorno con le stesse istruzioni di percorso. Dopo aver raccolto il tronco, non vuole più passare dai vicini successivi. Riesci ad aiutare Belinda a trovare queste istruzioni di percorso?

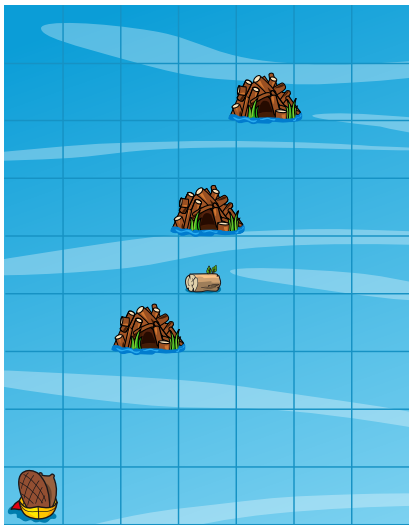
Puoi usare le seguenti istruzioni:

Istruzione	Spiegazione
	Belinda si sposta in avanti di una sola casella, nella direzione in cui guarda. Se Belinda arriva su una casella con un tronco, lo raccoglie automaticamente.
	Belinda ruota sul posto verso destra / sinistra.
	Se trascini delle istruzioni dentro il blocco, queste vengono ripetute (`repeat` = ripeti).
	Clicca sul campo chiaro e scegli quante volte ripetere le istruzioni. Con `until` puoi scegliere tra: • `lodgeFront` (finché Belinda non si trova direttamente davanti a una tana di castori), • `foundLog` (finché Belinda non ha raccolto il tronco). In tal caso, il cicolo termina dopo l'ultima istruzione contenute nel blocco `repeat`. Se non scegli niente, le istruzioni vengono ripetute *all'infinito*.

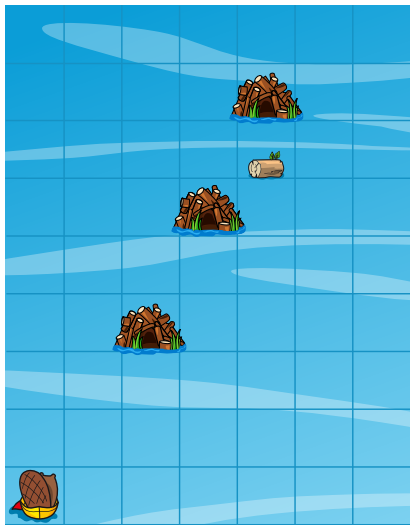
Scrivi delle istruzioni di percorso con cui Belinda raccoglie il tronco in tutti e tre i giorni. Appena ha raccolto il tronco, Belinda non deve più proseguire verso i vicini successivi.



Giorno 1



Giorno 2



Giorno 3

 © Castoro Informatico 2026, SSII

Soluzione

Una soluzione corretta è la seguente:

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

Per arrivare al tronco in tutti i giorni, bisogna prima di tutto capire che il percorso può essere descritto con uno schema che si ripete:

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

Per arrivare il terzo giorno fino al vicino con il tronco, si scrivono le istruzioni dello schema tre volte di seguito. In questo modo Belinda raccoglie il tronco anche il primo e il secondo giorno. Le istruzioni di percorso sarebbero quindi 27 in tutto. Per accorciarle, si possono anche far ripetere le istruzioni dello schema. Per farlo, si mettono in un blocco `repeat` e si scelgono `3` ripetizioni:

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

Così le istruzioni di percorso non sono più 27, ma solo dieci.

In realtà abbiamo già trovato delle istruzioni di percorso per Belinda. Però la soluzione con `repeat` e un numero fisso di ripetizioni non è del tutto soddisfacente. Infatti, se il villaggio dei castori non avesse tre tane, ma per esempio cinque, dovremmo cambiare ogni volta il numero di ripetizioni nelle istruzioni di percorso.

Risolviamo il problema tenendo conto della seconda richiesta di Belinda: dopo aver raccolto un tronco, non vuole più passare dai vicini successivi. Ma è proprio quello che succederebbe se usassimo sempre `repeat` con un numero fisso di ripetizioni. Il primo giorno Belinda andrebbe comunque dal secondo e dal terzo vicino, anche se ha già raccolto il tronco dal primo vicino. Il secondo giorno succederebbe qualcosa di simile.

Per questo motivo usiamo la condizione until `foundLog` nel blocco `repeat` anziché un numero specifico:

```
repeat until foundLog
  turnRight()
  move()
  move()
  turnLeft()
  move()
  move() <-- qui Belinda raccoglie il tronco
  turnLeft()
  move()
  turnRight() <- qui raggiunge la sua nuova posizione di partenza
```

Infatti ora le istruzioni dello schema vengono ripetute solo fino a quando Belinda non ha raccolto il tronco. Prima che le istruzioni nel blocco `repeat` vengano eseguite di nuovo, la condizione `until foundLog` verifica se è già stato trovato un tronco. Se è così, le istruzioni non vengono ripetute e Belinda non prosegue verso i vicini successivi.

Attenzione: sebbene Belinda raccolga il tronco già dopo il quarto comando `move` (e quindi la condizione `foundLog` sia tecnicamente soddisfatta), i tre comandi successivi vengono comunque eseguiti tutti. Infatti, il controllo (ripetuto) tramite la condizione `foundLog` avviene sempre e solo dopo che tutti i comandi all'interno del blocco `repeat` sono stati eseguiti completamente. Perciò Belinda non si ferma subito nel punto in cui raccoglie il tronco, ma prosegue fino a raggiungere la sua successiva posizione di partenza (senza però continuare verso il vicino seguente).

```
repeat until foundLog
  turnRight()
  move()
  move()
  turnLeft()
  move()
  move() <-- qui Belinda raccoglie il tronco
  turnLeft()
  move()
  turnRight() <-- qui raggiunge la sua nuova posizione di partenza
```

Approfondimento: Abbiamo descritto il percorso con uno schema. Nel blocco `repeat` lo schema è diviso in due parti:

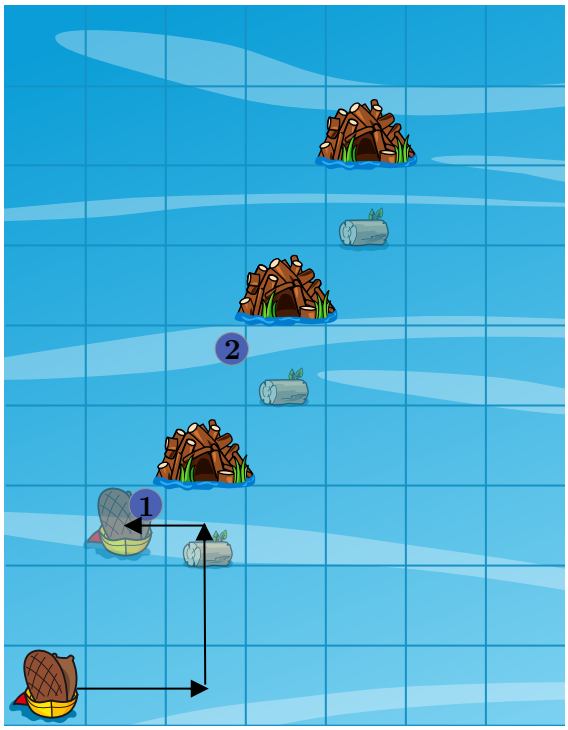
Parte 1: percorso fino al tronco

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

Parte 2: percorso fino alla nuova posizione di partenza

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

Se dividi il tuo schema in queste due parti, Belinda può seguire un percorso qualsiasi fino al tronco **e** un percorso qualsiasi fino alla posizione finale (posizione 1 per il primo tronco, posizione 2 per il secondo tronco). Belinda raccoglie comunque sempre tutti i tronchi e non prosegue mai fino al vicino successivo.



Questa è l'informatica!

Un'*invariante* (in questo caso: *invariante di ciclo*) è una proprietà che vale prima e dopo ogni ripetizione di un ciclo. Aiuta a descrivere lo stato di un sistema e a verificare che un procedimento sia

corretto. Le invarianti si usano molto spesso nei cicli, per mostrare che certe condizioni restano vere a ogni ripetizione del ciclo. Nel nostro esempio dobbiamo orientare la nostra castora in modo che la posizione di partenza (prima della ripetizione del ciclo) e la posizione finale (dopo la ripetizione del ciclo) siano identiche. La posizione di partenza/finale è definita dalla direzione in cui guarda la castora e dal punto in cui si trova rispetto alla tana di castori successiva.

Parole chiave e siti web

- Invariante di ciclo: `https://en.wikipedia.org/wiki/Loop_invariant`
- Ripetizione (ciclo): `https://it.wikipedia.org/wiki/Ciclo_(informatica)`

A. Autori dei quesiti

Esraa Almajhad

Sakuya Araki

Masiar Babazadeh

Wilfried Baumann

Hans-Joachim Böckenhauer

Graeme Buckie

Leonardo Cavalcante

Cesar Cepeda

Maria Cepeda

Špela Cerar

Kris Coolsaet

Valentina Dagienė

Christian Datzko

Heather Duan

Oscar Durango

Susanne Falkenberg-Thut

Gerald Futschek

Christian Giang

Arnheiður Guðmundsdóttir

Jens Hartmann

Heikki Hyyro

Filiz Kalelioğlu

Asterios Karagiannis

David Khachatryan

Geetanjali Khatri

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

Maiko Shimabuku

Itsuki Shimada

Bronius Skūpas

Emma Sotomayor

Jacqueline Staub

Alieke Stijf

Monika Tomcsanyiova

Marcel Vlastuin

Michael Weigend

Chris Wetherell

Kyra Willekes

Alexander Wiss

© Castoro Informatico 2026, SSII

B. Partner accademici

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, Cantone di Berna
https://www.bkd.be.ch/de/start/ueber-uns/die-organisation/amt-fuer-kindergarten-volksschule-und-beratung.html



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



Fondazione Informatica Svizzera
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

© Castoro Informatico 2026, SSII