

Lorenzo Repetto

# Dai giochi agli algoritmi



**Un'introduzione non convenzionale  
all'informatica**

**Edizioni Kangourou Italia**

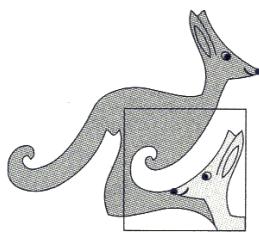


Lorenzo Repetto

# Dai giochi agli algoritmi

*Un'introduzione non convenzionale  
all'informatica*

*Seconda edizione, riveduta e ampliata*



© *Edizioni Kangourou Italia 2019*  
**ISBN 978-88-89249-62-8**

Qualsiasi traduzione, adattamento o riproduzione, anche parziale, con qualsiasi mezzo, in qualsiasi nazione, senza preventiva autorizzazione, è illecita ed espone i contravvenitori a procedimenti giudiziari.

*A tutti i miei allievi*

# Sommario

Prefazione alla seconda edizione	1
Prefazione alla prima edizione	7
<b>Parte prima. Con carta, matita, gomma... e computer</b>	<b>9</b>
<b>1. Un inizio “problematico”</b>	<b>11</b>
Il percorso più breve	11
Un percorso a stadi successivi	18
Una vacanza in Grecia	20
<i>Compilation</i>	22
La ruota della fortuna	25
Successioni “autosomiglianti”	27
I percorsi dell’ape	31
Indovina il numero pensato!	32
Pile di carte	34
<b>2. Omaggio a Smullyan</b>	<b>35</b>
Un’altra torta rubata	35
A proposito di nani sinceri e nani bugiardi...	36
Sei proposizioni	38
Ed ecco a voi Ray Smullyan!	39
Una curiosa macchina numerica	41
Macchine che parlano di sé stesse	45
L’isola G	48
Un indovinello “doppiamente gödeliano”	49
Due macchine che parlano di sé stesse e ciascuna dell’altra	50
I programmi dei professori Roberts	50
I contributi di Gödel alla logica moderna e ciò che ne consegue	54
Dedicato agli scacchisti	57
<b>3. Piaceri e limiti del calcolo</b>	<b>61</b>
Come esprimere un procedimento di calcolo... purché lo si possa fare!	62
Problemi indecidibili	68
Algoritmi del passato... ma sempre attuali	73
Grammatiche, linguaggi e automi	78
Grammatiche libere dal contesto	81
Una grammatica per le espressioni aritmetiche col suo <i>parser</i>	83
Le classi delle grammatiche meno restrittive	88
Alcuni quesiti	92

<b>4. Problemi intrattabili o quasi</b>	<b>95</b>
Problemi esponenziali e super-esponenziali	95
Le classi P e NP	100
Cicli hamiltoniani e torri di Hanoi	107
Problemi di ottimizzazione combinatoria: il commesso viaggiatore	109
Un algoritmo di <i>branch-and-bound</i>	114
Un algoritmo di programmazione dinamica	119
Il TSP METRICO e gli alberi ricoprenti di costo minimo	121
Il problema dell'imballaggio	124
Il problema dello zaino	127
Problemi NP-ardui	131
Primalità	137
Fattorizzazione	142
L'algoritmo di crittografia RSA	145
<b>5. Tappezzerie, decorazioni e altre cose ancora</b>	<b>151</b>
Tappezzerie per la mente	151
Decorazioni per uova Fabergé	154
Immagini ottenute con formule ricorrenti	155
Tappeti persiani “ricorsivi”	156
Montagne “frattali”	159
Sistemi dinamici “caotici”	162
Frattali	165
<b>Parte seconda. Da zero a due giocatori: divertimenti con il computer</b>	<b>171</b>
<b>6. Automi cellulari e macchine di Turing</b>	<b>173</b>
Che cos’è un automa cellulare?	173
Un automa di Ulam	178
Neve aliena	179
A che cosa può servire un automa cellulare?	181
Macchine di Turing bidimensionali	184
<b>7. La corsa del cavallo sulla scacchiera</b>	<b>191</b>
In breve, una lunga storia	191
Come si può trovare una soluzione?	196
Quante sono le soluzioni?	200
Giri semimagici	203
Un algoritmo di esaurimento	205
Campi e metodi per il giro del cavallo	208
Ricerca esauriente di tutte le soluzioni	210
Il problema delle $n$ regine	211
Il problema dei sei cavalli	212
I problemi del labirinto	213

<b>8. La soluzione sta nella traccia!</b>	<b>215</b>
Rane e rospi	215
Un solitario con le pedine proposto da Dudeney	216
Che cosa cambia nell'algoritmo di esaurimento?	217
Un solitario con le pedine proposto da Tait	220
Troviamo le soluzioni più brevi!	223
Le sei rane e altri giochi di cambio di posizione	229
Il gioco del quindici	232
Ricerca euristica di una delle soluzioni più brevi	234
Versioni grandi e piccole	235
A prua e a poppa, ovvero rane e rospi bidimensionali	237
<i>Appendice:</i> una definizione completa della classe parametrica STACK<ITEM>	240
<b>9. Una partita a tris</b>	<b>245</b>
Le prime nozioni: stati e albero di gioco	246
L'analisi, in generale	249
Idee per la realizzazione di un programma	251
Rendiamo migliore il metodo!	253
Giochi strettamente determinati	255
Un gioco isomorfo al filetto	257
Due varianti del filetto	258
<b>10. Awari... ma con un occhio agli scacchi!</b>	<b>259</b>
Un po' di storia...	263
L'Awari è stato risolto!	264
Come realizzare un programma che giochi "piuttosto bene"	266
Ricordiamo le varianti principali!	270
Potature dell'albero di gioco	272
A proposito di scacchi...	278
Chi vuol cimentarsi nella progettazione di un gioco?	280
<b>11. Uno dei due vince sempre!</b>	<b>283</b>
Babylone	283
Babylone-one	286
Il gioco di Grundy	289
Il gioco di Euclide	290
MIN	292
I giochi del (non) ritorno e della (non) ripetizione	292
Chi arriva a un euro?	296
NIM	297
... Facciamo trentuno!	300

<b>Appendice</b>	<b>301</b>
1. Il Sudoku, un po' più in dettaglio	303
2. L'esatta copertura di un insieme	314
3. Polimini, in particolare pentamini	317
4. Altri puzzle con i polimini	325
5. Giochi per due con i pentamini	329
6. Quadratini colorati	333
7. La linea più ampia e la minima copertura di un insieme	336
8. L'insieme dominante, la copertura per nodi e l'insieme indipendente	341
9. Giocando con le biglie	349
10. Tchouka e Tchoukaillon	359
11. Il solitario bulgaro	363
12. Ancora qualcosa sui giochi di strategia	367
13. Chi andrà a giocare in cortile? Ovvero, la programmazione logica	376
13.1. Il problema della negazione	382
13.2. Programmi logici generali e <i>database</i> deduttivi	390
14. Il <i>database</i> dell'agenzia di viaggi	397
15. L'albero di Natale e gli alberi rosso-neri	404
16. Fuochi d'artificio, bandiere e filastrocche	410
16.1. Codifica <i>run-length</i>	415
16.2. Codifica a lunghezza variabile	416
16.3. Algoritmo di Lempel-Ziv (1977)	419
16.4. Algoritmo di Lempel-Ziv-Welch (1984)	420
17. Torri di Hanoi e ribaltamento delle frittelle	424
<b>Indice dei nomi</b>	<b>429</b>
<b>Galleria di immagini</b>	<b>435</b>

## Prefazione alla seconda edizione

Nella primavera del 2011 accolsi, con grande entusiasmo, la proposta dell'amico editore Angelo Lissoni, presidente dell'Associazione Culturale Kangourou Italia, di scrivere un libro utile – e anche un po' divertente – soprattutto per gli studenti che avrebbero partecipato a gare di carattere informatico. Attingendo in parte al materiale che – nel corso di tanti anni d'insegnamento – avevo già approntato, e adattando o ideando *ex novo* rompicapi di vario genere, mi accinsi alacremente all'opera, e in aprile del 2012 fu pubblicato *Dai giochi agli algoritmi. Un'introduzione non convenzionale all'informatica*. In effetti, come recitava il sottotitolo, mio intento precipuo era avviare all'apprendimento di nozioni e idee certamente non banali, prendendo ispirazione, per quanto possibile, da questioni ludiche. Tuttavia, ero consapevole del fatto che la mia esposizione – non propriamente divulgativa – era, in talune parti, poco accessibile al lettore che non avesse già discrete conoscenze in ambito logico-matematico o fosse all'oscuro di alcuni concetti fondamentali della scienza informatica.

In questa nuova edizione *on-line* ho così aggiunto una corposa “Appendice”, contenente giochi, alcuni già noti al grande pubblico, e quesiti di forse più facile comprensione (e magari soluzione), arricchendo in tal modo la parte maggiormente accessibile al lettore, corredata di adeguati commenti, pur senza rinunciare a un inquadramento dei problemi proposti nel loro giusto contesto scientifico. Tuttavia, non ho rinunciato, nemmeno in quella sede, a diverse digressioni – un tantino più impegnative, lo ammetto – che però il lettore può tralasciare in tutta tranquillità: mi riferisco ai problemi di copertura (esatta o minima) di un insieme, agli algoritmi sui grafi, alla programmazione logica, alle procedure di compressione dei dati.

Ma questo è soltanto uno dei diversi motivi che mi hanno indotto a preparare una seconda edizione: sebbene il testo sia per lo più improntato a fatti certi e basilari, e a risultati ormai da tempo consolidati (salvo improvvisi sovertimenti, che di quando in quando accadono persino in una scienza esatta), dopo ben sette anni è comunque opportuno un aggiornamento, accompagnato da una – spero accurata – revisione.

Dall'anno scolastico 2015/2016, il *Kangourou dell'Informatica* ha lasciato il posto alle gare *Bebras dell'Informatica*. “Bebras” in lituano significa “castoro”: la prima competizione, proposta da Valentina Dagienė dell'Università di Vilnius, si svolse infatti in Lituania nel 2004; oggi la comunità Bebras è internazionale, anzi intercontinentale, poiché accorpa insegnanti di informatica da paesi di tutti i continenti, i quali annualmente preparano, revisionano e infine scelgono piccoli e divertenti quesiti o giochi, sempre nuovi, da sottoporre – durante la seconda settimana di novembre – a milioni di studenti, dalle scuole primarie alle scuole superiori, in più di cinquanta nazioni in tutto il mondo, Italia compresa.

Si tratta di una sfida: non vi sono premi per coloro che riescono meglio nel compito, ma il premio assicurato per tutti i partecipanti alla “gara del castoro” consiste nell’affinamento delle proprie abilità e nello sviluppo del cosiddetto *pensiero computazionale* o *algoritmico*, che poi sta alla base delle competenze in informatica, ovvero in *computer science*. Nel nostro paese continua comunque la propria attività Kangourou Italia, membro di un’altra comunità intercontinentale, che propone competizioni a vari livelli concernenti diverse discipline, *in primis* la matematica, che della logica e dell’informatica è parente stretta.

Alla pagina web <https://bebras.it/materiali.html> si trovano i copiosi e interessanti materiali riguardanti le ormai lontane edizioni delle gare *Kangourou dell’Informatica* (il sito <https://bebras.it/> è curato dai docenti di ALaDDIn, Laboratorio di Divulgazione e Didattica dell’Informatica, afferenti al Dipartimento di Informatica dell’Università degli Studi di Milano), e volendo anche all’indirizzo <http://kangourou.di.unimi.it/> (o attraverso il sito ufficiale di Kangourou Italia, <http://www.kangourou.it/>) si può accedere all’archivio completo delle sette edizioni del *Kangourou dell’Informatica*, a cui nel presente libro si fa spesso riferimento: si trattava di quesiti talvolta più difficili di quelli che attualmente compaiono nelle gare *Bebras*, ma per certi aspetti forse più stimolanti...

Il grande logico, nonché simpatico e raffinato divulgatore Raymond Smullyan, al quale è dedicato il secondo capitolo di questo libro, ci ha lasciati a febbraio del 2017, nel novantottesimo anno della sua vita: le pagine che parlano di lui e delle sue opere siano anche un modesto ma sentito tributo alla sua memoria!

Quattro “nuovi” numeri primi di Mersenne sono stati trovati – l’ultimo, annunciato il 7 dicembre 2018 e confermato dopo ben due settimane di calcoli nonostante la velocità dei *test* per i numeri di questa forma, ha quasi 25 milioni di cifre decimali! – sicché quelli oggi conosciuti sono 51, ed è stato verificato che, al di sotto del quarantasettesimo noto, non ve ne sono altri da scoprire.

Alcuni nuovi conteggi sono stati compiuti, nel frattempo, su classici *puzzle*, quali le corse del cavallo sulla scacchiera e il gioco del 24 (estensione del gioco del 15), e algoritmi un poco più efficienti dei precedenti conosciuti, inerenti a vari problemi di ottimizzazione combinatoria, sono stati ideati nell’ultimo lustro.

Nel decimo capitolo sono delineati i principî e le tecniche su cui si basano i software “tradizionali” in grado di giocare (ad Awari, a scacchi) contro un avversario, umano o anch’esso artificiale. Un ruolo chiave, e assai delicato, è rivestito dalle funzioni di valutazione delle posizioni e dalle regole euristiche per l’ordinamento delle mosse, pensate e messe a punto, di solito “artigianalmente”, da esperti (umani) del gioco in questione. Raffinati metodi di ricerca, funzioni di valutazione accurate e complesse, “trucchi” specifici, sostenuti da algoritmi e strutture di dati per aumentare l’efficienza nonché da immense “librerie” di aperture, finali e intere partite, hanno permesso ai programmi di superare largamente i migliori giocatori umani di scacchi e di Shōgi...

Degli scacchi diremo qualcosa, fra i capitoli nono e decimo. Attualmente, i più forti programmi tradizionali sono Komodo 11.2, Houdini 6 e, sopra tutti, Stockfish 10: questi tre superano i 3400 punti secondo il sistema Elo, quando l'attuale campione del mondo, il norvegese Magnus Carlsen, non arriva a 2900. Stockfish, sviluppato originariamente dal norvegese Tord Romstad e dall'italiano Marco Costalba, in linguaggio C++, è un software multipiattaforma, che ha il grande pregio di essere *open-source* (<https://stockfishchess.org/>); gli altri due sono commerciali, ma vengono forniti a prezzi piuttosto modesti, con un'ampia documentazione, e comunque si possono installare ed eseguire su un normale personal computer.

Quanto allo Shōgi, gli scacchi giapponesi, Akara fu il primo software a vincere un campione umano: sviluppato da ricercatori di due università di Tokyo, e dotato di quattro “motori” e una rete di 169 computer, nel 2010 sconfisse la campionessa Ichiyo Shimizu. Dall’anno successivo, Ponanza iniziò a mietere successi, giungendo a battere nel 2017 il grande maestro Amahiko Satō. Sempre nel 2017, il software “elmo” (una funzione di valutazione, più una libreria di aperture) combinato con “yaneura ou” (un efficiente motore di ricerca con “potature”) vinse il campionato del mondo fra giocatori artificiali. Ormai questi ultimi, nei giochi classici come gli scacchi e lo Shōgi, hanno raggiunto livelli così alti da confrontarsi esclusivamente fra loro, non più con gli umani! In effetti, la capacità dei computer di sconfiggere l'uomo giocando costituisce da sempre un significativo punto di riferimento per i progressi nel campo dell'intelligenza artificiale.

Solo qui, in sede di prefazione, mi limito ad accennare ai più recenti e radicalmente diversi approcci alla progettazione e alla realizzazione di giocatori artificiali, poiché tecnologie talmente avanzate esulano dagli scopi del presente volume.

Fra il 2015 e il 2016 è accaduto, per la prima volta nella storia, che alcuni tra i migliori maestri di Go – gioco ancor più complesso dello Shōgi, quanto a numero di possibili configurazioni sul tavoliere – siano stati sconfitti da un software, *AlphaGo*, sviluppato con una metodologia innovativa da DeepMind, una società britannica con sede a Londra, guidata dal neuroscienziato Demis Hassabis e acquistata da Google nel 2014 (si veda: David Silver, Aja Huang e altri, *Mastering the game of Go with deep neural networks and tree search*, Nature, Vol. 529, No. 7587, 28 gennaio 2016, pp. 484-489).

L'algoritmo di ricerca combina due reti neurali (una per la scelta delle mosse, l'altra per la valutazione delle posizioni) con il *Monte Carlo Tree Search*, un metodo probabilistico basato su campionamenti statistici: per l'autoapprendimento, esso simula rapidamente una serie di intere partite, scegliendo casualmente ad ogni passo tra le mosse ritenute più promettenti, allo scopo di stimare quante volte una mossa da valutare potrebbe condurre alla vittoria. Così facendo, i vantaggi si apprezzano quando lo spazio di ricerca è assai ampio, come accade nel Go più che negli scacchi, e non si dispone di una buona funzione di valutazione euristica.

AlphaGo ha iniziato a “imparare” il gioco da un vasto catalogo di circa 30 milioni di esempi di posizioni e relative mosse, compilato da esperti umani. Indi s’è addestrato, giocando contro sé stesso milioni di volte, istruendo ulteriormente dapprima la rete di scelta delle mosse e poi, sulla base dei risultati ottenuti, la rete di valutazione delle configurazioni, con le relative mosse, mediando col succitato metodo probabilistico. Memorabile è stata la sconfitta del leggendario campione sudcoreano Lee Sedol, a marzo 2016, in un *match* di cinque partite, quattro delle quali vinte da AlphaGo; a maggio 2017, perse tre partite su tre il “numero uno” mondiale, il cinese Kē Jié.

Il passo successivo compiuto da DeepMind è stato *AlphaGo Zero*, un software in grado di migliorare viepiù sensibilmente le proprie capacità di gioco, ma pur sempre dedicato esclusivamente al Go. Esso combina le due reti – quella per scegliere le mosse e quella per prevedere l’evoluzione di uno stato del gioco – in una sola (ma con molti più livelli di neuroni artificiali “sintonizzabili”), che man mano si migliora vicendevolmente con l’algoritmo di ricerca ad albero, per prevedere l’andamento della partita. AlphaGo Zero trae un limitato vantaggio dalle conoscenze di natura umana, ad esempio dalle simmetrie del tavoliere, e non ricorre a trucchi né a librerie di partite o a particolari configurazioni. In base alle regole del Go, durante i primi tre giorni di autoallenamento, esso ha giocato quasi cinque milioni di partite contro sé stesso, in rapida successione, acquisendo abilità sovrumane – e di gran lunga superiori persino a quelle di AlphaGo, che impiegò mesi per battere un campione – pur senza essere stato preventivamente “istruito” da esperti umani a riconoscere specifiche o inusuali posizioni: da solo, è riuscito a padroneggiare il gioco!

In un’altra versione, appositamente predisposta per fare un confronto, AlphaGo Zero è stato addestrato sulla base di partite giocate da campioni umani: ha sì imparato più rapidamente, ma ha dato prestazioni peggiori nel lungo periodo.

In sintesi, si può affermare che, lasciato a sé, AlphaGo Zero abbia imparato in modo diverso dagli umani, combinando mosse sì conosciute, ma in un differente ordine, e trovando sequenze di mosse “geniali”, fino ad allora sconosciute. Stupisce come si possano scoprire, in così poco tempo, inedite conoscenze in un gioco ove, per giungere alle attuali, sono occorsi millenni!

Spero che questo tipo di algoritmi, che si rinforzano costantemente mediante autoapprendimento, trovi ben altre applicazioni nel prossimo futuro, magari in medicina, o per scoprire farmaci o materiali con nuove proprietà, sebbene il mondo reale sia incerto e disordinato, a differenza del dominio, ben strutturato, delle regole precise e limitate del Go!

Un ulteriore salto di qualità si è avuto più recentemente con *AlphaZero*, un sistema “generalizzato”, in grado di giocare sia a scacchi, sia a Shōgi, sia a Go (si veda: David Silver, Thomas Hubert, Julian Schrittwieser e altri ricercatori di DeepMind, *A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play*, Science, Vol. 362, No. 6419, 7 dicembre 2018, pp. 1140-1144).

Si tratta di un sistema ancora sperimentale che, come i suoi predecessori, ha costi elevatissimi e non è disponibile in commercio; impiega una sola rete neurale (“convoluzionale profonda, addestrata per rinforzo”) e impara da sé ciascun gioco partendo da *tabula rasa*, se si escludono, ovviamente, le mere regole del gioco... ma nulla di più! Sfruttando un sistema assai potente, che permette un uso massiccio del calcolo parallelo, AlphaZero ha giocato contro sé stesso innumerevoli partite e, in qualche decina di ore di autoapprendimento, ha raggiunto livelli sovrumani in tutti e tre i giochi. E non solo: ha superato la forza dei migliori motori del momento, vincendo – già sul finire del 2017 – tre *match* di 100 partite, con la disponibilità di un minuto per ogni mossa, contro Stockfish 8, elmo/yaneura ou e AlphaGo Zero, rispettivamente, nei tre giochi citati. La decisione di fissare un tempo per mossa fu assai criticata da Tord Romstad, poiché Stockfish può individuare i momenti critici, riservandosi l’opportunità di spendere più tempo per le mosse ritenute dubbie.

Sorprende anche il notevole divario tra il numero di posizioni analizzate da Stockfish (circa 70 milioni al secondo) e da AlphaZero (appena 80 mila al secondo): quest’ultimo usa la sua rete neurale “profonda”, continuamente aggiornata, per concentrarsi assai più selettivamente sulle varianti più promettenti.

Un aspetto chiave – e un po’ dolente – di questa tecnologia è che i criteri con cui è condotta una partita non sono direttamente comprensibili, nemmeno da parte di chi ha progettato la rete neurale: non si può sapere *perché* la rete abbia calibrato i suoi pesi o parametri, che si contano a milioni, proprio su quei certi valori; non sappiamo *come* abbia “ragionato” per giungere a quei risultati, e quindi – in sostanza – l’algoritmo non può essere descritto, né tantomeno spiegato.

Si può tentare di capire o dedurre, a posteriori, quali regole abbia adottato il sistema, analizzando a fondo le partite da esso giocate nei *match*; ed è proprio ciò che, per quanto concerne gli scacchi, si sono proposti il grande maestro Matthew Sadler e la maestra internazionale Natasha Regan, con un recentissimo e corposo volume di settecento pagine, assai impegnative: *Game Changer* (New In Chess, 2019). Nella prefazione, l’ex-campione del mondo Garry Kasparov giudica AlphaZero un giocatore eccezionale, dotato di uno stile di gioco del tutto insolito ed estremamente complicato, propenso ai sacrifici di pezzi in vista di un vantaggio strategico sul lungo termine. In effetti, lo stile appare dinamico e aperto, con preferenze per le posizioni aggressive e rischiose... Si può parlare di “comprensione superiore” del gioco rispetto a quella umana? Certo è, come si è detto, che AlphaZero analizza molti meno stati dei giocatori artificiali tradizionali, ed è altrettanto vero che genera nuova conoscenza da cui si può imparare: starà agli esperti spiegarla!

Devo però ricordare un altro motore scacchistico, *open-source* come Stockfish, *Leela Chess Zero* (<https://lczero.org/>), sviluppato nel 2018 da Gary Linscott e altri, in modo assai simile ad AlphaZero, poiché usa le stesse rivoluzionarie tecniche: il mese scorso – febbraio 2019 – ha vinto, imbattuto, la seconda edizione del Top Chess Engine Championship. Vedremo che cosa accadrà nelle prossime sfide...

Per concludere questa già troppo lunga prefazione, dico soltanto che ho approfittato dell'occasione per rimediare a varie imprecisioni nel testo stampato nel 2012, a un errore (una *h*, mutata in *x*, in fondo alla quarta riga di pagina 45) e a un refuso (in un cognome a pagina 272), oltre che per chiarire o giustificare o approfondire taluni passaggi in modo conveniente, allo scopo di migliorare la comprensione dei concetti esposti.

Sono riuscito comunque a mantenere gli stessi numeri di pagina d'inizio di ciascuno degli undici capitoli che costituivano la prima edizione.

Gli algoritmi sono presentati in diversi stili: con una semplice descrizione a parole, per quanto possibile precisa, o con uno pseudo-codice, o con un codice dettagliato, in diversi linguaggi o paradigmi di programmazione, scegliendo di volta in volta la forma più adatta, a seconda delle esigenze dell'esposizione.

Alcune questioni sono soltanto accennate o lasciate aperte, con l'auspicio che siano di stimolo per riflessioni e ricerche individuali da parte del lettore.

Nell'appendice aggiunta alla presente edizione, ho toccato qualche aspetto rilevante delle classiche applicazioni informatiche, perlomeno qualcuno dei tanti argomenti del tutto trascurati o appena menzionati nella prima edizione. Alla pagina 301 è presentato il contenuto di questa appendice, che appresso seguirà.

Infine, ho aggiornato i riferimenti alle pagine *web*, taluni ormai obsoleti o addirittura non più disponibili (come purtroppo quello al *database* di Awari, presso l'università di Amsterdam), e ho compilato l'indice dei nomi.

Confido, almeno, che il mio lavoro torni utile a qualcuno, pur nella consapevolezza dell'effimerità di questo genere di testi.

Genova, 4 marzo 2019

*L. R.*

## Prefazione alla prima edizione

Questo libro è nato nella scuola e per la scuola. Esso prende impulso dalla mia duplice esperienza di insegnante di informatica nella scuola secondaria superiore e di componente del comitato scientifico per le gare *Kangourou dell'Informatica*; vuol essere pertanto, da un lato, una rivisitazione e un arricchimento di alcune mie lezioni – quelle di argomento più o meno ludico – raccolte nel corso degli anni e, dall'altro, un approfondimento delle problematiche proposte nelle prove del *Kangourou*.

I destinatari naturali di questo lavoro sono dunque gli studenti (sia coloro che intendano attrezzarsi per partecipare alle gare di informatica organizzate nel mondo della scuola, sia quelli già arrivati agli studi universitari), sebbene il volume sia strutturato in modo da poter essere consultato da tutti i cultori di “passatempi informatici” o della programmazione di giochi al computer, qualunque sia il loro livello di preparazione tecnica.

Perciò il linguaggio usato è piano e divulgativo, seppur concettualmente rigoroso. Tuttavia la complessità di certi temi affrontati ha reso inevitabile l'esposizione di alcune parti di difficile lettura per chi non abbia una buona conoscenza della programmazione; il lettore non abbia alcun timore a tralasciare le parti più tecniche o per lui ostiche, poiché la comprensione delle linee generali del discorso non ne sarà affatto compromessa.

Non ostacola la comprensione neppure la voluta omissione di risposte, o soluzioni, ad alcune domande, o piccoli problemi, formulati nel corso della trattazione, dal momento che essi vogliono costituire uno stimolo alla riflessione e hanno valenza di esercizio per il lettore che desideri scendere più nel dettaglio o che intenda cimentarsi nella progettazione e nella conseguente realizzazione di programmi al computer.

I ricorrenti riferimenti a quesiti proposti nelle passate edizioni delle gare *Kangourou dell'Informatica* rinviano ad un ampio materiale reperibile interamente in rete, al sito ufficiale di Kangourou Italia, <http://www.kangourou.it/>, che raccomando comunque di visitare, a prescindere dalla ricerca di detti riferimenti.

Il libro è idealmente organizzato in due parti, la prima comprendente i capitoli fino al quinto, la seconda i rimanenti sei. Il primo capitolo contiene una rassegna di quesiti, allo scopo di illustrare alcuni algoritmi efficienti che operano su grafi, per trovare ad esempio i cammini di costo minimo o il flusso massimo in una rete di trasporto. Il problema dello *scheduling*, più oneroso quanto all'aspetto computazionale, fornisce poi il pretesto per introdurre la nozione di algoritmo di approssimazione; gli ultimi quesiti riguardano invece successioni numeriche con certe curiose proprietà. Il secondo capitolo è dedicato alla logica e ai suoi legami con l'informatica, con particolare attenzione alle questioni che non possono essere risolte da una macchina, ancorché programmabile.

Questo discorso prosegue nel capitolo successivo, dove si cerca di capire che cosa sia essenziale per poter esprimere qualsivoglia procedimento di calcolo. Diversi metodi, antichi eppur moderni, sono considerati – a titolo d'esempio – per il calcolo della radice quadrata. Si discute poi di come descrivere la sintassi di un linguaggio (non solo di programmazione) e di quali automi siano in grado di compierne – se possibile – il riconoscimento.

Il quarto capitolo è il più lungo e forse quello di più ardua lettura, poiché dedicato ai problemi decisamente impegnativi a causa dell'enorme mole di calcoli che richiedono – per quanto si sa – al crescere della dimensione dei loro dati. Oltre a tre classici problemi (detti del commesso viaggiatore, dell'imballaggio e dello zaino) sono pure approcciati quelli della primalità e della fattorizzazione di grandi numeri, ed è esemplificato un famoso sistema di crittografia a chiave asimmetrica.

La prima parte si conclude con un capitolo che illustra differenti metodi, alquanto semplici, per ottenere immagini d'effetto con il computer, giungendo a dare uno sguardo ai sistemi dinamici il cui comportamento dà origine al caos o a un frattale. Una selezione di tali immagini è presentata nelle tavole a colori fuori testo, in coda al volume.

Mentre nella prima parte i procedimenti di calcolo sono rappresentati, in prevalenza, mediante una pseudo-codifica (senza parentesi che racchiudano sequenze di istruzioni, bensì sfruttando l'incolonnamento: si ponga quindi attenzione specie alle componenti di un ciclo o ad associare un “altrimenti” al “se” sotto il quale è scritto), nella seconda parte è usato un linguaggio a oggetti, il C++, per dettagliare le operazioni più interessanti e utili per la realizzazione di un gioco al computer.

Dedicato il sesto capitolo ai cosiddetti automi cellulari (usati per modellare tanti fenomeni fisici, e pur dotati di capacità di computazione universale), i due successivi (da leggere nell'ordine) spiegano alcuni noti rompicapi, ma con diverse caratteristiche dal punto di vista informatico: dal giro del cavallo sulla scacchiera al gioco del quindici (risalenti, rispettivamente, al IX e al XIX secolo), con variazioni sui temi e relativi algoritmi, sia euristici sia esaustivi. In un'appendice è riportata la definizione completa (sotto forma di classe parametrica) di un tipo di dato che riveste notevole importanza in informatica: la pila. Il nono e il decimo sono i soli altri capitoli da leggere in sequenza: riguardano i giochi, tra due avversari e a informazione perfetta, come gli scacchi. (Se qualcuno dei lettori non conosce le regole degli scacchi, a cui più volte è fatto riferimento, non me ne voglia; degli altri giochi, invece, sono elencate le regole.) Vi si spiegano le procedure che, in generale, un programma può adottare per decidere la prossima mossa in un tempo ragionevole.

L'ultimo capitolo concerne, in particolare, quei giochi dove non c'è distinzione di campo e che non possono finire in parità; uno di questi (*Babylone*, di cui è suggerita una variante) è trattato in modo piuttosto approfondito, poiché una sua analisi accurata non si trova in rete.

A tal proposito, rammento che nel libro i riferimenti (ad autori, testi, articoli, siti internet) sono esplicitati ove occorre, con specifico riguardo alle origini o alla storia di un algoritmo o di un gioco; il lettore potrà quindi cercare autonomamente in rete tutto ciò che gli sarà necessario per una maggior comprensione dei contenuti, visitando ad esempio i siti di università italiane, o di quelle americane citate nel corso dell'esposizione.

Concludo porgendo i miei sinceri ringraziamenti all'editore Angelo Lissoni, presidente di Kangourou Italia, che mi ha proposto la stesura di questo volume, rendendo così fruibili appunti e idee a un pubblico ben più nutrito dei miei allievi (almeno questa è la mia prima speranza); a Mauro Torelli dell'Università di Milano, per i consigli e le puntuali osservazioni su alcuni dei problemi presentati nel primo capitolo; a Giuseppe Rosolini dell'Università di Genova, per i preziosi suggerimenti relativi al secondo capitolo; a Nicola Rebagliati per le sue letture, accompagnate da pareri e incoraggiamenti costanti; a mia moglie Silvia per il consistente aiuto e la pazienza richiesta dalla revisione del manoscritto; ad Alessandro Bonanno – che da un quarto di secolo condivide con me la sempre nuova esperienza dell'insegnamento – per l'assistenza prestata nella preparazione delle immagini e dell'impaginazione finale.

Ringrazio anticipatamente anche coloro che esprimeranno critiche o suggerimenti, o segnaleranno sviste o eventuali errori, scrivendomi all'indirizzo di *e-mail*:

[lorenzo.repetto@calvino.edu.it](mailto:lorenzo.repetto@calvino.edu.it)

Buona lettura!

Genova, 10 marzo 2012

*L. R.*

## **Parte prima**

**Con carta, matita, gomma  
... e computer**



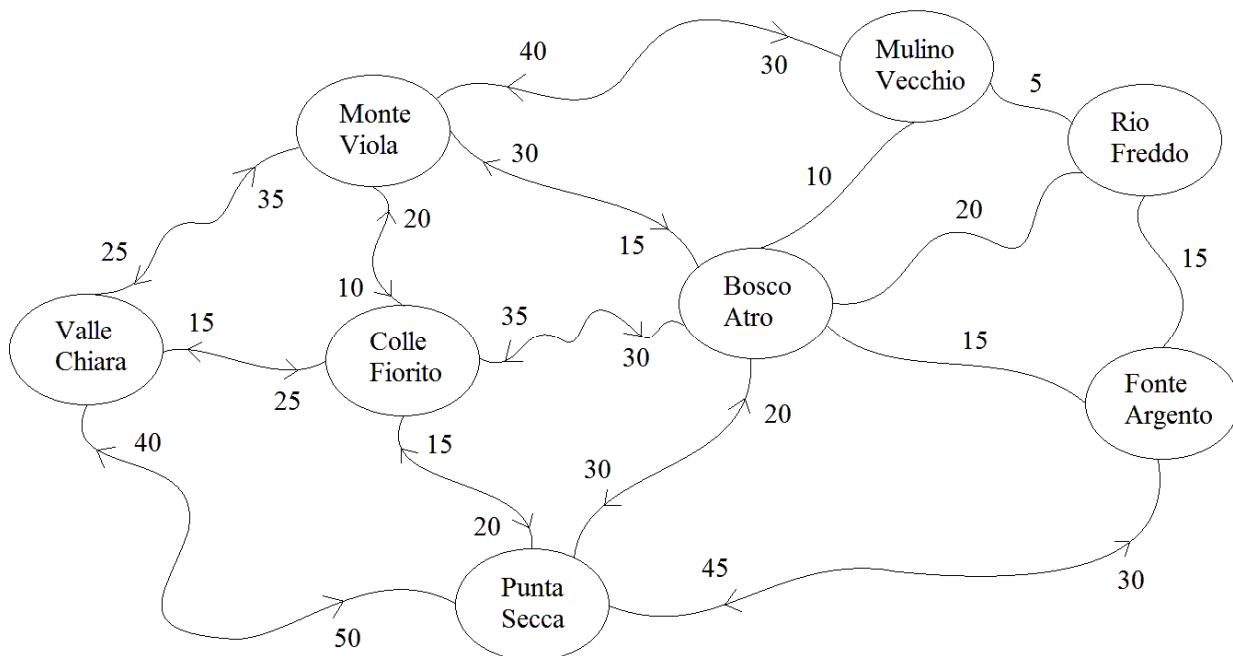
# 1. Un inizio “problematico”

Nel primo capitolo proponiamo una *miscellanea* di quesiti nello stile delle passate gare *Kangourou dell'Informatica*. Di ciascuno sarà data e discussa la soluzione, inquadrando il problema nel suo contesto informatico; ulteriori approfondimenti di taluni aspetti paradigmatici troveranno più ampio spazio nei capitoli successivi.

## Il percorso più breve.

Aldo si trova a Valle Chiara e vuole andare a Fonte Argento. Egli dispone di una mappa sulla quale sono riportati i tempi di percorrenza (espressi in minuti) dei sentieri che può seguire tra le località segnalate. Solo per alcuni tratti, più pianeggianti, è stato indicato un unico tempo, che vale per entrambi i sensi di percorrenza.

Qual è il percorso che gli permette di arrivare a destinazione nel tempo più breve?



**Soluzione.** Gli conviene salire al Monte Viola e quindi passare nel Bosco Atro: in 65 minuti arriverà a Fonte Argento...

Ma noi come possiamo giungere alla risposta? Esistono diversi procedimenti, in generale; ne descriviamo uno ormai classico: l'algoritmo formulato nel 1959 da uno dei grandi nomi dell'informatica, l'olandese Edsger W. Dijkstra (1930-2002), che diede fondamentali contributi allo sviluppo dei linguaggi di programmazione, anche concorrenti, e alle dimostrazioni di correttezza dei programmi.

Astraendo, la mappa di Aldo è riconducibile a un *grafo*, che – per massima semplicità – può essere rappresentato da una *matrice* siffatta:

-	35	25	50	-	-	-	-
25	-	10	-	15	30	-	-
15	20	-	20	30	-	-	-
40	-	15	-	20	-	-	30
-	30	35	30	-	10	20	15
-	40	-	-	10	-	5	-
-	-	-	-	20	5	-	15
-	-	-	45	15	-	15	-

Nel nostro caso, questa matrice – chiamiamola  $C$  – contiene i tempi di percorrenza; in particolare, il valore dell’elemento di riga  $i$  e colonna  $j$ , che nel seguito indicheremo con  $C(i, j)$ , è il tempo richiesto per percorrere il tratto di sentiero (*arco*) dalla località (*nodo*)  $i$  alla località  $j$ . Il trattino indica l’assenza dell’arco diretto corrispondente; in particolare, la diagonale principale contiene soltanto trattini. (Per tutti i grafi, ipotizzeremo sempre che, per ogni  $i$  e  $j$ , vi sia, al più, un arco da  $i$  a  $j$ .)

La matrice è dunque quadrata,  $n \times n$ , se  $n$  è il numero di nodi nel grafo.

Naturalmente, per ordinare i tempi nella matrice, abbiamo dapprima numerato le località, da 1 a 8, in maniera arbitraria:

1 – Valle Chiara	2 – Monte Viola	3 – Colle Fiorito
4 – Punta Secca	5 – Bosco Atro	6 – Mulino Vecchio
7 – Rio Freddo	8 – Fonte Argento	

In generale, la matrice contiene i *costi* o *pesi* (non negativi) degli archi, che ad esempio possono rappresentare, anziché i tempi di percorrenza, le lunghezze o i prezzi da pagare relativi ai singoli tratti.

Supponiamo, dunque, che sia fissato il nodo di partenza; chiamiamolo  $v_0$  (nell’esempio,  $v_0 = 1$ ). Sebbene ci interessi, in particolare, un *cammino ottimo* (cioè di costo minimo) per arrivare a un certo nodo (nell’esempio, il nodo 8), nei casi più sfavorevoli (com’è appunto quello dell’esempio) finiremo col calcolare un cammino ottimo per arrivare a *ciascun nodo* raggiungibile da  $v_0$ .

L’algoritmo usa una *struttura di dati* ausiliaria,  $S$ , che rappresenta l’insieme dei nodi  $v$  per cui sia noto il costo minimo da  $v_0$  a  $v$ . Inizialmente, l’unico nodo in  $S$  è proprio  $v_0$ : banalmente, per andare da  $v_0$  a  $v_0$ , il costo è nullo; procedendo sino in fondo, fatti  $n - 1$  passi,  $S$  sarà costituito da tutti i nodi del grafo (anche quelli eventualmente non raggiungibili a partire da  $v_0$ , per i quali il costo minimo non sarà definito).

Per rappresentare  $S$ , si può usare un *array di  $n$  bit* (cifre binarie): possiamo pensarlo come una matrice di una sola riga, dove l’elemento  $i$ -esimo vale 1 se e soltanto se il nodo  $i$  appartiene a  $S$ . (Qui gli elementi sono distinti con un indice da 1 a  $n$ .)

L’algoritmo prepara i risultati in altri due array,  $D$  e  $P$ , entrambi di  $n$  elementi numerici. Ad ogni passo del calcolo che illustreremo, se  $v$  appartiene a  $S$ , allora  $D(v)$  è il costo minimo *definitivo* da  $v_0$  a  $v$ , altrimenti è il costo minimo *temporaneo* per andare da  $v_0$  a  $v$  passando esclusivamente per nodi appartenenti a  $S$ .

Se non è possibile andare da  $v_0$  a  $v$  transitando soltanto per nodi in  $S$ , allora si può assegnare a  $D(v)$  un valore negativo (ad esempio  $-1$ ), da riguardare proprio secondo questo preciso significato, e se ne rimanda il calcolo ad un eventuale momento più favorevole.

L'array  $P$  servirà invece a ricostruire i cammini ottimi: ad ogni passo del calcolo,  $P(v)$  sarà il penultimo nodo sul cammino da  $v_0$  a  $v$  che per il momento è il migliore (“penultimo” vuol dire dal quale si giunge in  $v$  percorrendo l'ultimo arco).

L'idea di fondo dell'algoritmo è questa: ad ogni passo del calcolo, aggiungiamo a  $S$  un nuovo nodo  $w$ , tale che il costo da  $v_0$  a  $w$ , passando soltanto per nodi che già appartengono a  $S$ , sia il minimo possibile.

L'algoritmo è *greedy* (ingordo, goloso): espande  $S$  includendovi il nodo “più vicino” a  $S$ , sempre considerando i costi per raggiungerlo a partire da  $v_0$ . (Si tratta della stessa idea su cui si basa la costruzione di un *albero ricoprente di costo minimo* secondo Prim-Jarník, che fu presentata nel quesito “Gli elettricisti” della gara finale *Kangourou dell'Informatica* di maggio 2009 e che ritroveremo nel quarto capitolo.) Vediamo come funziona l'algoritmo di Dijkstra sull'esempio proposto.

Gli array di cui si è detto sono così inizializzati:

$S$	=	1	0	0	0	0	0	0
$D$	=	0	35	25	50	—	—	—
$P$	=	—	1	1	1	—	—	—

Infatti, da Valle Chiara (nodo 1) si può giungere, percorrendo un tratto di sentiero, a Monte Viola o a Colle Fiorito o a Punta Secca, rispettivamente con costi 35, 25, 50. Ci accorgeremo che qui occorrerà procedere sino in fondo, facendo tutti gli  $n - 1 = 7$  passi: infatti, la nostra meta sarà raggiunta soltanto all'ultimo passo.

Al primo passo, quale nodo è aggiunto a  $S$ ? Certamente il nodo 3 (Colle Fiorito), poiché qualsiasi altro percorso per giungervi deve passare da Monte Viola o da Punta Secca, quindi in un tempo sicuramente maggiore. Aggiungiamo dunque il nodo 3 a  $S$  e aggiorniamo di conseguenza gli altri due array:

$S$	=	1	0	1	0	0	0	0
$D$	=	0	35	25	45	55	—	—
$P$	=	—	1	1	3	3	—	—

Per il nodo 2 non è cambiato nulla: arrivare a Monte Viola passando da Colle Fiorito non migliora il tempo di 35 minuti che si aveva con il sentiero diretto.

Invece, per il nodo 4, le cose cambiano: conviene passare da Colle Fiorito perché si risparmiano 5 minuti.

Infine, poiché adesso all'insieme  $S$  si è aggiunto il nodo 3, si può arrivare a un nuovo nodo, il 5 (Bosco Atro), e – per quanto ci è dato sapere al momento – in 55 minuti dalla partenza, giungendovi appunto dal nodo 3.

Al secondo passo, il nodo “più vicino” a  $S$  (tra 2, 4 e 5) è il 2 (che ha associato un tempo di 35 minuti): per giungervi da altre vie, si deve passare da 4 o da 5, con tempi

sicuramente superiori. Possiamo quindi affermare di aver trovato il tempo minimo per arrivare al nodo 2. Gli array saranno così aggiornati:

S =	1	1	1	0	0	0	0	0
D =	0	35	25	45	50	65	-	-
P =	-	1	1	3	2	2	-	-

È cambiata la situazione per il nodo 5: ora vi si può arrivare passando dal nuovo nodo in S, il 2, e risparmiando 5 minuti. Inoltre, dal nodo 2 si può arrivare in 30 minuti al nodo 6 (che non era direttamente raggiungibile né da 1, né da 3), e per arrivare al nodo 6 dalla partenza si impiega un tempo dato dalla somma di questi 30 minuti con il tempo richiesto per arrivare al nodo 2 (35 minuti): in tutto 65 minuti, per il momento. I tre passi successivi apportano le seguenti modifiche ai tre array:

3)	S =	1	1	1	1	0	0	0
	D =	0	35	25	45	50	65	- 75
	P =	-	1	1	3	2	2	- 4
4)	S =	1	1	1	1	1	0	0
	D =	0	35	25	45	50	60	70 65
	P =	-	1	1	3	2	5	5 5
5)	S =	1	1	1	1	1	1	0 0
	D =	0	35	25	45	50	60	65 65
	P =	-	1	1	3	2	5	6 5

e se, ad esempio, la nostra meta fosse Mulino Vecchio, potremmo fermarci qui. Infine, gli ultimi due passi aggiungono a S rispettivamente i nodi 7 e 8, ma non cambiano più nulla né in D, né in P.

In conclusione, per andare dal nodo 1 al nodo 8, si impiegheranno  $D(8) = 65$  minuti; il percorso si determina procedendo a ritroso: al nodo 8 si arriva dal nodo  $P(8) = 5$ , a questo si arriva da  $P(5) = 2$ , e a questo da  $P(2) = 1$ , cioè dal nodo di partenza.

In questo caso tutti i nodi sono stati raggiunti a partire da 1; i cammini ottimi da 1 a ciascun altro nodo formano un *albero* (ottimo) che ha come *radice* il nodo di partenza (e che nella fattispecie ricopre interamente il grafo: si provi a disegnarlo), i cui *rami* o parti di ramo costituiscono a loro volta cammini ottimi.

In generale, dati il nodo  $v_0$  di partenza e la matrice  $C$  ( $n \times n$ ) dei costi, l'algoritmo di Dijkstra può essere descritto come alla pagina seguente. Bisognerebbe dimostrare, ovviamente, che esso *termina sempre* – questo è un compito semplice! – e che è *corretto*, ossia che fornisce i risultati giusti per ricavare i costi e i cammini ottimi a partire dal nodo fissato, secondo la modalità illustrata per il caso esemplificato.

Se si prova ad eseguirlo a partire dal nodo 8 (Fonte Argento), si scopre che il percorso più rapido per il ritorno a Valle Chiara prevede di ripassare per Bosco Atro, ma poi per Colle Fiorito, impiegando ancora 65 minuti; lo stesso tragitto dell'andata ne richiederebbe 5 in più.

```

per  $v = 1, \dots, n$ :
     $S(v) \leftarrow 0$ 
 $S(v_0) \leftarrow 1$ 
per  $v = 1, \dots, n$ :
    se esiste l'arco da  $v_0$  a  $v$  allora
         $D(v) \leftarrow C(v_0, v)$ ;  $P(v) \leftarrow v_0$ 
    altrimenti
         $D(v) \leftarrow -1$ ;  $P(v) \leftarrow -1$  // -1 corrisponde al trattino in output nell'esempio
 $D(v_0) \leftarrow 0$ ;  $P(v_0) \leftarrow -1$ 
 $i \leftarrow 2$ 
finché  $i < n$ :
     $min \leftarrow -1$ 
    per  $v = 1, \dots, n$ :
        se  $S(v) = 0$  allora
            se  $D(v) \geq 0$  allora
                se  $min < 0$  o  $D(v) < min$  allora
                     $w \leftarrow v$ ;  $min \leftarrow D(v)$ 
            altrimenti
                se  $min < 0$  allora  $w \leftarrow v$ 
        se  $min < 0$  allora FINE.
         $S(w) \leftarrow 1$ 
    per  $v = 1, \dots, n$ :
        se esiste l'arco da  $w$  a  $v$  e  $S(v) = 0$  allora
             $c \leftarrow min + C(w, v)$ 
            se  $D(v) < 0$  o  $c < D(v)$  allora
                 $D(v) \leftarrow c$ ;  $P(v) \leftarrow w$ 
 $i \leftarrow i + 1$ 

```

Ogni volta che si desidera conoscere i costi e i cammini ottimi da un dato nodo a ciascun altro nodo di un dato grafo, eseguendo l'algoritmo sopra specificato, ci si deve attendere un tempo di elaborazione che, al crescere di  $n$  (numero dei nodi), tende ad aumentare proporzionalmente al *quadrato* di  $n$ .<sup>1</sup>

Raccomandiamo una ricerca su un altro classico algoritmo, dovuto a Robert W. Floyd, per ricavare – tutti in una volta – costi e cammini ottimi relativi a *ciascuna coppia* di nodi: di ancor più facile stesura, il suo costo computazionale tende ad aumentare in modo proporzionale al cubo di  $n$ .

La struttura più semplice per rappresentare un grafo *non pesato* consiste in una matrice di bit,  $A$  ( $n \times n$ ), detta *matrice di adiacenza*:  $A(i, j) = 1$  se e soltanto se esiste l'arco diretto dal nodo  $i$  al nodo  $j$ . Anche in tal caso ha senso chiedersi quali siano i cammini più brevi: quelli formati dal minor numero di archi.

Nel quarto capitolo torneremo a parlare di grafi, e vedremo che, quando vogliamo

<sup>1</sup> L'efficienza può essere migliorata, specialmente trattando grafi “non densi” (cioè con una modesta percentuale di archi rispetto a tutti i possibili), se si adottano strutture di dati più sofisticate, come le *liste di adiacenza*, contenenti i costi dei singoli archi, e le *code a priorità*.

trovare invece i cammini più lunghi o che abbiano particolari caratteristiche, come quella di toccare una e una sola volta tutti i nodi, allora il costo computazionale è assai più elevato, in generale, stando alle attuali conoscenze in materia.

Qui elenchiamo alcuni tipici problemi che riguardano i grafi in generale, anche non pesati. Invitiamo il lettore a svolgere un'indagine personale sugli algoritmi (efficienti) per risolverli, che richiedono un tempo di esecuzione polinomiale nel numero di nodi.

- ◆ *Costruzione di un albero di visita*, a partire da un nodo dato, i cui rami tocchino tutti i nodi raggiungibili. (Ne vedremo un paio di soluzioni qui sotto.)
- ◆ *Test di connessione forte*: stabilire se, per ogni coppia (ordinata) di nodi, esiste un cammino dal primo al secondo. (Si può procedere costruendo un albero di visita a partire da ciascun nodo... Si veda anche il quesito “Network”, pubblicato con le opportune spiegazioni nel libretto *Kangourou dell’Informatica* del 2015.)
- ◆ *Ordinamento topologico dei nodi*: ammesso che il grafo sia *aciclico*, cioè privo di cicli, rinumerare i nodi in modo tale che, per ogni nodo, tutti i nodi da esso raggiungibili (percorrendo uno o più archi) abbiano un numero maggiore del suo. (Si veda il quesito “Cerimoniale”, anch’esso nel libretto del 2015.)
- ◆ *Calcolo della chiusura transitiva*: calcolare la matrice di adiacenza  $T$  tale che  $T(i, j) = 1$  se e soltanto se nel grafo dato vi sia un cammino (costituito da uno o più archi) dal nodo  $i$  al nodo  $j$ . Ovviamente, la nuova matrice  $T$  avrà a valore 1 almeno gli elementi che hanno valore 1 nella matrice di adiacenza del grafo dato. (Si veda, in particolare, l’algoritmo dovuto a Stephen Warshall, basato sulla stessa idea di quello di Floyd – riconducibile alla *programmazione dinamica* di cui parleremo più avanti – e pubblicato nello stesso anno, il 1962.)
- ◆ *Test di aciclicità*: stabilire se non vi siano cicli. (Per risolvere la questione, si possono adattare sia la visita in profondità – alla quale ora accenneremo – sia gli algoritmi che calcolano un ordinamento topologico o la chiusura transitiva.)

Illustriamo brevemente le due usuali modalità di visita di un grafo: in profondità (*depth-first search*, DFS) e in ampiezza (*breadth-first search*, BFS), allo scopo di costruire un albero di visita. La prima, in *profondità*, può essere effettuata sfruttando la *ricorsione* (e quindi la *pila* di cui è dotato il sistema di elaborazione).

Sia data la matrice di adiacenza  $A$  ( $n \times n$ ) e sia fissato il nodo di partenza  $v_0$  (che sarà la radice dell’albero di visita). Il risultato sarà costruito in un array  $S$  (come prima), con  $S(v_0)$  inizializzato a 1 e gli altri elementi a 0 (soltanto il nodo di partenza è stato visitato), e una matrice di adiacenza  $T$ , con tutti gli elementi (binari) inizializzati a 0.

DFS (*nodo\_corrente*):

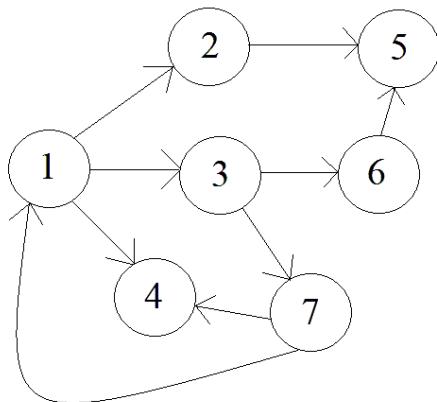
```

per  $v = 1, \dots, n$  :
    se  $A(\text{nodo\_corrente}, v) = 1$  e  $S(v) = 0$  allora
        // si può andare con un arco da nodo_corrente a  $v$ , non ancora visitato
         $T(\text{nodo\_corrente}, v) \leftarrow 1$ 
         $S(v) \leftarrow 1$ 
        DFS( $v$ )
    
```

Il calcolo sarà innescato applicando la procedura (*ricorsiva*) DFS al nodo  $v_0$ . Se alla fine qualche elemento di  $S$  rimane a 0, vuol dire che non tutti i nodi sono stati raggiunti. La matrice  $T$  individua gli archi che formano l’albero di visita. Il calcolo può essere descritto da una procedura *iterativa* che usi esplicitamente una *pila*, inizialmente vuota, la quale possa ospitare *coppie* ordinate di interi (ciascuna delle quali rappresenterà un arco):

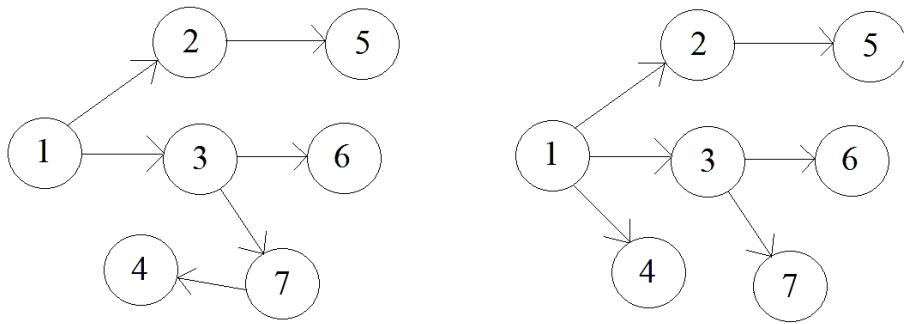
```
// S e T inizializzati come sopra, in particolare S( $v_0$ ) ← 1
per  $v = 1, \dots, n$ :
  se  $A(v_0, v) = 1$  allora
    pone sulla cima della pila la coppia  $(v_0, v)$ 
finché la pila non è vuota :
  preleva dalla pila la coppia che si trova in cima: sia essa  $(a, b)$ 
  se  $S(b) = 0$  allora
     $T(a, b) \leftarrow 1$ 
     $S(b) \leftarrow 1$ 
    per  $v = 1, \dots, n$ :
      se  $A(b, v) = 1$  allora
        pone sulla cima della pila la coppia  $(b, v)$ 
```

Alla pila si accede soltanto da una parte (la cima). Se al posto della pila si usa una *coda* (alla quale si accede da entrambe le parti: da una si inserisce, dall’altra si preleva), allora si effettua una visita in *ampiezza*, atta a determinare i cammini col minor numero di archi (si vedano i quesiti “Segnali di fumo” e “Cambiavalute”, nei libretti del 2014 e 2015). Ripareremo di pile e code, usandole in diverse occasioni. Per fare un esempio conclusivo, consideriamo il grafo qui sotto rappresentato.



Notiamo la presenza di un ciclo:  $[1, 3, 7, 1]$ ; inoltre, soltanto a partire da uno dei tre nodi 1, 3 e 7, indifferentemente, è possibile raggiungere un qualsiasi altro nodo.

Supponendo di partire dal nodo 1, l’algoritmo *ricorsivo* di visita in profondità toccherà gli altri nodi nel seguente ordine: 2, 5, 3, 6, 7, 4; l’algoritmo *iterativo* di visita in ampiezza seguirà invece l’ordine: 2, 3, 4, 5, 6, 7. I rispettivi alberi di visita sono rappresentati, a sinistra e a destra, nella figura che segue. (Quale sarà invece l’albero prodotto dall’algoritmo iterativo di visita in profondità delineato sopra?)



*Nota.* L'algoritmo alla pagina precedente può essere semplificato se lo si specializza per il caso di visita *in ampiezza*, con uso di una *coda di interi* (ossia nodi) inizialmente vuota:

// stesse inizializzazioni di S e T: tutti gli elementi a 0, tranne  $S(v_0) \leftarrow 1$   
mette in coda  $v_0$

**finché** la coda non è vuota :

estrae dalla coda il primo elemento: sia esso  $i$  ;

**per**  $j = 1, \dots, n$  :

**se**  $A(i, j) = 1$  e  $S(j) = 0$  **allora**

$T(i, j) \leftarrow 1$

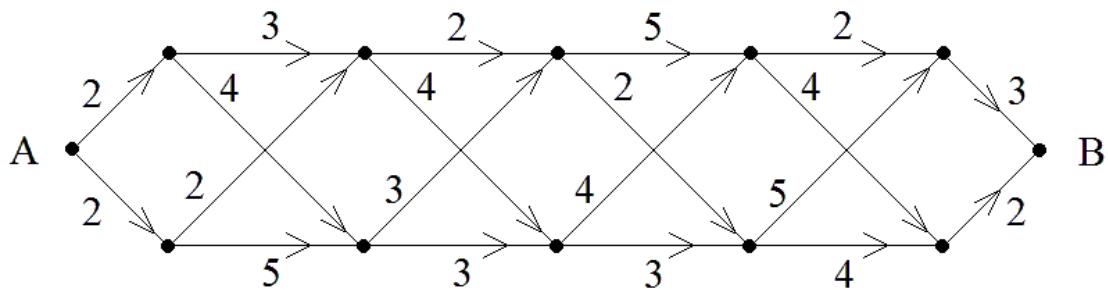
$S(j) \leftarrow 1$

mette in coda  $j$

Il risultato prodotto è lo stesso. Attenzione: se qui si usasse una pila (anziché una coda), il risultato *non* sarebbe una visita in profondità, in generale!

## Un percorso a stadi successivi.

Si deve partire dal punto A e arrivare al punto B, procedendo da sinistra verso destra, obliquamente o in orizzontale, com'è indicato dalle frecce.



Da A si hanno due vie possibili, e così pure da ciascuno dei punti dei cinque stadi successivi; ma, una volta giunti all'ultimo stadio, ci sarà soltanto la via per B.

- Quanti sono i percorsi possibili da A a B?
- E quanti invece sarebbero, se ci fossero tre punti ad ogni stadio (anziché due soltanto), direttamente raggiungibili da ciascuno dei punti allo stadio precedente?
- In generale, come si può determinare un percorso – senza esplorarli tutti – corrispondente alla minima (o alla massima) somma dei “pesi” associati ai segmenti che lo compongono?

**Soluzioni.** Con due punti, come in figura, i percorsi possibili da A a B sono  $2^5$ ; con tre,  $3^5$ ; in generale, se  $p$  è il numero di punti ad ogni stadio e  $s$  è il numero di stadi, i percorsi possibili sono  $p$  elevato alla potenza  $s$ . (Dal computo degli stadi sono esclusi il punto iniziale e quello finale, unici e obbligati.)

Per determinare un percorso con la minima (o la massima) somma dei pesi associati ai segmenti che lo compongono, si può procedere efficientemente in questo modo:

- a ciascun punto dell'ultimo stadio si associa il peso del segmento che lo unisce a B;
- per ogni stadio, procedendo a ritroso dal penultimo al primo: a ciascun punto dello stadio corrente si associa la somma minima (o massima) tra il peso del segmento che lo unisce a un punto dello stadio successivo e il numero (già calcolato!) associato a tale punto – quindi si fanno  $p$  somme per ciascun punto;
- *idem*, infine, per il punto A – e quindi in totale si sono fatte  $p^2(s - 1) + p$  somme (tante quanti sono tutti i segmenti meno  $p$ );
- il numero associato ad A è la somma dei pesi dei segmenti da percorrere fino a B;
- un percorso come richiesto si può ora determinare, procedendo in avanti, purché, per ogni punto, si sia ricordato quale punto dello stadio successivo ha determinato il numero ad esso associato…

Nel caso proposto, il percorso con la minima somma dei pesi (14) è unico: giù, su, orizzontale, giù, orizzontale, e infine su in B; due sono invece i percorsi con la massima somma dei pesi (21): giù, orizzontale, su, orizzontale, giù, su in B, oppure: giù, orizzontale, orizzontale, orizzontale, su, giù in B.

Avremmo potuto anche iniziare dal primo stadio, procedendo in avanti:

- a ciascun punto del primo stadio si associa il peso del segmento che vi arriva da A;
- per ognuno degli stadi successivi: a ciascun punto dello stadio corrente si associa la somma minima (o massima) tra il peso del segmento che vi arriva da un punto dello stadio precedente e il numero (già calcolato!) associato a tale punto;
- *idem*, infine, per il punto B, a cui rimane associata la somma dei pesi dei segmenti da percorrere per giungervi;
- un percorso come richiesto si può ora determinare, procedendo a ritroso, purché, per ogni punto, si sia ricordato quale punto dello stadio precedente ha determinato il numero ad esso associato. Il numero delle operazioni fatte è esattamente lo stesso.

L'oggetto del problema proposto è un caso particolare di *grafo orientato aciclico*. Naturalmente, l'applicazione dell'algoritmo di Dijkstra avrebbe portato allo stesso risultato, eseguendo però un maggior numero di operazioni.

L'idea risolutiva che abbiamo esposto è quella della *programmazione dinamica*, tecnica assai versatile concepita da Richard E. Bellman intorno alla metà degli anni '50 del Novecento. (Per inciso, l'algoritmo di Dijkstra costituisce un esempio di estensione dello stesso principio a una più ampia categoria di grafi.)

Se supponiamo di operare nello spazio degli stati relativi a un certo problema, ciò che è importante non è tanto come si sia giunti nello stato attuale, quanto il proseguire nel modo migliore verso lo stato finale. In altri termini, ciascuna parte terminale di una traiettoria ottima è essa stessa una traiettoria ottima, relativa a un problema di ridotta dimensione.

Questo principio ci tornerà utile in alcuni esempi di algoritmi, che proporremo nel quarto capitolo, per risolvere nel modo noto più rapido, in generale, certi problemi particolarmente onerosi in termini di tempo. Tuttavia, possiamo già intuire il rovescio della medaglia, e cioè la quantità di memoria richiesta per mantenere tutti i risultati parziali, che alla fine serviranno per la ricostruzione della traiettoria ottima.

Il fatto che si parli di traiettoria o di stadi successivi non deve necessariamente far pensare a processi temporali, magari in cui si devono prendere delle decisioni in base alle informazioni disponibili sullo stato attuale e a ciò che da esse si riesce a dedurre, con probabilità o con certezza, circa lo sviluppo futuro del sistema – sebbene, in origine, Bellman abbia coniato la locuzione “programmazione dinamica” riferendosi proprio a quei processi in cui si richiede di trovare le decisioni migliori l’una dopo l’altra. Rientrano in questa casistica certi problemi di controllo ottimo, di programmazione delle scorte, di produzione di serie o di approvvigionamenti in regime variabile di prezzi del mercato.

La “filosofia” della programmazione dinamica è pure applicabile ad una più vasta gamma di disparate situazioni, ad esempio: dimensionamento e ubicazione di impianti o di magazzini, calcolo del prodotto di una catena di matrici, riconoscimento del linguaggio generato da una grammatica libera (si veda il terzo capitolo), gestione di tabelle ausiliarie dei “motori scacchistici”, confronto fra sequenze di simboli (allineamento di stringhe, sottosequenza – comune o crescente – più lunga, distanza tra due sequenze o *edit distance*: sono problemi che si ritrovano in bioinformatica, ad esempio per l’analisi strutturale dell’acido ribonucleico).

Nel quarto capitolo discuteremo di un paio di algoritmi di programmazione dinamica, a proposito dei problemi del *commesso viaggiatore* e dello *zaino*.

Principalmente a Bellman si deve anche un algoritmo per il calcolo dei cammini minimi a partire da un nodo. Questo algoritmo processa ripetutamente gli archi, anziché i nodi come fa quello di Dijkstra, e accetta pure archi (ma non cicli) con pesi negativi; una versione “distribuita” è oggi impiegata in alcuni protocolli di *routing*.

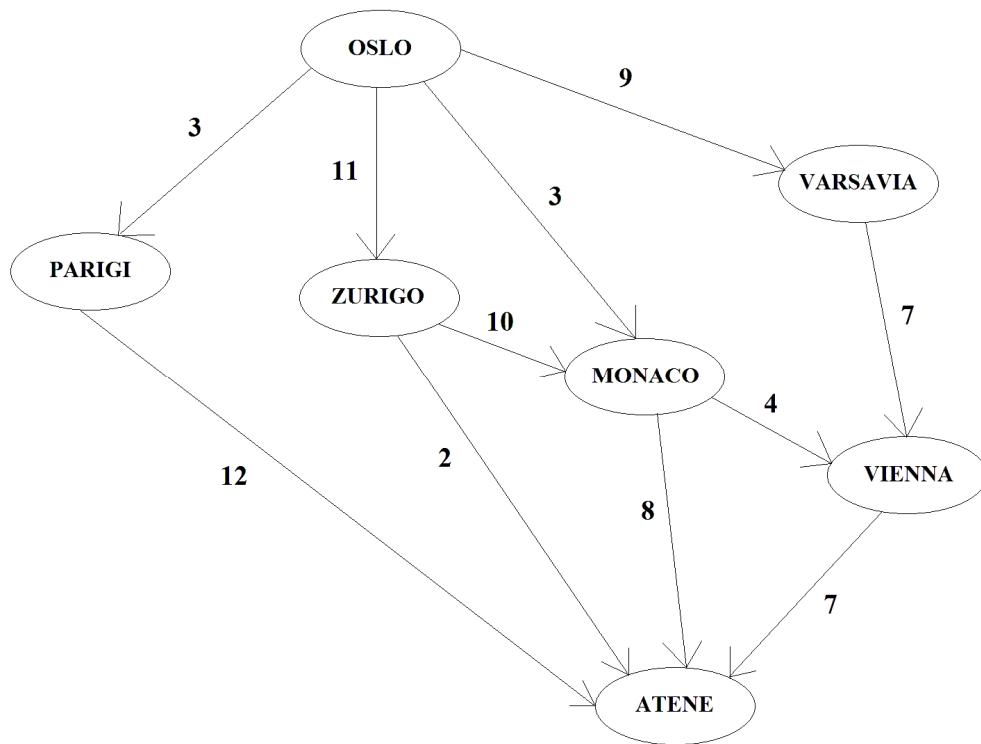
## Una vacanza in Grecia.

Un gruppo di venti turisti norvegesi ha deciso all’ultimo minuto di trascorrere le vacanze di Pasqua in Grecia. Sebbene essi non abbiano prenotato alcun volo, arrivano fiduciosi all’aeroporto di Oslo con l’intenzione di raggiungere Atene in serata. Sono però disponibili soltanto i posti sui voli indicati nella figura alla pagina successiva.

È davvero un bel problema! È evidente che se, ad esempio, arrivassero a Vienna in più di sette (provenienti alcuni da Varsavia, altri da Monaco) non tutti potrebbero

giungere ad Atene per la sera (dato che sul volo da Vienna ad Atene sono soltanto sette i posti disponibili)...

- Come possono farcela, in modo tale che nessuno di loro sia costretto a prendere quattro voli (Oslo-Zurigo-Monaco-Vienna-Atene)?
- Se si aggiungesse un altro turista, potrebbero farcela ugualmente a raggiungere tutti quanti Atene in serata?



**Soluzioni.** Si suddividono in quattro comitive:

- 3 seguono il percorso Oslo-Parigi-Atene,
- 2 seguono il percorso Oslo-Zurigo-Atene,
- 8 seguono il percorso Oslo-Zurigo-Monaco-Atene,
- 7 seguono il percorso Oslo-Varsavia-Vienna-Atene.

Non è l'unica possibilità. Quattro potrebbero separarsi dall'ultima comitiva: tre di essi seguirebbero il percorso Oslo-Monaco-Vienna-Atene, però il quarto sarebbe costretto a prendere quattro voli! Ci sono ulteriori possibilità?

Se all'ultimo momento si aggiungesse un altro turista, allora non sarebbe più possibile accontentare tutti, neanche se qualcuno di essi si adattasse a prendere quattro voli.

Si tratta, in effetti, di un *problema di flusso massimo*: se tracciamo una linea (*taglio*) che lasci al di sotto gli scali di Parigi e Atene, il flusso massimo che la può attraversare è dato da  $3 + 2 + 8 + 7 = 20$ , per cui è sicuro che più di 20 passeggeri non possano giungere ad Atene per la sera.

Anche questo problema, formulato nel 1954 a proposito del traffico ferroviario sovietico, è tra i numerosi che furono studiati a iniziare dalla metà degli anni '50 del secolo scorso. Sebbene possa riguardarsi come problema di *programmazione lineare* (vi accenneremo brevemente più avanti), per esso sono stati sviluppati algoritmi *ad hoc*: Lester R. Ford e Delbert Ray Fulkerson idearono il primo nel 1957, in seguito migliorato; in particolare, all'inizio degli anni '70, Jack R. Edmonds e Richard M. Karp – e, indipendentemente e ancor prima, l'allora sovietico Yefim Dinitz (E. A. Dinic) – proposero una più efficiente esplorazione dei percorsi, secondo una ricerca in ampiezza a partire dalla “sorgente” della rete. Il più recente algoritmo risolutivo (James B. Orlin, 2013) richiede un tempo di ordine dato dal prodotto tra il numero di nodi  $n$  e il numero di archi  $m$  del grafo (o anche inferiore, sotto ipotesi usualmente verificate), mentre per quello di Edmonds-Karp (o Dinitz) l'ordine è  $n \cdot m^2$  (o  $n^2 \cdot m$ ). Le tipiche applicazioni riguardano problemi di massimo soddisfacimento di richieste di una determinata merce, disponibile in un punto di produzione, o problemi di regolazione del traffico.

In generale, il grafo in esame è una *rete di trasporto* (o *di flusso*), cioè un grafo orientato, connesso e privo di *cappi* (archi che partono da un nodo e arrivano al nodo stesso), con un unico nodo *sorgente* (senza archi entranti, nell'esempio Oslo) e un unico nodo *pozzo* (senza archi uscenti, nell'esempio Atene); ad ogni arco è associato un peso (positivo) che esprime la sua *capacità* (massima) in termini di portata.

Un *taglio* è caratterizzato dalla ripartizione dei nodi in due sottoinsiemi, A e B; la sorgente appartiene ad A, il pozzo a B. Quindi, se  $n$  sono i nodi della rete, sorgente e pozzo compresi, i possibili tagli sono  $2$  elevato alla potenza  $n - 2$ . La *capacità di un taglio* è la somma delle capacità di tutti gli archi che partono da un nodo di A e giungono a un nodo di B. Dalla sorgente può uscire un *flusso* che si propaga attraverso la rete, per poi giungere al pozzo, purché per ogni nodo intermedio il flusso entrante (somma delle quantità di flusso sugli archi che arrivano) sia uguale a quello uscente (somma delle quantità di flusso sugli archi che si dipartono), e di nessun arco sia superata la capacità. Si dimostra che il valore di un qualsiasi flusso uscente dalla sorgente non può superare la capacità di un qualsiasi taglio; inoltre, il massimo valore di flusso è pari alla minima tra le capacità di tutti i tagli.

### ***Compilation.***

Aldo vuole ripartire le registrazioni di dodici brani musicali, di varie durate, su tre supporti di memoria, in modo tale che siano occupati il più possibile equamente. Le durate, espresse in minuti, sono:

6, 11, 12, 15, 17, 18, 18, 21, 25, 27, 31, 33.

Come può ripartirle?

**Soluzione.** Siccome la somma di tutte le durate è di 234 minuti, l'ideale sarebbe ripartirle in tre gruppi di 78 minuti ciascuno. In effetti è possibile, addirittura in tre diversi modi:

- a) 1.  $33 + 27 + 18 = 78$   
     2.  $31 + 18 + 17 + 12 = 78$   
     3.  $25 + 21 + 15 + 11 + 6 = 78$
- b) 1.  $33 + 27 + 18 = 78$   
     2.  $31 + 21 + 15 + 11 = 78$   
     3.  $25 + 18 + 17 + 12 + 6 = 78$
- c) 1.  $33 + 27 + 12 + 6 = 78$   
     2.  $31 + 21 + 15 + 11 = 78$   
     3.  $25 + 18 + 18 + 17 = 78$

E poiché nella raccolta ci sono due brani di uguale durata, 18 minuti, che soltanto nell'ultimo caso si trovano insieme, in realtà Aldo dispone di ben cinque diverse *combinazioni* che ripartiscono i brani musicali in modo ottimo sui tre supporti di memoria.

Il quesito proposto è un'istanza del *problema dello scheduling deterministico*, di 12 processi indipendenti, su 3 processori aventi le stesse caratteristiche.

Con *scheduling* si intende l'ordinamento dei processi che devono essere eseguiti e, se il sistema di elaborazione è multiprocessore, la loro assegnazione ai processori che dovranno eseguirli; “deterministico” significa che si conosce il numero dei processi, e di ognuno la durata esatta (il tempo impiegato da uno dei processori, identici, per eseguirlo), e altri non se ne aggiungeranno in corso di elaborazione. I processi sono indipendenti quando non è stabilita tra essi alcuna relazione di precedenza.

Nel caso di sistema multiprocessore, e nell'ipotesi di non poter interrompere l'esecuzione di un processo una volta iniziata, l'obiettivo dello *scheduling* è appunto la minimizzazione dell'intervallo di tempo T tra l'inizio dell'elaborazione e la terminazione dell'ultimo processo in esecuzione.

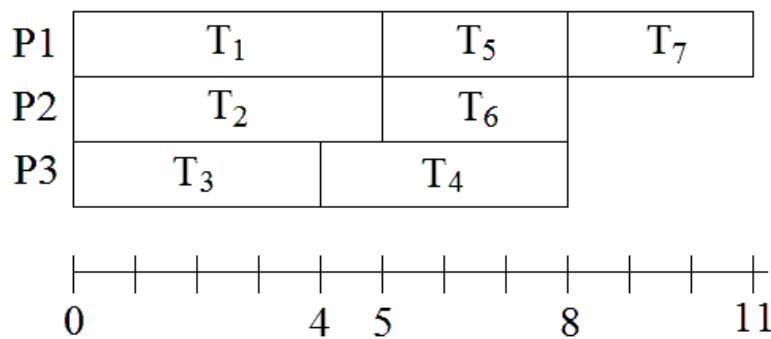
In generale, se si vuole risolvere all'ottimo, si tratta attualmente di un problema assai più oneroso, in termini di passi di calcolo, di tutti quelli incontrati nei precedenti paragrafi! È classificato infatti tra i problemi *NP-ardui*, di cui parleremo nel quarto capitolo, ma è stato anche oggetto del primo articolo sugli *algoritmi di approssimazione* (Ronald L. Graham, *Bounds for Certain Multiprocessing Anomalies*, The Bell System Technical Journal, Vol. 45, No. 9, novembre 1966, [http://www.math.ucsd.edu/~ronspubs/66\\_04\\_multiprocessing.pdf](http://www.math.ucsd.edu/~ronspubs/66_04_multiprocessing.pdf)).

In effetti, esiste un (buon) algoritmo sub-ottimo che consiste semplicemente nell'ordinamento dei processi secondo il loro tempo di esecuzione e quindi, ogniqualvolta si liberi un processore, nell'assegnazione ad esso del processo, non ancora assegnato, che richiede il maggior tempo di esecuzione. Questa regola (euristica) è nota con l'acronimo LPTF (*Longest-Processing-Time First*): dà infatti la precedenza al processo col tempo di elaborazione più lungo.

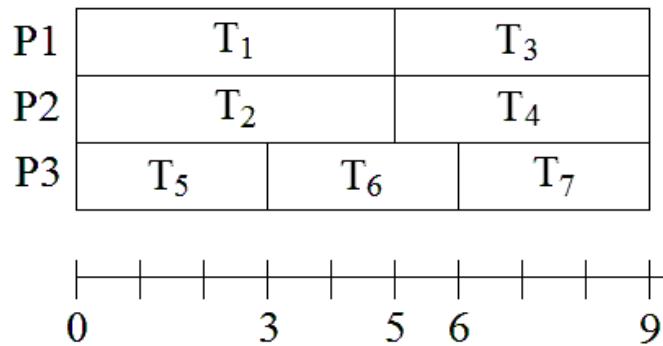
Quanto sarà buona? Un algoritmo di approssimazione deve fornire un'indicazione sulla bontà della soluzione trovata, ad esempio quanto sia lontana, nel peggioro dei casi, dalla soluzione ottima, che rimane sconosciuta. Per il problema in esame, l'algoritmo LPTF garantisce un tempo  $T$  che, al massimo, sarà dato dal tempo ottimo  $T_{min}$  moltiplicato per il fattore  $4/3 - 1/(3m)$ , dove  $m$  è il numero dei processori. Ad esempio, se i processori sono 3, tale fattore è  $11/9$ : ciò vuol dire che, nel peggioro dei casi, l'algoritmo LPTF porterà a un aumento di tempo pari a circa il 22% del tempo ottimo.<sup>2</sup>

È stato infatti dimostrato che la circostanza più sfavorevole per l'algoritmo LPTF si ha quando i processi sono  $2m + 1$ , con i tempi di esecuzione dati rispettivamente da  $2m - (i + 1)/2$  (qui la divisione è intera), con  $i = 1, \dots, 2m$ , e infine ancora un tempo  $m$  per l'ultimo processo.

Ad esempio, sempre con  $m = 3$ , vi siano 7 processi con tempi  $T_i = 5, 5, 4, 4, 3, 3, 3$ ; allora l'algoritmo LPTF porta al seguente *diagramma di Gantt*:



mentre la soluzione ottima è la seguente:




---

<sup>2</sup> Esistono altre euristiche, più sofisticate, con fattori migliori. Si noti poi che nel caso banale  $m = 1$ , si ottiene comunque il tempo ottimo.

Tuttavia, nel caso di sistema monoprocesso, è ovvio che l'obiettivo debba essere un altro. Può aver senso minimizzare il tempo medio di permanenza di un processo nel sistema, per cui la politica da adottare, per avere questa volta la certezza di ottenere l'ottimo, è esattamente opposta: i processi devono essere eseguiti in ordine *crescente* di tempo di elaborazione!

Si osservi che tale obiettivo è del tutto ragionevole anche nel caso di sistema multiprocesso; la regola SPTF (dove S sta per *Shortest*) dà ancora una soluzione ottima.

Nel caso della *compilation* di Aldo, l'algoritmo LPTF propone una soluzione in cui il “più lungo” dei tre gruppi ha una durata complessiva di 80 minuti (e gli altri due di 79 e 75 minuti): nemmeno il 3% in più del tempo ottimo.

Sono state studiate molte varianti di questo problema. In particolare, quando i processi non sono indipendenti, ma tra (alcuni) di essi è stabilita una *relazione di precedenza*, esprimibile con un grafo orientato aciclico (non è detto connesso): se un processo deve essere eseguito prima di un altro, questo fatto sarà rappresentato con un arco diretto dal primo al secondo; ovviamente, non deve nascere alcun ciclo, altrimenti nessun processo del ciclo potrebbe essere eseguito prima degli altri!

Esiste un algoritmo efficiente che porta all’ottimo quando tutti i processi hanno la stessa durata e i processori (uguali) sono soltanto due; esso è basato su un particolare ordinamento topologico del grafo: il lettore interessato potrà reperirlo facilmente.

Se i processi hanno tutti la stessa durata e il grafo delle precedenze è un *albero* (eventualmente, quando vi si invertono tutti gli archi), allora – anche con più di due processori – vi è un modo efficiente, e molto semplice, per giungere alla soluzione ottima: ogniqualvolta si liberi un processore, gli si assegna uno dei (sotto)processi tra quelli, non ancora assegnati e i cui predecessori siano stati tutti terminati, che abbiano il più alto livello (di precedenza) secondo l’albero.

Nella gara *Kangourou dell’Informatica* di marzo 2010 fu proposto un quesito assai interessante, dal titolo “Lavori di manutenzione”, dove i “processori” erano tre: si trattava di tre ragazzi che dovevano dividersi certi compiti, sì da completare l’intero carico di lavoro nel più breve lasso di tempo. Purtroppo, tali compiti non avevano la stessa durata, né le precedenze erano espresse da un albero...

### **La ruota della fortuna.**

Aldo vuole allenarsi a dosare con precisione la forza necessaria a far girare la “ruota della fortuna” di un dato numero di settori. Inizia pertanto a far avanzare la ruota di un posto, dal settore 1 al settore 2; poi di due, dal settore 2 al settore 4; quindi di tre, dal 4 al 7; e così via, sempre aumentando di uno il numero di posti da saltare.

La ruota usata da Aldo per queste sue prove ha 11 settori, sicché i settori che via via “escono” sono:

$$2, 4, 7, 11, 5, 11, 7, 4, 2, 1, 1,$$

dopodiché questa sequenza si ripeterà per sempre!

Aldo nota dunque che gli altri settori (3, 6, 8, 9 e 10) non usciranno mai.

Quale caratteristica deve avere il numero di settori della ruota, per far sì che, prima o poi, escano *tutti*?

**Soluzione.** Deve essere una potenza di 2. In effetti, il caso della “ruota della fortuna” di Aldo è tra quelli... sfortunati! Quando il numero dei settori è un numero primo  $p$  maggiore di 2, allora ne sono toccati soltanto  $(p+1)/2$ . Tuttavia, in altri casi (quali?) la frazione di settori raggiunti è ancor minore: provate, ad esempio, con 9 o con 15. (Si veda anche il quesito “Nastro trasportatore”, nel libretto del 2014.)

Supponiamo di voler memorizzare delle *chiavi* (costituite da sequenze di caratteri qualsiasi, anche piuttosto lunghe), man mano che si presentano, allo scopo di sapere rapidamente, tutte le volte che occorrerà, se una data chiave si è già presentata oppure no, aggiungendola a sua volta in memoria in quest'ultimo caso.

Se sappiamo quante chiavi al massimo dovremo trattare, possiamo costruire una tabella di  $N$  locazioni, con  $N$  sufficiente a contenerle tutte (e magari un po' sovrabbondante). Quando una chiave si presenta, le applichiamo una particolare funzione (detta di *hash*), che la "trituri" in maniera opportuna e ne ricavi un numero compreso tra 1 e  $N$ : questo numero ci indicherà la locazione in cui memorizzarla. Per vedere se c'è già, applicheremo lo stesso procedimento.

Non sorgerebbe alcun problema se la funzione di *hash* fosse *iniettiva*, ossia non associasse mai a chiavi diverse una stessa locazione: ma questa, chiaramente, non è un'ipotesi realistica, poiché la tabella dovrebbe avere almeno tante locazioni quante sono *tutte* le chiavi che potrebbero in teoria presentarsi.

Quando la locazione indicata dalla funzione di *hash* è già occupata da una chiave diversa da quella che si è ora presentata, dobbiamo continuare la ricerca... ad esempio, nella locazione successiva, ed eventualmente (saltandone una) nella quarta (rispetto a quella iniziale), poi (saltandone due) nella settima, e così via (considerando la tabella "circolare"), fino ad arrivare a una locazione "vuota" (concludendo che la chiave corrente non si era ancora presentata, e lì allora può trovare posto) oppure a una locazione che contenga la chiave corrente (concludendo che quest'ultima si era già presentata).

Procedere con questo criterio (detto *rehashing quadratico*), anziché andando avanti di una sola locazione per volta (secondo il più ovvio *rehashing lineare* con passo unitario), ha un vantaggio: evita l'accumulo delle chiavi in locazioni adiacenti, derivante dalla coincidenza di sottosequenze che pur iniziano da due differenti locazioni vicine.

Per garantire allora che, nel caso peggiore, tutte le locazioni siano raggiungibili, il numero  $N$  deve essere appunto una potenza di 2.

Accenniamo soltanto al fatto che vi sono soluzioni migliori: ad esempio, il metodo del *quoziente quadratico*, oppure il *rehashing casuale*, per "spargere"<sup>3</sup> le chiavi all'interno della tabella senza seguire alcun ordine particolare, ma in modo il più possibile uniforme. Ovviamente, le sequenze "casuali" devono essere facilmente ricostruibili per l'operazione di ricerca.

Il problema si complica quando l'operazione di ricerca, nel caso in cui abbia successo, possa preludere a un'operazione di cancellazione della chiave trovata dalla tabella (l'arrivo in una locazione "cancellata" non dovrà infatti interrompere alcuna ricerca successiva); oppure quando non si riesca a fissare un  $N$  ragionevole...

---

<sup>3</sup> Il verbo inglese *to hash* significa sia "triturare" sia "fare confusione": la distribuzione delle chiavi nella tabella deve apparire il più possibile casuale, e nel contempo le chiavi non sono mantenute in alcun ordine particolare, per cui è difficile riordinarle o semplicemente accedervi in un certo ordine!

## Successioni “autosomiglianti”.

In due edizioni del *Kangourou dell'Informatica* (la finale di maggio 2010 e la gara di marzo 2011) sono stati proposti alcuni quesiti riguardanti successioni numeriche in qualche modo “autosomiglianti”. Riprendiamo qui il discorso, passando poi a considerare altre successioni di questo genere, assai interessanti e piuttosto note.

**1.** Si parte con 0, dopodiché, a ciascuna iterazione successiva, si sostituisce ogni 0 con 01 e ogni 1 con 0. (Si tratta della stessa successione capitata nella finale del 2010, con 0 e 1 scambiati.)

Numeriamo le righe iniziando da 0 e, via via, scriviamo il risultato della  $i$ -esima iterazione sulla riga  $i$ . Questi sono i risultati delle prime sette iterazioni:

riga 0	0
riga 1	01
riga 2	010
riga 3	01001
riga 4	01001010
riga 5	0100101001001
riga 6	010010100100101001010
riga 7	01001010010010100101001001

Si noti che, ad ogni iterazione, si concatena una sequenza di bit a quella già scritta sulla riga precedente. Per ottenere la successione completa, si dovrebbe proseguire all’infinito.

- Calcoliamo il numero di 0, il numero di 1 e il numero complessivo di cifre (binarie) in ciascuna riga: quale famosa successione seguono questi numeri?
- Se (da una riga sufficientemente lunga) si cancellano le cifre 0 una sì e una no, e così pure le cifre 1 una sì e una no, quale sequenza di cifre si ottiene?

**Soluzioni.** Probabilmente il lettore già conosce la cosiddetta *successione di Fibonacci* ( $F_0 = 0$ ,  $F_1 = 1$ ,  $F_{n+2} = F_{n+1} + F_n$  per ogni naturale  $n$ ), che compare per la prima volta nel *Liber Abaci*, scritto nel 1202 da Leonardo Pisano, meglio noto come Fibonacci, al quale soprattutto si deve l’ampia diffusione delle cifre indo-arabe (zero incluso) in Europa. Già l’*incipit* di quest’opera fondamentale merita una volgata citazione: «Le nove cifre indiane sono: 9, 8, 7, 6, 5, 4, 3, 2, 1. Con queste nove cifre, e con il segno 0, che gli Arabi chiamano *zefiro*, si scrive qualsiasi numero.»

Tornando alle sequenze di bit che si trovano sulle righe scritte qui sopra, si ha che, sulla riga  $n$ , il numero di 1 è proprio  $F_n$ , il numero di 0 è  $F_{n+1}$  e quindi il numero complessivo di bit è  $F_{n+2}$ .

Per quanto concerne la seconda domanda, si ottiene la prima parte della sequenza stessa. Se immaginiamo di fare le suddette cancellazioni sull’intera successione (infinita), si ottiene la successione stessa: e così è giustificata l’autosomiglianza!

**2.** Come al punto precedente, con una sola variazione: ogni 1 è sostituito con 10. Questi sono i risultati delle prime quattro iterazioni:

riga 0	0
riga 1	01
riga 2	0110
riga 3	01101001
riga 4	0110100110010110

Pure qui, ad ogni iterazione, la sequenza già scritta non cambia!

Immaginiamo di continuare con successive iterazioni...

- Quanti bit contiene la riga  $n$ ?
- È vero che, in ciascuna riga dalla 1 in poi, il numero di cifre 0 è uguale al numero di cifre 1?
- Ciascuna riga dalla 1 in poi può essere ottenuta dalla precedente facendola seguire da... che cosa?
- Quali righe sono *palindrome* (cioè non cambiano se le rovesciamo)?
- Se prendiamo una cifra sì e una cifra no in una riga qualsiasi dalla 1 in poi, che cosa otteniamo?
- Ci saranno mai più di due cifre 0 o più di due cifre 1 consecutive?

**Soluzioni.** La riga  $n$  contiene  $2^n$  bit. Ciascuna riga dalla 1 in poi contiene un ugual numero di cifre 0 e di cifre 1, e può essere ottenuta dalla precedente facendola seguire da una copia della stessa con i bit “negati”; risultano palindrome tutte le righe di posto pari.

Immaginando di operare sull’intera successione (infinita), se prendiamo una cifra sì e una cifra no otteniamo la successione stessa. E il medesimo risultato si ha quando prendiamo una cifra ogni 4, oppure una ogni 8 eccetera. Non solo: anche quando prendiamo 2 cifre sì e 2 no, oppure 4 sì e 4 no, oppure 8 sì e 8 no...

Non ci saranno mai più di due cifre 0, né più di due cifre 1 consecutive; ma queste non sono le sole interessantissime proprietà di questa successione, che può essere definita, cifra per cifra (la prima in corrispondenza di zero), nel seguente modo:

$$\begin{aligned} t(0) &= 0 \\ t(2n) &= t(n) \\ t(2n+1) &= 1 - t(n) \end{aligned}$$

Data una qualsiasi parte finita di questa successione, chiamiamola  $X$ , si può calcolare di conseguenza una lunghezza  $l$  tale che  $X$  occorra in *ogni* parte di lunghezza  $l$  nella successione. Ciononostante, la successione non è periodica, neanche da un certo punto in poi!

Nella successione si possono trovare molte parti ripetute due volte consecutive, cioè della forma  $XX$ , ma nessuna sequenza della forma  $0X0X0$  o  $1X1X1$ . Inoltre, non vi

si trova mai una ripetizione per *tre* volte consecutive di una *qualsiasi* sequenza di bit! Quest'ultima proprietà fu usata dal matematico olandese Machgielis (Max) Euwe, campione del mondo di scacchi dal 1935 al 1937, per dimostrare l'inefficacia di una regola posta per scongiurare la possibilità di partite senza fine. Tale regola, detta “tedesca”, stabiliva che una partita fosse dichiarata patta quando una stessa sequenza di mosse si ripete consecutivamente per tre volte.

In un suo articolo del 1929, Euwe provò che, sotto questa regola, sono possibili partite a scacchi infinitamente lunghe. Si consideri infatti la nostra successione e, ad esempio, si associ:

ad ogni 0 la sequenza di mosse Cg1-f3, Cg8-f6; Cf3-g1, Cf6-g8  
 ad ogni 1 la sequenza di mosse Cb1-c3, Cb8-c6; Cc3-b1, Cc6-b8.

Il regolamento da tempo in vigore prevede tuttavia delle condizioni di patta che non permettono partite senza fine, anche qualora nessuno dei due contendenti la richieda o l’arbitro si astenga dal decretarla d’ufficio: la partita è infatti patta sia quando si verifica tre volte la stessa *posizione* sulla scacchiera, sia quando da entrambe le parti sono state giocate almeno 50 mosse senza che si siano verificate catture, né movimenti di pedoni. Per inciso, basta una sola delle due condizioni a impedire partite infinitamente lunghe.

La successione in esame ha una lunga storia e molti nomi. Di solito è detta di Prouhet-Thue-Morse (sequenza A010060 in OEIS, *The On-Line Encyclopedia of Integer Sequences*): dapprima implicitamente applicata in teoria dei numeri da Eugène Prouhet nel 1851, fu esplicitata da Axel Thue che la usò nel 1906 per i suoi sistemi grammaticali (vi accenneremo nel terzo capitolo), ma deve la notorietà a Harold C. Marston Morse, che nel 1921 la impiegò in geometria differenziale e poi ne discusse proprietà e applicazioni in un articolo pubblicato insieme con Gustav A. Hedlund nel 1944. Euwe la riscoprì indipendentemente dagli altri studiosi, così come era capitato a Morse. A chi desideri approfondire l’argomento consigliamo l’interessante volume di Jean-Paul Allouche e Jeffrey O. Shallit: *Automatic sequences. Theory, Applications, Generalizations*, Cambridge University Press, 2003.

Suggeriamo inoltre un’idea di sicuro effetto: proseguendo le iterazioni di cui sopra, si arrivi a una riga sufficientemente lunga, la si riporti in una matrice quadrata (a quali righe ci si potrà fermare e di quanti bit dovrà essere il lato?), si associa ad ogni 0 un pixel bianco e ad ogni 1 un pixel nero e si disegni sullo schermo del computer.

Nel sesto capitolo parleremo di *automi cellulari*. La successione di Prouhet-Thue-Morse può essere calcolata anche da un automa cellulare (Stephen Wolfram, *A New Kind of Science*, Wolfram Media, 2002, pp. 83, 890-895, 1091-1092 e 1186).

Se poi programmiamo un automa – come la *tartaruga Logo* – in modo tale che quando riceve 0 si muova in avanti di un’unità e quando riceve 1 ruoti di 60 gradi in senso antiorario, fornendogli in input la successione di Prouhet-Thue-Morse la sua traiettoria convergerà all’*isola* (o *fiocco di neve*) di von Koch, che ritroveremo nel quinto capitolo.

**3.** Scriviamo i numeri naturali in notazione binaria e calcoliamo il numero (chiamiamolo  $n_1$ ) di cifre 1 in ciascuno di essi, come illustrato in tabella:

n1			n1		
0	0	0	8	1000	1
1	1	1	9	1001	2
2	10	1	10	1010	2
3	11	2	11	1011	3
4	100	1	12	1100	2
5	101	2	13	1101	3
6	110	2	14	1110	3
7	111	3	15	1111	4
					...

- Se prendiamo un termine sì e uno no nella successione  $n_1$ , che cosa otteniamo?
- Se, nella successione  $n_1$ , sostituiamo ciascun numero pari con 0 e ciascun numero dispari con 1, che cosa otteniamo?

**Soluzioni.** Riotteniamo la stessa successione, e così di nuovo se prendiamo un termine ogni 4, oppure ogni 8 eccetera.

Per quanto concerne la seconda domanda... si ricava la successione di Prouhet-Thue-Morse! Ecco dunque un modo alternativo per calcolare  $t(n)$ : si esprime  $n$  nella base 2, si calcola la somma delle cifre 1 e se ne considera il resto della divisione per 2. Ne ripareremo all'ultimo capitolo!

**4.** Questa volta si parte con 1, dopodiché, a ciascuna iterazione successiva, si scrive 10 davanti a ognuna delle cifre.

Questi sono i risultati delle prime tre iterazioni:

riga 0	1
riga 1	101
riga 2	101100101
riga 3	101100101101100100101100101

Anche qui, ad ogni iterazione, la sequenza già scritta non cambia.

- Quanti bit contiene la riga  $n$ ? Come sono ripartiti tra 0 e 1?
- Ci saranno mai più di due cifre 0 o più di due cifre 1 consecutive?

**Soluzioni.** La riga  $n$  contiene  $3^n$  bit, che dunque sono in numero dispari; le cifre 1 sono una in più delle cifre 0.

Se si suddividono le sequenze in tre parti, man mano che vengono scritte, è facile accorgersi che la sequenza su ciascuna riga dalla 1 in poi può essere ottenuta scrivendo tre volte la riga precedente e sostituendo l'ultimo bit della parte centrale, che è un 1, con uno 0.

Non vi compariranno mai più di due bit 0 o più di due bit 1 consecutivi.

A questa successione Clifford A. Pickover dedica un capitolo del suo suggestivo libro *Wonders of Numbers. Adventures in Math, Mind, and Meaning* (Oxford University Press, 2000, poi pubblicato in italiano col titolo *Le meraviglie dei numeri*). Là egli la introduce in maniera più divertente, con un gioco di parole di tipo “look and say”. Si parte scrivendo una lettera “a” e chiedendo di proseguire dicendo ciò che si vede scritto sulla riga precedente: “una a”, in inglese “an a”. Quindi sulla riga successiva si scriverà “ana”. Che cosa si è scritto? “An a, an n, an a”, e quindi si continua scrivendo “anaannana”, e così via.

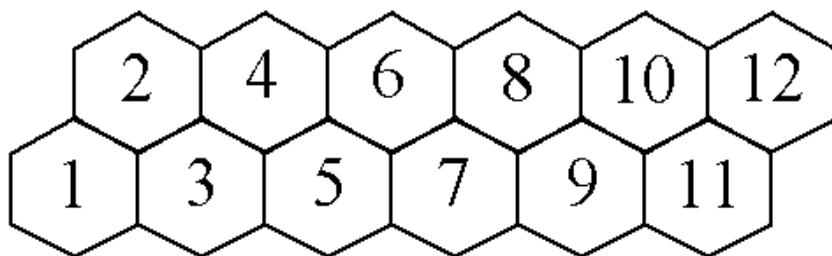
Assai noto è il gioco analogo con le cifre decimali, partendo ancora con una cifra 1 e scrivendo su ciascuna riga (in cifre) ciò che si vede sulla riga precedente:

1	
11	un “uno”
21	due “uno”
1211	un “due”, un “uno”
111221	un “uno”, un “due”, due “uno”
312211	tre “uno”, due “due”, un “uno”
13112221	un “tre”, un “uno”, due “due”, due “uno”

eccetera. Di solito, si proponevano all’interlocutore le prime poche righe e gli si chiedeva come prosegue la serie. Ma una volta svelato l’arcano, altre domande sono più interessanti: comparirà mai la cifra 4? Come aumenta la lunghezza delle righe? Si può ancora, in qualche modo, parlare di “autosomiglianza”?

### I percorsi dell’ape.

Un’ape può spostarsi, lungo un’arnia di due file di celle esagonali, sempre e soltanto in una cella adiacente verso destra, in diagonale o in orizzontale, rispetto a quella che occupa. Ad esempio, quando si trova nella cella 7 (si veda la figura) può spostarsi, a sua scelta, o nella 8 o nella 9; se si trova nella cella 8 può spostarsi o nella 9 o nella 10.



Le domande sono: se parte dalla cella 1, quanti percorsi diversi può fare per raggiungere la cella 7? E per raggiungere la cella 12?

**Soluzioni.** Naturalmente è stato l'accenno ai numeri di Fibonacci, nel paragrafo precedente, a ricordarmi questo famoso, vecchio problema...

Il numero dei percorsi possibili per raggiungere la cella  $n$  è dato da  $F_n$ : quindi sono 13 per raggiungere la cella 7, e 144 per raggiungere la 12.

Una significativa applicazione dei numeri di Fibonacci sarà discussa nell'ultimo capitolo.

### Indovina il numero pensato!

Aldo dispone i numeri da 1 a 16 su due file:

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16

quindi chiede a Beppe di pensare un numero e di indicargli la fila in cui si trova. Beppe, dopo aver pensato un numero tra questi, gli dice che sta nella fila in basso. Allora Aldo, partendo da sinistra, prende un numero dalla fila in basso e uno da quella in alto, in modo alterno, disponendoli su due nuove file:

9	1	10	2	11	3	12	4
13	5	14	6	15	7	16	8

Aldo chiede di nuovo in quale fila si trovi il numero pensato e Beppe risponde sempre in quella in basso. Aldo ripete la stessa procedura:

13	9	5	1	14	10	6	2
15	11	7	3	16	12	8	4

La terza volta che Aldo pone la domanda, Beppe risponde che ora sta nella fila in alto; allora Aldo prende ancora i numeri in modo alterno, iniziando questa volta dalla fila in alto:

13	15	9	11	5	7	1	3
14	16	10	12	6	8	2	4

All'ultima richiesta di Aldo, Beppe risponde che adesso si trova nella fila in basso. Qual è il numero pensato da Beppe?

**Soluzione.** Beppe ha pensato il numero 14, che si trova al *primo posto* nella fila indicata con la sua ultima risposta. Il procedimento è spiegato (invero un po' confusamente) nel Capitolo LXIX della raccolta di "giochi" *De viribus quantitatis*, scritta dal grande matematico rinascimentale Luca Pacioli, negli anni intorno al 1500.

Seguiamo gli spostamenti dei numeri nell'esempio. Dopo il primo passo, i numeri che erano in seconda fila (quella indicata da Beppe) si sono distribuiti nelle colonne di posto dispari; le altre quattro colonne (quindi la metà dei numeri) sono da

scartare. Dopo il secondo passo, i numeri ancora “buoni” (dal 13 al 16) finiscono nella prima e nella quarta colonna (e un’altra metà di numeri è eliminata). All’ultimo passo rimane soltanto la prima colonna.

È facile generalizzare al caso di  $2^n$  numeri, con  $n$  successive indicazioni di riga.

Per confondere un po’ le idee al nostro interlocutore, ché altrimenti capisce subito il “trucco”, suggeriamo di usare 16 carte da gioco qualsiasi (tutte diverse) e – stando attenti a non sbagliare – alternare qualche variante, ad esempio quella che ora spiegheremo (continuando a usare i numeri da 1 a 16, per semplicità).

La procedura da seguire può essere così descritta: ricevuta l’indicazione di una delle due file, Aldo prende dapprima tutta l’altra fila, un numero dopo l’altro partendo da destra, e dispone i numeri sulle due nuove file, alternandoli uno in alto e uno in basso; poi compie la stessa operazione con la fila indicata da Beppe, quella che contiene il numero da lui pensato.

Riprendendo la disposizione originaria, supponiamo che Beppe pensi il numero 12, e quindi indichi la fila in basso. Allora Aldo disporrà i numeri nel modo seguente:

$$\begin{array}{cccccccc} 8 & 6 & 4 & 2 & 16 & 14 & 12 & 10 \\ 7 & 5 & 3 & 1 & 15 & 13 & 11 & 9 \end{array}$$

Ora Beppe indica la fila in alto e quindi, dopo che Aldo ha ripetuto la procedura, la situazione è questa:

$$\begin{array}{cccccccc} 9 & 13 & 1 & 5 & 10 & 14 & 2 & 6 \\ 11 & 15 & 3 & 7 & 12 & 16 & 4 & 8 \end{array}$$

Beppe indica la fila in basso e l’ultima manovra di Aldo porta a:

$$\begin{array}{cccccccc} 6 & 14 & 5 & 13 & 8 & 16 & 7 & 15 \\ 2 & 10 & 1 & 9 & 4 & 12 & 3 & 11 \end{array}$$

Infine, Beppe indica la fila in basso: il numero da lui pensato è quello che vi occupa il *sesto* posto. Se ora avesse indicato la fila in alto, il numero pensato sarebbe stato il 16.

La regola del sesto posto nell’ultima fila indicata vale però soltanto con 16 numeri o carte che siano. Generalizzando, dopo il primo passo il numero pensato sta nella seconda metà delle colonne, dopo il secondo passo sta nella prima metà della seconda metà, dopo il terzo passo sta nella seconda metà della prima metà della seconda metà, e così via. Giocando con  $2^n$  numeri, dopo  $n$  successive indicazioni di riga, il posto occupato dal numero pensato sarà quello di ordine  $p(n)$ , essendo:

$$p(1) = 1, \quad p(2k) = 2 \cdot p(2k-1), \quad p(2k+1) = 2 \cdot p(2k) - 1.$$

Questa successione è calcolata partendo da 1 e poi, in modo alterno, moltiplicando per 2 oppure moltiplicando per 2 e sottraendo 1: 1, 2, 3, 6, 11, 22, 43 ...

Procedendo sulla stessa falsariga, si possono trovare altre varianti!

## Pile di carte.

Concludiamo la rassegna con un sorprendente giochetto da proporre con le carte, che certamente sarà stato ben noto al protagonista del prossimo capitolo...

Da un mazzo di carte scegliamone 14 e disponiamole una sotto l'altra, a formare una pila, nell'ordine che più ci piace.

Poi ripetiamo per quattro volte queste azioni:

1. dividiamo la pila di carte a metà, chiamiamo A la metà inferiore e B quella superiore;
  2. prendiamo alternativamente una carta da A e una da B, finché ve ne sono, formando una nuova pila.
- Che cosa otteniamo alla fine?
  - E se rifacessimo il gioco daccapo, scegliendo questa volta 16 carte dal mazzo?
  - Quante carte dovremmo prendere dal mazzo per avere gli stessi effetti dopo *cinque* ripetizioni (anziché quattro) dei passi sopra descritti?

**Soluzioni.** Con 14 carte, avremo infine la sequenza nello stesso ordine iniziale!

Con due carte in più, otterremo invece l'ordine inverso rispetto a quello iniziale.

Per giungere agli stessi risultati dopo cinque iterazioni, dovremmo prendere 30 o 32 carte, rispettivamente.

Possiamo ancora fare una piccola riflessione sul legame tra questo gioco, con 16 carte (o altra potenza di 2), e quello di Pacioli. Indichiamo con i numeri da 1 a 16 le carte nella pila iniziale, a partire dalla carta alla sommità. Quando suddividiamo la pila in due metà, chiamiamo quindi B la parte corrispondente alla fila in alto, e A quella che corrisponde alla fila in basso nel gioco di Pacioli.

Se il nostro interlocutore scegliesse il numero 16, ci indicherebbe quattro volte la fila in basso, e se ripetessimo la procedura di riordino una quarta volta, allora i numeri si disporrebbero proprio nell'ordine inverso rispetto a quello iniziale!

E se invece scegliesse il numero 1?

**Parole chiave:** grafi orientati (con, al più, un arco, per ogni coppia ordinata di nodi) pesati e non pesati, matrici di adiacenza e dei costi, cammini di costo minimo, algoritmo di Dijkstra, visite in profondità e in ampiezza, grafo aciclico, programmazione dinamica, rete di trasporto, flusso massimo, *scheduling* deterministico, algoritmo di approssimazione, funzione di *hash*, *rehashing quadratiko*, successione di Fibonacci, successione di Prouhet-Thue-Morse, sequenze “look and say”, numerazione binaria.

## 2. Omaggio a Smullyan

L’informatica è collegata strettamente alla logica; anzi, la logica, con la matematica e l’analisi dei processi cognitivi, è una delle discipline in cui l’informatica affonda le proprie radici. In questa sede, non possiamo pretendere di affrontare un tema così vasto e complesso come i rapporti tra logica e informatica nella maniera che sarebbe più appropriata; tuttavia, attraverso quesiti e racconti che speriamo piacevoli per chi li leggerà, tratteremo aspetti interessanti della questione e giungeremo a riflettere su idee che hanno fatto storia nell’evoluzione del pensiero razionale del secolo scorso.

Nel perseguire questo obiettivo, già di per sé ambizioso, fortunatamente ci possiamo avvalere dell’opera di un grande e singolare personaggio, Raymond Smullyan, che nei suoi libri – invero piuttosto accessibili – è riuscito come mai nessun altro a divulgare, quasi sempre in maniera spassosa, concetti senza dubbio difficili da comprendere, quanto importanti nel nostro campo, pure per le applicazioni.

Da lì prenderemo spunto per ciò che presenteremo in questo capitolo; in particolare, per un approfondimento (notevole) delle problematiche sottese ad un paio di quesiti proposti nelle gare finali *Kangourou dell’Informatica* 2009 e 2011: “La macchina dell’abc” (così intitolato perché nella versione semplificata la macchina manipolava soltanto sequenze di simboli di un alfabeto ternario) e “I programmi RUN”. Per formularli, infatti, avevamo già abbondantemente attinto a due testi di Smullyan: *The Lady or the Tiger? And other logic puzzles including a mathematical novel that features Gödel’s great discovery* e *Satan, Cantor, and Infinity: and other mind-boggling puzzles*, entrambi editi da Knopf (New York, 1982 e 1992) e poi tradotti in italiano con i titoli *Donna o tigre?... e altri indovinelli logici, compreso un racconto matematico sul teorema di Gödel* (Zanichelli, 1985) e *Satana, Cantor e l’infinito e altri inquietanti rompicapi* (Bompiani, 1994).

A onor del vero, un quesito ispirato da Smullyan – che infatti era ricordato nella soluzione come logico di primo piano – compariva già nel libretto della prima gara: “Le gemelle” (marzo 2009). Nelle successive edizioni ne sono stati proposti altri del genere logico-argomentativo, tra i quali: “Carte rosse e carte blu” (marzo 2010), “Quattro proposizioni” (maggio 2010) e gli importanti “Gatti e computer” (marzo 2011), “Mappa colorata” (libretto del 2014), “Quadrati e cerchi” e “Passeggiata nella foresta” (libretto del 2015); prima di procedere oltre, può essere utile rileggerli.

E comunque, per prepararci nel modo migliore a comprendere come funzionano le strane macchine di McCulloch e dei professori Roberts, affrontiamo dapprima qualche nuovo, istruttivo problemino…

### Un’altra torta rubata.

Nella gara di marzo 2009, era stato posto anche un quesito dal titolo “Biancaneve e il nano goloso”, in cui Biancaneve doveva scoprire, col minimo numero di domande, quale dei sette nani avesse mangiato la torta da lei lasciata sul tavolo della cucina. Si trattava di una semplice applicazione del *metodo dicotomico*, e qui ne approfitto per

ricordare che, in generale, la ricerca (binaria) di un elemento in un elenco ordinato di  $N$  elementi (ad accesso diretto, o equivalentemente in un albero binario di ricerca con  $N$  nodi perfettamente bilanciato) richiede di confrontare tale elemento con, al più,  $k$  elementi presenti nell'elenco, dove  $k$  è dato da:

$$\text{floor}(\log_2 N) + 1$$

Si può anche dire da *ceiling* ( $\log_2(N+1)$ ), in maniera equivalente nel nostro caso.<sup>4</sup> Questo è, in generale, il miglior risultato che si possa ottenere nel peggior dei casi; se poi si fosse certi del successo – com'era in quella circostanza, in cui uno dei nani doveva per forza essere il colpevole, o nei giochi con le carte visti poc'anzi, ma ciò capita piuttosto raramente in informatica! – allora la ricerca richiederebbe, al massimo, un confronto in meno, e cioè  $k = \text{floor}(\log_2 N)$ .

Anche oggi, sul tavolo della cucina, c'era una torta, appena sfornata da Biancaneve, e tutti e sette i nani l'hanno vista... La torta è sparita, e Biancaneve vuole scoprire chi l'abbia presa; addirittura sospetta che questa volta potrebbe essere stato più d'uno di loro. Ella sa bene che, da un po' di tempo, i nani hanno preso questa abitudine: ciascuno di essi risponde con un "sì" o con un "no" alle domande che gli vengono rivolte, tuttavia – a seconda della giornata – alcuni dicono la verità e gli altri invece mentono (invertendo la risposta) a qualsiasi domanda venga loro posta, per la quale abbia senso rispondere con un "sì" o con un "no".

- Biancaneve ci pensa un po' su, e poi decide di rivolgere una stessa domanda a ciascuno dei nani, che le permetta di individuare il colpevole o i colpevoli. Quale domanda può formulare?
- Se Biancaneve rivolgesse a Cucciolo la domanda: « Se io ti chiedessi "La torta era sul tavolo della cucina?" la tua risposta sarebbe diversa da quella che mi darai adesso? », quali possibili risposte potrebbe ottenere? (Ma... Cucciolo non era muto?! Però potrebbe pur sempre rispondere con un cenno del capo!)

**Soluzioni.** Biancaneve può porre a ciascuno dei nani questa domanda: « Se io ti chiedessi "Hai rubato la torta?", che cosa mi risponderesti? ». Chi ha rubato risponderà comunque "sì", mentre chi non ha rubato risponderà comunque "no"! Se Biancaneve ponesse a Cucciolo la domanda riportata nel testo, non potrebbe comunque ottenere alcuna risposta! Sia che Cucciolo oggi abbia deciso di dire il vero sia – viceversa – di dire il falso, non può rispondere, né con un "sì" né con un "no", senza contraddirsi la propria decisione!

### A proposito di nani sinceri e nani bugiardi...

Abbiamo detto che, a seconda della giornata, alcuni dei sette nani dicono la verità e gli altri invece mentono.

---

<sup>4</sup> Le funzioni *floor* ("pavimento") e *ceiling* ("soffitto") danno il massimo intero minore o uguale all'argomento e il minimo intero maggiore o uguale all'argomento, rispettivamente.

- Un giorno, Brontolo, Cucciolo e Dotto aiutano Biancaneve a preparare il pranzo. «Oggi siamo tutti e tre bugiardi» dice Dotto, ma Brontolo lo smentisce affermando: « Soltanto uno di noi tre oggi è sincero ». Com'è in realtà oggi ciascuno di questi tre nani?
- Il giorno dopo, Dotto afferma « Oggi siamo tutti e tre sinceri » e Brontolo lo smentisce con la stessa frase del giorno precedente! Come sono Cucciolo e Dotto?
- Il terzo giorno, Dotto afferma «Soltanto uno di noi tre oggi è bugiardo» e, per la terza volta, Brontolo lo smentisce con la sua solita frase! Come sono Cucciolo e Dotto?
- Il quarto giorno, stanco di essere sempre smentito da Brontolo, Dotto lo accusa «Brontolo oggi è bugiardo» e Brontolo questa volta ribatte: «Cucciolo e Dotto oggi sono dello stesso genere». Com'è Cucciolo?
- Il quinto giorno, ci sono tutti e sette ad aiutare Biancaneve.  
Brontolo: «Oggi soltanto uno di noi è bugiardo!»  
Dotto: «Oggi esattamente due di noi sono bugiardi!»  
Eolo: «Oggi esattamente tre di noi sono bugiardi!»  
Gongolo: «Oggi esattamente quattro di noi sono bugiardi!»  
Mammolo: «Oggi esattamente cinque di noi sono bugiardi!»  
Pisolo: «Oggi esattamente sei di noi sono bugiardi!»  
Cucciolo resta muto. Com'è ciascuno di loro?

**Soluzioni.** Anzitutto è evidente che, in ognuno dei primi tre giorni, le due frasi pronunciate da Dotto e da Brontolo non possono essere entrambe vere, cioè sono in *contraddizione*.

*Primo giorno:* Brontolo è sincero, mentre Cucciolo e Dotto sono bugiardi.

Dotto è certamente bugiardo; infatti non può essere sincero comprendendo sé stesso tra i bugiardi. Quindi, Dotto compreso, i bugiardi devono essere o uno o due. Se Brontolo mentisse, i bugiardi sarebbero due (lui e Dotto): ma allora direbbe la verità! Pertanto Brontolo deve essere sincero, e i due bugiardi sono necessariamente Dotto e Cucciolo.

*Secondo giorno:* Cucciolo e Dotto sono di nuovo bugiardi, di Brontolo non si può dir nulla.

Dotto non può dire il vero, perché altrimenti anche Brontolo sarebbe sincero; ma, come abbiamo già osservato, la sua affermazione e quella di Dotto non possono essere entrambe vere. Brontolo di nuovo afferma che i bugiardi sono due: se questo è vero, allora i bugiardi sono Dotto e Cucciolo; altrimenti tutti e tre i nani sono bugiardi. Quindi di sicuro Cucciolo è bugiardo.

*Terzo giorno:* Cucciolo e Dotto sono o entrambi bugiardi o entrambi sinceri; nel primo caso di Brontolo non si può dir nulla, nel secondo caso Brontolo deve essere bugiardo.

Si noti che qui le frasi pronunciate da Dotto e da Brontolo non possono che avere valori di verità diversi. Se è vero ciò che dice Dotto, allora Brontolo è bugiardo, e

quindi Cucciolo non è come Dotto, ossia è bugiardo. Se invece quel che dice Dotto è falso, allora Brontolo è sincero, e quindi Cucciolo è come Dotto, ossia è bugiardo.

*Quarto giorno:* Cucciolo è certamente bugiardo, degli altri due si può dire soltanto che uno è sincero e l'altro è bugiardo.

*Quinto giorno:* al più una delle sei affermazioni può essere vera, e quindi i bugiardi sono almeno cinque. Ne segue immediatamente che Brontolo, Dotto, Eolo e Gongolo sono bugiardi per forza.

Se Mammolo è sincero (i bugiardi sono proprio cinque), allora Pisolo è bugiardo e Cucciolo, sebbene taccia, deve essere sincero.

Se invece Mammolo è bugiardo, allora Cucciolo deve essere pure bugiardo, mentre di Pisolo non si può dire nulla: se Pisolo è sincero, allora è proprio lui l'unico sincero, altrimenti tutti e sette i nani sono bugiardi (ammesso che possa darsi questa eventualità).

In una versione classica di questo indovinello, Cucciolo avrebbe esclamato: «Oggi tutti e sette siamo bugiardi!». In tal caso, nessun dubbio: l'unico sincero sarebbe stato Pisolo.

## Sei proposizioni.

Ecco qui di seguito sei proposizioni, ciascuna delle quali parla di qualcun'altra. Quali sono vere?

1. Una delle proposizioni 2 e 3 è vera, l'altra è falsa.
2. Una delle proposizioni 3 e 5 è vera, l'altra è falsa.
3. Una delle proposizioni 4 e 6 è vera, l'altra è falsa.
4. Una delle proposizioni 2 e 6 è vera, l'altra è falsa.
5. Le proposizioni 1 e 4 sono o entrambe vere o entrambe false.
6. Le proposizioni 1 e 5 sono o entrambe vere o entrambe false.

**Soluzione.** Sono vere la 4 e la 6, mentre tutte le altre sono false.

Ci si può arrivare in questo modo. Supponiamo 6 vera; allora i casi sono due:

a) 1 e 5 entrambe vere, ma ora 5 vera implica 4 vera, che implica 2 falsa, che implica 3 vera: e qui si giunge a una contraddizione, perché 4 e 6 sono entrambe vere;

b) 1 e 5 entrambe false, e 5 falsa implica 4 di nuovo vera, che implica 2 falsa, che adesso implica 3 falsa, e tutto è consistente.

D'altra parte, se supponiamo 6 falsa, i casi sono altri due, entrambi inconsistenti:

c) 1 vera e 5 falsa, ma ora 5 falsa implica 4 falsa, che implica 2 falsa, che implica 3 falsa: e a questo punto 1 non sarebbe più vera;

d) 1 falsa e 5 vera, e 5 vera implica 4 di nuovo falsa, che implica 2 falsa, che adesso implica 3 vera, che è in contraddizione con 4 e 6 entrambe false.

## **Ed ecco a voi Ray Smullyan!**

Raymond Merrill Smullyan, detto Ray, è stato un uomo dall'ingegno poliedrico: fu infatti matematico, logico, filosofo, scacchista, scrittore, saggista, prestigiatore e pianista. Nato a Far Rockaway, New York, il 25 maggio 1919, dopo una vita lunga e ricca di soddisfazioni, ci ha lasciati il 6 febbraio 2017.

In gioventù fu combattuto tra la musica e la matematica, disciplina che approfondì da autodidatta, ma in realtà la sua prima carriera fu quella di prestigiatore. Ottenne infine il diploma di Bachelor of Science nel 1955 presso l'Università di Chicago, mentre già insegnava al Dartmouth College nel New Hampshire, e poi il Ph. D. in matematica nel 1959 a Princeton, dove rimase fino al 1961, anno in cui pubblicò l'elegante monografia *Theory of Formal Systems*, esemplare spiegazione – con varianti e miglioramenti – della teoria degli insiemi ricorsivamente enumerabili.<sup>5</sup> In seguito insegnò a New York, in diversi istituti e università, fino al suo incarico all'Indiana University, Bloomington.

Nel 1957, mentre si occupava di scacchi e studiava per il dottorato (in quel periodo fu uno dei più brillanti allievi di Alonzo Church), pubblicò sul *Journal of Symbolic Logic* un articolo, *Languages in which self-reference is possible*, in cui sosteneva che l'*incompletezza* si ritrova in sistemi formali più elementari di quelli considerati dal celebre logico austriaco Kurt Gödel (1906-1978) nel suo fondamentale lavoro datato 1931. In seguito, Smullyan rilevò che almeno una parte dell'ammirazione suscitata dal teorema di Gödel dovrebbe spettare a quello di Tarski sull'*indefinibilità della verità*, secondo il nostro autore ugualmente rivoluzionario dal punto di vista filosofico, oltre che più facile da dimostrare: sotto certe ipotesi, un sistema coerente (o consistente, cioè che non dia luogo a contraddizioni) non può contenere in sé la definizione della propria verità, che va espressa in un opportuno *metalinguaggio*.

Mentre insegnava, Smullyan continuò a scrivere di logica e di fondamenti della matematica, tanto che il suo ruolo è riconosciuto preminente nella storia della logica moderna. Le sue numerose opere accademiche hanno tutte una caratteristica rara e mirabile: affrontano argomenti assai ardui non solo in modo chiaro e comprensibile, ma anche affascinante e piacevole.

Il punto di arrivo delle sue riflessioni sui classici teoremi limitativi della logica matematica, che lo hanno accompagnato lungo il corso della sua lunga vita, è costituito da un saggio abbastanza accessibile: “Gödel's Incompleteness Theorems”, in Lou Goble ed., *The Blackwell Guide to Philosophical Logic* (2001), pp. 72-89, <http://books.google.it/books?id=aaO2f60YAwIC>.

Tuttavia, Smullyan è noto al pubblico più vasto soprattutto come autore di molti, eccezionali libri di logica e matematica ricreative, alcuni dei quali tradotti anche in italiano; tra i più famosi: *Qual è il titolo di questo libro? L'enigma di Dracula e altri*

---

<sup>5</sup> Un insieme si dice *ricorsivamente enumerabile* se esiste un *algoritmo* (non necessariamente sempre terminante) che, prima o poi, risponde “sì” alla domanda “ $x$  (dato) appartiene all'insieme?” tutte le volte che “sì” è la risposta giusta; altrimenti, potrebbe non dare mai risposta.

*indovinelli logici* (Zanichelli, 1981; l'edizione americana, *What Is the Name of This Book? The riddle of Dracula and other logical puzzles*, fu pubblicata da Prentice-Hall nel 1978), che si concludeva con un capitolo dedicato proprio alla sua prima divulgazione del teorema di Gödel, e il già citato *Donna o tigre?* di quattro anni posteriore. Oltre all'altro suo libro che abbiamo menzionato, *Satana, Cantor e l'infinito*, ce n'è ancora uno, tra quelli apparsi in Italia, che è doveroso ricordare: *To Mock a Mockingbird and other logic puzzles* (Knopf, 1985), tradotto in *Fare il verso al pappagallo e altri rompicapi logici* (Bompiani, 1990), introduzione ricreativa – ma come sempre elegante e profonda – alla logica combinatoria di Schönfinkel e Curry (1924-30), che prelude al *lambda-calcolo* di Church (1931-32), prototipico linguaggio funzionale universalmente espressivo: si arguisce che vi siano trattati temi di importanza fondamentale nello sviluppo dell'informatica, e non solo teorica. Smullyan si interessò pure di misticismo religioso e filosofie orientali, specialmente del taoismo e di come questo possa unificarsi con la logica e la matematica; compose problemi di scacchi (fin dalla gioventù, in particolare di analisi retrograda), aveva l'hobby dell'astronomia, e scrisse un'autobiografia intitolata *Some Interesting Memories: A Paradoxical Life*.

In vecchiaia, fu Professore Emerito di Filosofia all'Indiana University; visse al di là del fiume Hudson, tra i monti Catskill, e tornò alla musica – di nuovo una delle sue principali attività! – giungendo a incidere alcune esecuzioni al pianoforte dei suoi pezzi classici preferiti, composti da Bach, Scarlatti, Schubert e altri.

Celebri rimangono i suoi aforismi, ad esempio: « Alcuni sono sempre critici nei confronti delle dichiarazioni ambigue. Io tendo piuttosto ad essere critico verso le affermazioni precise: esse sono le sole che possono essere correttamente bollate come errate. » O le sue divertenti battute, talvolta veri e propri *nonsense*: « La superstizione porta sfortuna. »

Una volta, durante una conferenza, Melvin Fitting – suo allievo e poi collaboratore presso la City University di New York – lo annunciò al pubblico con queste parole: « E ora vi presento il professor Smullyan, che vi dimostrerà che o lui non esiste o che voi non esistete, ma non saprete quale delle due asserzioni è quella vera... »

Uno degli aneddoti che amava raccontare, ricordando il primo appuntamento con Blanche, la donna che diventò sua moglie, riguarda l'insolita richiesta che le rivolse alla fine della serata: « Io farò un'affermazione. Se è vera, allora tu mi farai un autografo, altrimenti no. » Avendo ottenuto il di lei consenso a questa innocente proposta, le disse: « Non mi darai né il tuo autografo, né un bacio. » È facilmente intuibile quale fu l'inevitabile, logica conseguenza...

Prima di passare ai problemi più seri, vorrei concludere queste brevi note biografiche con un ricordo personale. Proprio nel giorno del suo novantesimo compleanno, tenni nella mia scuola un seminario sull'opera di Smullyan e poi gli inviai le *slides* che avevo preparato per l'occasione. Mi rispose che le aveva apprezzate e – non sapendo sfortunatamente leggere l'italiano – pensava di farsele tradurre. Aggiunse che avrebbe avuto piacere di spedirmi una copia del suo ultimo libro, *Rambles Through My Library*, insieme con alcuni DVD di sue sonate al

pianoforte e letture dai suoi testi, e altre cose ancora. Lo ringraziai e gli chiesi la cortesia di non dimenticarsi di autografare il volume. Spedito il pacco, mi avvisò via *e-mail* e mi raccomandò, qualora non fosse arrivato, di comunicargli la data e l'ora esatta in cui *non* era arrivato! Il plico per fortuna mi fu recapitato pochi giorni dopo, e all'interno del libro trovai scritto: «I wish you the best, but I refuse to sign this book!»; naturalmente seguiva la firma.

## Una curiosa macchina numerica.

Norman McCulloch – un ex-compagno di studi a Oxford dell'ispettore Craig di Scotland Yard, personaggio che compare spesso nei racconti-rompicapi di Smullyan – ha costruito una macchina che è in grado di manipolare certe sequenze di cifre. Le cifre che si possono usare sono quelle decimali da 1 a 9, e la macchina può ricevere (cioè *accetta in input*) una qualsiasi sequenza (di lunghezza finita) della forma:  $2s$ ,  $32s$ ,  $332s$ ,  $3332s$ , ... (cioè un numero arbitrario di 3, anche nessuno, seguiti da un 2 e da  $s$ , che sta per una qualsiasi sequenza di cifre finita e *non vuota*). Se si introduce in input una sequenza di cifre che non abbia questa forma, allora la macchina si arresta e *non produce nulla in output*.

La macchina, chiamiamola  $M$ , funziona in base a queste due regole:

- se  $M$  riceve la sequenza  $2x$ , allora produce la sequenza (non vuota)  $x$ ;
- se  $M$  riceve la sequenza  $3x$ , allora emula una copia di sé stessa (che agisce ovviamente secondo le stesse regole) la quale riceve la sequenza  $x$ ; se questa copia produce (prima o poi) la sequenza (non vuota)  $y$ , allora  $M$  produce  $y2y$ .

Ad esempio: 213 produce 13; 222 produce 22; 321 produce 121 (poiché 21 produce 1); 3321 produce 1212121 (poiché 321 produce 121).

*Prima domanda:* esiste una sequenza che produce sé stessa, cioè tale che, quando è introdotta nella macchina  $M$ , da questa esce la sequenza stessa?

*Seconda domanda:* esiste una sequenza  $z$  che produce  $z2z$ ?

*Terza domanda:* che cosa produce in output la sequenza di input 3323?

Possiamo definire in modo rigoroso il comportamento della macchina  $M$  per mezzo di una funzione (pura e totale)  $M$  scritta in linguaggio ML, il capostipite dei moderni linguaggi funzionali. Vogliamo far sì che, ad esempio, dare alla macchina  $M$  la sequenza di cifre 321 equivalga ad applicare la funzione  $M$  alla *lista di interi* [3, 2, 1], notazione che abbrevia  $3 :: (2 :: (1 :: []))$ , dove [] denota la *lista vuota*: essa è infatti la lista che si ottiene dalla lista vuota, inserendo via via in testa 1, 2 e 3.

Possiamo agevolmente definire  $M$  per *pattern matching*, come alla pagina seguente. In una qualunque applicazione, i “casi” saranno esaminati nell’ordine in cui sono stati dichiarati, fermandosi al primo ove il parametro “fa match” (si accoppia, trova corrispondenza) con l’argomento, cioè ha la sua stessa precisa forma.

```

fun M []      = []
| M (2 :: xs) = xs
| M (3 :: xs) =
  let val ys = M xs
  in
    if ys = []
    then []
    else ys @ [2] @ ys
  end
| M _      = [];

```

Questo è dunque il significato della definizione scritta qui sopra:

1. M applicata a [] restituisce [];
2. M applicata a una lista che inizia con 2 ne restituisce la *coda*, cioè la (parte di) lista che rimane escludendo il primo elemento; si noti che, se la lista ha come unico elemento il 2 (cioè se la sua coda xs è vuota), allora è restituita la lista vuota, da interpretare come “nessuna uscita”, e il calcolo *termina* comunque;
3. M applicata a una lista che inizia con 3 innesca un calcolo *ricorsivo*, poiché a M è sottoposta la coda di tale lista (che è una lista più corta!): se questo calcolo fornirà [] allora il risultato sarà [], se invece fornirà una lista non vuota ys allora il risultato sarà la concatenazione delle liste ys, [2] e ancora ys;
4. in tutti gli altri casi, il risultato del calcolo è [] (il simbolo \_ denota una “variabile anonima”, che fa match con qualsiasi argomento): ciò è in linea col fatto, asserito, che la macchina non produce nulla in uscita, ma *si ferma*.

Se l'ultimo caso non fosse esplicitamente previsto, l'interprete di ML avvertirebbe l'utente con un messaggio di *warning* simile a «Patterns not exhaustive»; allora, se la lista argomento di un'applicazione di M non fosse vuota né iniziasse per 2 o per 3, sarebbe sollevata un'eccezione di mancato match. Infatti, vorrebbe dire che la funzione è stata applicata a un argomento per il quale essa non è stata definita. Normalmente, se un'eccezione non è catturata, per poi essere trattata a dovere, provoca un errore a *run time* e il conseguente aborto del processo di valutazione. In virtù del quarto (e ultimo) caso, il primo può essere omesso. Osserviamo ancora che scrivere soltanto l'espressione ys @ [2] @ ys tra i delimitatori **in** ed **end** sarebbe un errore grossolano: ad esempio, l'applicazione di M a [3, 3, 2] restituirebbe [2, 2, 2] anziché []. Infatti la macchina M non accetta la sequenza 332: dovrebbe esserci almeno un'altra cifra dopo il 2! Quando la funzione M dà come risultato [] significa che la macchina M si ferma senza produrre nulla. Notate che la funzione M dà sempre un risultato, come la macchina M termina su ogni input.

Volendo indagare ulteriormente sul comportamento della macchina da lui costruita, McCulloch la dota di un dispositivo in grado di prendere la sequenza prodotta in output e di riportarla in input per sottoporla a sua volta alla macchina. Attenzione:

anche qualora si inizi fornendo alla macchina una sequenza della forma da essa accettata, è possibile che poi giunga in input una sequenza che non produce alcun output; in tal caso, come abbiamo detto, la macchina semplicemente si fermerà.

Ad esempio: se si introduce 213, si ottiene 13 che, sottoposta alla macchina stessa, la farà fermare. Se invece si introduce 332 (che non è accettata), M attiva – come prima – una copia di sé stessa con input 32, questa copia ne attiva un'altra con input 2, la quale non produce nulla (2 non è una sequenza accettata) e, come effetto finale, si suppone ancora che la macchina si arresti (senza produrre nulla in output).

Com'è facile intuire, M col nuovo dispositivo potrà non fermarsi mai più: questo certamente accade, ad esempio, quando una sequenza produce sé stessa (si veda la prima domanda) o quando, dopo due o più passi, si ritrova in output la sequenza inizialmente data in input a M o magari un'altra delle sequenze passate in input successivamente. Ma possono esserci delle sequenze che non faranno mai terminare il processo pur non ripetendosi...

*Quarta domanda:* che cosa si può dire circa le sequenze della forma  $3d3$ , dove  $d$  è una sequenza (non vuota) di sole cifre 2?

*Quinta domanda:* che cosa si può dire circa le sequenze della forma  $3d32$ , dove  $d$  è una sequenza (non vuota) di sole cifre 2?

*Sesta domanda:* che cosa si può dire circa le sequenze costituite da 32 ripetuto almeno due volte e terminate da una cifra 3, e cioè 32323, 3232323, ...?

La macchina che itera il procedimento può essere rappresentata mediante la funzione MI così definita in ML:

```
fun MI [] = [] : int list
  | MI L = MI (M L);
```

Se MI è applicata a una lista L non vuota, allora l'output sarà dato da una nuova applicazione di MI al risultato restituito dall'applicazione di M alla lista L.

Oltre al fatto che questa macchina rende bene sia l'idea della *ricorsione* sia quella affine dell'*iterazione*, vi è un altro aspetto assai interessante per un informatico: si può provare piuttosto facilmente che *non esiste* alcuna sequenza  $h$  tale che, per ogni sequenza accettabile  $x$ , se  $x$  non farà mai terminare il processo allora  $hx$ , prima o poi, farà arrestare la macchina, e viceversa se  $x$ , prima o poi, farà arrestare la macchina allora  $hx$  non farà mai terminare il processo... E questa circostanza è strettamente legata all'*indecidibilità* del “problema dell'arresto” (*halting problem*), provata nel 1936 dal grande matematico e logico inglese Alan M. Turing (1912-1954).

Vale la pena di fare uno sforzo per comprenderne la ragione, visto che richiama un risultato di grande rilevanza non soltanto in informatica, ma nella storia del pensiero: fu il primo esempio concreto di problema *indecidibile* (cioè non completamente risolvibile, in generale, con un *algoritmo*), che segnò per sempre un limite a ciò che è calcolabile in modo automatico.

Turing dimostrò che non può esistere un algoritmo (oggi potremmo dire un programma *software* eseguibile da un *computer* senza alcun limite, né di memoria né di tempo) che, presa la codifica  $p$  di un algoritmo arbitrario e preso un input  $w$  (una sequenza finita di *bit*) pure arbitrario, decide se  $p$  termina su  $w$ : questa affermazione che coinvolge  $p$  e  $w$  è *soltanto semidecidibile*, nel senso che, in generale, tutto ciò che si può fare in modo automatico si riduce a eseguire l'algoritmo con codifica  $p$  sull'input  $w$  e segnalare in uscita soltanto il caso in cui l'esecuzione termina. Tale compito può essere svolto da un *algoritmo universale*, di cui un moderno computer costituisce una realizzazione limitata (almeno quanto a disponibilità di memoria).

Detto con altre parole: un algoritmo universale è *incompleto*, nel senso che non può risolvere tutti i problemi riguardanti il proprio funzionamento. In particolare, benché possa emulare localmente, passo dopo passo, qualsiasi algoritmo su qualsiasi input, non può decidere a priori, globalmente, se un tale processo darà o no un risultato.

Come notò Church, la *logica predicativa* è *indecidibile*: non esiste alcun algoritmo in grado di stabilire se una sua formula arbitraria è *valida* (cioè vera in ogni possibile modello, ciò che qui coincide con la dimostrabilità) o no; può solo semideciderlo. Un tale algoritmo potrebbe infatti decidere, tra l'altro, le istanze del problema dell'arresto, giacché è possibile tradurre queste istanze in formule logiche. Dunque, una limitazione sulla dimostrabilità appare già nella logica, non solo nell'aritmetica! Dal risultato di Turing deriva immediatamente che la non-terminazione di  $p$  su  $w$  (entrambi arbitrari) non è nemmeno semidecidibile: è *totalmente indecidibile*.

Un'altra conseguenza, tra le tante: in pratica, non v'è speranza di poter verificare in modo automatico la correttezza (a livello semantico) di un programma software arbitrario – si pensi a un compilatore oppure a un modulo che realizza la specifica di un tipo di dato astratto con tutte le sue operazioni... Le dimostrazioni di correttezza possono essere tentate, caso per caso, associando alle istruzioni certe condizioni che devono essere soddisfatte, prima o dopo l'esecuzione, e derivando da queste una condizione finale di coerenza. Tale procedimento, che può sì avvenire con l'aiuto dell'elaboratore, ma che coinvolge *in primis* l'opera della mente, oltre che difficile è estremamente laborioso, e può essere di fatto applicato a programmi piuttosto brevi. Di solito i programmi vengono controllati mediante la verifica di un gran numero di casi sperimentali – indicativi e, si spera, esaurienti circa le possibilità d'errore – piuttosto che con una rigorosa dimostrazione matematica. A volte, quando sono resi operativi, contengono ancora dei "bachi" e non sono immuni da eventi imprevisti...

Ma torniamo alla macchina di McCulloch, e cerchiamo dunque di capire il motivo per cui non esiste alcuna sequenza  $h$  che abbia la proprietà suddetta.

Osserviamo dapprima il seguente fatto. Supponiamo che  $x$  e  $y$  siano due sequenze tali che  $x$ , sottoposta alla  $M$  originaria, produca  $y$ . Se  $y$  provocherà prima o poi l'arresto della macchina "iterativa", allora lo farà anche  $x$  (infatti, se  $y$  conduce a una sequenza inaccettabile  $z$  in  $n$  passi, allora  $x$  condurrà a  $z$  in  $n+1$  passi); altrimenti, cioè se  $y$  non conduce mai a una sequenza inaccettabile, di nuovo  $x$  farà la stessa cosa. Quindi, se  $x$  produce  $y$ , allora  $x$  e  $y$  si comporteranno allo stesso modo rispetto alla terminazione.

Consideriamo ora una qualsiasi sequenza  $h$ . In base alle regole a cui obbedisce la macchina, esiste una sequenza  $x$  che produce  $hx$  (precisamente,  $x$  è  $32h3$ ). Pertanto, rispetto alla terminazione,  $32h3$  e  $h32h3$  si comporteranno allo stesso modo. Così non può esistere alcuna sequenza  $h$  tale che, *per ogni* sequenza  $x$ , le due sequenze  $x$  e  $hx$  si comportino in maniera diversa: ci sarà sempre la particolare sequenza  $32h3$  per cui ciò non è vero. Se una tale  $h$  esistesse, si sottoporrebbero (in parallelo) alla macchina gli input  $x$  e  $hx$ : prima o poi, uno e uno solo dei due processi terminerebbe. In conclusione, nessuna macchina che obbedisca alle regole che governano quella realizzata da McCulloch (ed eventualmente ad altre regole) può risolvere il problema del proprio arresto!

**Soluzioni.** Passiamo a dare le risposte alle domande fatte; per dimostrazioni, approfondimenti e ulteriori domande, si vedano le parti III e IV di *Donna o tigre?*, in particolare i capitoli 9 e 17.

- 1) Sì, la sequenza 323, che è l'unica ad avere tale proprietà.
- 2) Sì, la sequenza 33233, che è l'unica ad avere tale proprietà.
- 3) La sequenza 3232323; si noti che la sequenza data ha la forma  $3x$  e produce  $x2x$ .
- 4) Ciascuna di esse, prima o poi, conduce a sé stessa, e quindi la macchina non si fermerà.
- 5) Ciascuna di esse, prima o poi, conduce a una sequenza della stessa forma, con una cifra 2 in più, e quindi la macchina non si fermerà.
- 6) Ciascuna di esse, ad ogni passaggio successivo, conduce a una sequenza della stessa forma e più lunga, e quindi la macchina non si fermerà.

Nello stesso libro, vi è un altro capitolo, il 16, assai accattivante e abbastanza accessibile anche per un giovane studente, in cui si trova lucidamente espressa l'idea che sta alla base dell'incompletezza gödeliana riguardante l'aritmetica e i "sistemi affini": di nuovo con qualche licenza, vediamo come si presenta...

### Macchine che parlano di sé stesse.

Come si diceva, qui Smullyan è riuscito a porre sotto una luce particolarmente chiara un altro concetto centrale nella storia del pensiero, forse il più grande contributo alla logica (non soltanto matematica) dai tempi di Aristotele e Crisippo: infatti, l'idea che ha portato Gödel alla formulazione del suo (primo) teorema di incompletezza (che precede d'un lustro l'importante prova di Turing e il teorema di Tarski) è qui esposta ricorrendo ancora a una semplicissima macchina!

Consideriamo i quattro simboli: S, N, A,  $\neg$ , e tutte le sequenze (finite e non vuote) con essi costruibili, che chiamiamo *espressioni*; ad alcune, che diremo *proposizioni*, assegneremo un significato.

Supponiamo di avere una macchina che può stampare certe espressioni ma non altre. Diciamo che un'espressione è *stampabile* se la macchina è in grado di stamparla. Assumiamo che ogni espressione stampabile sarà prima o poi stampata dalla macchina, una volta che questa sia avviata.

Data un'espressione  $x$  qualsiasi, definiamo *associata* di  $x$  l'espressione  $x-x$  (che indicheremo con  $A-x$ ) e definiamo *proposizioni* tutte e sole le espressioni che hanno una delle seguenti quattro forme:

- $S-x$  (S sta per "stampabile") significa " $x$  è stampabile";
- $NS-x$  (N sta per "non") significa " $x$  non è stampabile";
- $SA-x$  significa "l'associata di  $x$  è stampabile";
- $NSA-x$  significa "l'associata di  $x$  non è stampabile".

Così, ad esempio, la proposizione  $S-ANN$  dice che l'espressione ANN è stampabile: ciò può essere vero o falso, ma è ciò che afferma! La proposizione  $NS-ANN$  dice invece l'esatto contrario, e cioè che l'espressione ANN non è stampabile.

Ora il lettore può ben chiedersi per quale motivo usiamo il trattino come simbolo: perché non scriviamo semplicemente  $Sx$  anziché  $S-x$  per esprimere l'affermazione che  $x$  è stampabile? La ragione è che l'omissione del trattino creerebbe ambiguità: che cosa vorrebbe dire, ad esempio,  $SAN$ ? Significherebbe che l'associata di N è stampabile oppure che l'espressione AN è stampabile? Con l'uso del trattino non si hanno ambiguità di questo tipo:  $SA-N$  ha il primo significato,  $S-AN$  il secondo.

Forse, prima di proseguire, può essere utile qualche altro esempio circa il significato di alcune proposizioni:

- $S--x$  " $-x$  è stampabile";
- $S--$  " $-$  è stampabile";
- $SA--$  "l'associata di  $-$  (cioè  $--$ ) è stampabile";
- $S----$  *idem*: " $--$  è stampabile";
- $NSA--S-A$  "l'associata di  $-S-A$  non è stampabile";
- $NS--S-A--S-A$  *idem*: " $-S-A--S-A$  non è stampabile".

Possiamo dunque stabilire che, per ogni espressione  $x$ , valgono le seguenti quattro *leggi* (della verità):

- $S-x$  è vera se e solo se  $x$  è stampabile (dalla macchina);
- $NS-x$  è vera se e solo se  $x$  non è stampabile;
- $SA-x$  è vera se e solo se  $x-x$  è stampabile;
- $NSA-x$  è vera se e solo se  $x-x$  non è stampabile.

Abbiamo qui una curiosa situazione "circolare": infatti, la macchina stampa proposizioni che fanno affermazioni su ciò che la macchina può o non può stampare! In questo senso, la macchina parla di sé stessa – o, più precisamente, stampa proposizioni su sé stessa.

Si dà ora il caso che la macchina sia assolutamente *corretta*, cioè che non stampi mai una proposizione falsa: stampa soltanto proposizioni vere.

Questo fatto ha parecchie conseguenze. Ad esempio, se stampa  $S-x$ , deve prima o poi stampare (o aver già stampato) anche  $x$ , poiché, dal momento che ha stampato  $S-x$ ,  $S-x$  deve essere vera, il che vuol dire che  $x$  è stampabile, e quindi prima o poi la macchina deve stampare anche  $x$ . Analogamente, se la macchina stampa  $SA-x$ , allora (poiché  $SA-x$  deve essere vera) deve stampare anche  $x-x$ .

Ne segue inoltre che, se la macchina stampa  $NS-x$ , *non può* stampare anche  $S-x$ , poiché queste due proposizioni non possono essere entrambe vere (infatti, la prima dice che la macchina non stampa  $x$ , la seconda dice che stampa  $x$ ), e *non può* stampare nemmeno  $x$  (perché, altrimenti, avrebbe stampato una proposizione falsa: proprio  $NS-x$ ).

*Sfida:* trovare una proposizione vera che la macchina non possa stampare!

Smullyan scrive che questo problema mette l'idea di Gödel «sotto la luce più chiara che io possa immaginare»...

**Soluzione.** Come sappiamo, per ogni espressione  $x$ , la proposizione  $NSA-x$  dice che l'associata di  $x$ , cioè  $x-x$ , non è stampabile. Prendiamo allora come  $x$  l'espressione  $NSA$ :  $NSA-NSA$  dice che l'associata di  $NSA$  non è stampabile. Ma l'associata di  $NSA$  è proprio  $NSA-NSA$ !

Quindi  $NSA-NSA$  afferma la propria non-stampabilità; in altre parole, la proposizione è vera se e solo se non è stampabile. Ciò vuol dire che o è vera e non stampabile, o è falsa e stampabile. Non si può dare la seconda alternativa, poiché la macchina è corretta. Pertanto deve valere la prima: la proposizione è vera, ma non stampabile!

Anche la macchina di McCulloch può essere collegata al teorema di Gödel nel modo seguente. Supponiamo di avere un sistema matematico con proposizioni, alcune delle quali sono dette vere, e alcune dimostrabili. Assumiamo che il sistema sia *corretto* cioè che ogni proposizione dimostrabile sia vera. Ad ogni “numero” (sequenza di cifre)  $x$  è associata una proposizione che chiameremo *Proposizione x*. Supponiamo che il sistema soddisfi le due condizioni:

*Mc1)* per ogni coppia di numeri  $x$  e  $y$ , se  $x$  produce  $y$  nella versione originale della macchina di McCulloch, allora la *Proposizione 8x* è vera se e solo se la *Proposizione y* è dimostrabile;

*Mc2)* per ogni numero  $x$ , la *Proposizione 9x* è vera se e solo se la *Proposizione x* non è vera.

In queste ipotesi, si può trovare un numero  $x$  tale che la *Proposizione x* sia vera ma non dimostrabile nel sistema.

**Soluzione.** La soluzione è 9832983. Notando l'analogia con la risposta alla sfida, motivata sopra, è facile verificarlo.

Infatti, 32983 produce 9832983; quindi, per *Mc1*, la *Proposizione 832983* è vera se e soltanto se la *Proposizione 9832983* è dimostrabile.

Inoltre, per *Mc2*, la *Proposizione 9832983* è vera se e soltanto se la *Proposizione 832983* non è vera.

Da questi due fatti segue che la *Proposizione 9832983* è vera se e soltanto se non è dimostrabile!

## L’isola G.

Tanti dei problemi logici proposti da Smullyan nei suoi libri sono sagaci varianti o estensioni di rompicapi e indovinelli classici. Già nella prima raccolta pubblicata nel 1978, *Qual è il titolo di questo libro?* – che Martin Gardner definì la più originale, profonda e spiritosa mai scritta nel suo genere – gli abitanti di varie isole sono cavalieri, che dicono sempre la verità, e furfanti, che mentono sempre.

Smullyan prende spunto dalla storia delle due porte e dei due guardiani, uno che dice sempre il vero e l’altro che mente sempre, ma non si sa quale sia l’uno e quale sia l’altro. Una porta conduce al paradiso, l’altra all’inferno, e il rompicapo consiste nello scoprire quale sia la porta per il paradiso, facendo una sola domanda a uno dei due guardiani. Una possibile soluzione è chiedergli «Quale porta mi indicherebbe l’altro guardiano se gli chiedessi la via per l’inferno?» e prendere senza indugio la porta indicata. Anche noi, parlando di nani sinceri e nani bugiardi, ci siamo ispirati agli indovinelli di Smullyan!

Nell’ultimo capitolo di quel volume, l’autore ha immaginato un’isola – chiamata G in onore di Gödel – in cui, al solito, ogni abitante è o un *cavaliere* (proposizione vera) che dice sempre la verità, o un *furfante* (proposizione falsa) che mente sempre. Tuttavia, alcuni cavalieri sono detti *cavalieri confermati* (e corrispondono alle nostre proposizioni dimostrabili, “stampabili”) e certi furfanti sono detti *furfanti confermati* (proposizioni *refutabili*, la cui negazione è dimostrabile).

Ora è impossibile per un qualsiasi abitante dell’isola G dire «io non sono un cavaliere» (che equivale ad affermare «io sono un furfante»), poiché un cavaliere non mentirebbe mai, dicendo di non esserlo, e un furfante non ammetterebbe mai, dicendo il vero, di non essere un cavaliere.

Questo ragionamento riconduce al classico “paradosso del mentitore”, a cui Crisippo di Soli dedicò ben 28 dei suoi 700 “libri” perduti. A questo filosofo, vissuto nel III secolo a. C., dobbiamo i fondamenti della logica proposizionale, in particolare l’adozione della conveniente *implicazione materiale* (limitandoci alla falsità strettamente necessaria, A implica B è falso se e soltanto se A è vero e B è falso), che egli riprese da Filone di Mégara, e di alcune essenziali regole di inferenza: *modus ponens*, *modus tollens*, sillogismo disgiuntivo e sillogismo ipotetico...

Bisogna però notare un fatto importante: è possibile che un abitante dell’isola G dica «io non sono un cavaliere *confermato*». Infatti, da questa affermazione non deriva alcuna contraddizione; tuttavia ne segue qualcosa di assai interessante, e cioè che colui che parla *deve essere in effetti* un cavaliere, e non confermato<sup>6</sup>... esattamente come una proposizione (vera) che afferma la propria indimostrabilità in un sistema logico-deduttivo che permetta la formalizzazione dell’aritmetica!

Smullyan continuò a raffinare la sua opera di divulgazione di queste idee anche dopo *Donna o tigre?*: in *Forever Undecided: A Puzzle Guide to Gödel* (Knopf, 1987)

---

<sup>6</sup> Un furfante non può fare questa affermazione, nel suo caso vera; quindi chi parla deve essere un cavaliere; e, poiché tutti i cavalieri dicono il vero, non può trattarsi di un cavaliere confermato.

trattò di sistemi formali e di ciò che può essere provato al loro interno, assimilandoli a persone che ragionano e alle loro credenze. Ad esempio, se un nativo di un’isola di cavalieri e furbanti dice a un pensatore sufficientemente consapevole «Non crederai mai che io sia un cavaliere», l’interlocutore non può credere che il nativo sia un cavaliere né che sia un furbante senza diventare incoerente, poiché in entrambi i casi arriverebbe ad avere due credenze tra loro contraddittorie.

Il (primo) teorema di incompletezza afferma che, per ogni sistema formale  $S$  “sufficientemente potente”, esiste una proposizione che può essere interpretata come “questa proposizione non può essere dimostrata in  $S$ ”. Quindi, se  $S$  è coerente, né tale proposizione (che è vera) né il suo contrario (cioè la sua negazione, che è falsa) potranno essere dimostrati all’interno di  $S$  stesso.

Nel 1936, il logico polacco Alfred Tarski (nato Teitelbaum, 1902-1983) – fondatore della moderna semantica e della teoria dei modelli – provò che, nelle stesse ipotesi, la verità in  $S$  non può essere un predicato del sistema stesso: non può essere definita nel linguaggio di  $S$ , bensì in un suo metalinguaggio...

In rete, vi è un articolo di Smullyan che anticipa i temi di *Forever Undecided: Logicians who reason about themselves*, presentato alla prima conferenza TARK (*Theoretical Aspects of Rationality and Knowledge*) nel 1986; si trova all’indirizzo [http://www.tark.org/proceedings/tark\\_mar19\\_86/p341-smullyan.pdf](http://www.tark.org/proceedings/tark_mar19_86/p341-smullyan.pdf).

### Un indovinello “doppiamente gödeliano”.

Ritorniamo alla macchina che stampa proposizioni su sé stessa per affrontare una nuova interessante questione: esistono due proposizioni  $x$  e  $y$  tali che *una* delle due è vera ma non stampabile, tuttavia è impossibile stabilire quale delle due. Sapreste trovarle?

**Soluzione.** Prendiamo  $x = S \neg y$  con  $y = \text{NSA} \neg S \neg \text{NSA}$ .

Il significato di  $y$  è che l’associata di  $S \neg \text{NSA}$  (che è  $S \neg \text{NSA} \neg S \neg \text{NSA}$ , cioè  $x$ ) non è stampabile. Quindi:  $y$  dice che  $x$  non è stampabile,  $x$  dice che  $y$  è stampabile.

Ora, supponiamo che  $x$  sia stampabile; allora  $x$  sarebbe vera, il che vorrebbe dire che  $y$  è stampabile. Ma allora anche  $y$  sarebbe vera, il che vorrebbe dire che  $x$  non è stampabile: e così si arriverebbe a una contraddizione!

Quindi, bisogna ipotizzare che  $x$  non sia stampabile; allora  $y$  deve essere vera. A questo punto si danno due casi: o  $x$  è vera, e allora  $y$  è stampabile, o  $x$  è falsa, e allora  $y$  non è stampabile. Riassumendo, nei due casi abbiamo:

- a)  $x$  è vera e non stampabile,  $y$  è vera e stampabile;
- b)  $x$  è falsa e non stampabile,  $y$  è vera e non stampabile.

Quindi nel primo caso è  $x$  che è vera e non stampabile, mentre nel secondo caso è  $y$ , e non v’è modo di stabilire quale delle due alternative si abbia!

(Per inciso, c’è un’altra soluzione al problema posto:  $x = \text{SA} \neg \text{NS} \neg \text{SA}$  e  $y = \text{NS} \neg x$ : il lettore è invitato a verificarlo!)

Analogamente, se sull’isola G ci sono due abitanti,  $x$  e  $y$ , e  $x$  afferma che  $y$  è un cavaliere confermato, mentre  $y$  afferma che  $x$  non è un cavaliere confermato, che cosa possiamo dedurre? Che uno di loro è un cavaliere non confermato, ma non v’è modo di stabilire quale dei due lo sia: se lo è  $x$ , allora  $y$  è un cavaliere confermato; se lo è  $y$ , allora  $x$  è un furbante (confermato o meno)... e non vi sono altre possibilità!

## Due macchine che parlano di sé stesse e ciascuna dell’altra.

Aggiungiamo al nostro alfabeto un quinto simbolo, R, e consideriamo *due* macchine, M1 ed M2, ognuna delle quali stampi delle espressioni composte con i cinque simboli (S, R, N, A,  $\neg$ ).

Interpretiamo ora “S” come “stampabile da M1” e “R” come “stampabile da M2”.

Le diverse forme di *proposizioni* raddoppiano: sono ora otto...

Supponiamo che M1 stampi soltanto proposizioni *vere*, M2 soltanto proposizioni *false*, e diciamo che una proposizione è *dimostrabile* se e solo se è stampabile da M1, *refutabile* se e solo se è stampabile da M2.

(Pertanto i simboli “S” e “R” possono essere letti come “dimostrabile” e “refutabile”, rispettivamente.)

*Ultima sfida:* trovare una proposizione che sia falsa, ma non refutabile.

**Soluzione.** RA–RA, che dice che l’associata di RA (che è RA–RA stessa) è refutabile.

Quindi RA–RA è vera se e solo se è refutabile. E poiché non può essere vera e refutabile, deve essere falsa e non refutabile!

## I programmi dei professori Roberts.

Nel successivo *Satana, Cantor e l’infinito*, del 1992, entrano in scena nuovi personaggi, tra i quali i due fratelli professori Charles Roberts e Daniel Chauncey Roberts – come apparirà chiaro, i nomi non sono scelti a caso!

Il primo ha ideato un linguaggio di programmazione in cui un programma è una qualsiasi sequenza (finita e non vuota) di lettere maiuscole.

Nel seguito, useremo una lettera *corsiva minuscola* ( $x$ ,  $y$ ,  $z$ , ...) per indicare uno qualsiasi di questi programmi.

Il professor Charles Roberts ha stabilito due regole per dare dei “nomi” ai suoi programmi:

*Regola Q:*  $Qx$  è un nome di  $x$

(Si noti che anche  $Qx$  è un programma: infatti si è aggiunta in testa a  $x$  una lettera maiuscola, Q.)

*Regola R:* se  $y$  è un nome di  $z$ , allora  $Ry$  è un nome di  $zz$

(Ovviamente,  $zz$  è il programma ottenuto scrivendo due volte di seguito le lettere che costituiscono il programma  $z$ .)

Quando un programma viene eseguito, produce in uscita un altro programma, secondo un'unica regola:

*Regola C:* se  $y$  è un nome di  $z$ , allora  $Cy$  produce  $z$

E ora veniamo alle domande!

- 1) Qual è il nome del programma ABC?
- 2) Quali nomi ha il programma AA?
- 3) RQABAB è un nome di ...?
- 4) RRQAB è un nome di ...?
- 5) Siete capaci di trovare un  $x$  che sia nome di sé stesso?
- 6) Può accadere che  $Rx$  sia un nome di  $x$ ?
- 7) Può accadere che  $Rx$  sia un nome di  $xx$ ?
- 8) RQABRQ è un nome di ...?
- 9) Che cosa produce il programma CRQABCRQ?
- 10) Siete capaci di trovare un programma che produce sé stesso?
- 11) Siete capaci di trovare un programma  $x$  che produce due volte sé stesso, cioè  $xx$ ?

**Soluzioni.** Qui ci limitiamo a dare le risposte, aggiungendovi qualche commento essenziale; per i necessari approfondimenti, rimandiamo al capitolo 11 di *Satana, Cantor e l'infinito*.

- 1) QABC
- 2) QAA e RQA
- 3) ABABABAB
- 4) ABABABAB di nuovo!
- 5) RQRQ è l'unico!
- 6) Sì, quando  $x$  è QQ
- 7) Sì, quando  $x$  è RRQRQ
- 8) ABRQABRQ
- 9) ABCRQABCRQ
- 10) CRQCRQ
- 11) CRRQCRRQ

Riguardo alla seconda e alla terza domanda, osserviamo che in generale  $RQx$  è un nome di  $xx$ , poiché  $Qx$  è un nome di  $x$  (è un'applicazione della *Regola R*, con  $Qx$  al posto di  $y$  e con  $x$  al posto di  $z$ ).

Un altro fatto da notare è che QQ è un nome di Q, ma Q non è il nome di alcun programma, perché un nome di programma che inizia con Q deve essere costituito da almeno due lettere maiuscole; Q però è un programma, così come lo è QQ! Ciò interessa la sesta domanda: QQ è un nome di Q e quindi (per la *Regola R*, con QQ al posto di  $y$  e con Q al posto di  $z$ ) RQQ è un nome di QQ.

Il programma ABABABAB, oltre ai due nomi che appaiono nella terza e nella quarta domanda, ha ovviamente anche il nome QABABABAB; analogamente, per

quanto concerne l'ottava domanda, il programma ABRQABRQ ha anche il nome QABRQABRQ.

Per capire se ora vi è chiaro questo meccanismo, vi proponiamo ancora una domanda: QQA, RQAA e RQAR sono nomi di quali (diversi) programmi?

Le ultime tre domande coinvolgono la *Regola C*, ossia quella di *creazione* – noi abbiamo detto “produce”, ma la sostanza non cambia: un programma, come un robot, dovrà pur far qualcosa!

In verità, Smullyan sviluppa questo tema in maniera assai più avvincente – e anche un poco più complicata! – immaginando un’isola popolata da robot “intelligenti”, ciascuno dei quali agisce in base a uno di questi programmi, che è “cablato” al suo interno; sicché questi robot ne possono creare altri, fornendoli di un’intelligenza sufficiente per costruire ancora robot intelligenti, i quali a loro volta ne creano altri, e così via all’infinito.

Il fratello di Charles, il professor Daniel Chauncey Roberts, ha aggiunto a questi automi una sola altra regola, quella di *distruzione*:

*Regola D*: se  $x$  produce  $z$ , allora  $Dx$  distrugge  $z$

Così le due regole di *creazione* e *distruzione* riguardano i robot con programma che inizia per C o per D, rispettivamente...

Ne succedono delle belle, come vi invitiamo a constatare leggendo il capitolo 11 del citato libro di Smullyan, e magari i successivi, fino al 15, dove l’autore migliora e completa l’*excursus* sugli approcci più semplici – ma non per questo semplicistici – ai risultati ottenuti da Gödel e Tarski.

Qui noi abbiamo parlato di linguaggi di programmazione con poche regole. Vogliamo ricordare che l’idea di definire linguaggi “essenziali”, con costanti e operatori indispensabili e nulla di superfluo, ha origini lontane.

Nel 1924 il matematico russo Moses I. Schönfinkel, precursore della *logica combinatoria*, nel tentativo di minimizzare il numero di simboli richiesti da una formalizzazione della logica del prim’ordine, trovò il modo di eliminare le variabili legate e di usare soltanto due termini, che chiamò K ed S, per rappresentare due funzioni *intensionali*, ossia due regole di calcolo per qualsiasi funzione di una o più variabili.

Pochi anni dopo, Haskell B. Curry riprese questo studio e ne ampliò la portata, coadiuvato da altri logici impegnati nello sviluppo del *lambda-calcolo*, strettamente collegato alla logica combinatoria. Curry propose un suo “calcolo di (due) combinatori”, le costanti K ed S, con un operatore binario ( $\_\_$ ), detto di combinazione (o applicazione), e due regole di riscrittura:

$$\begin{aligned} ((K\ e_1)\ e_2) &\rightarrow e_1 \\ (((S\ e_1)\ e_2)\ e_3) &\rightarrow ((e_1\ e_3)\ (e_2\ e_3)) \end{aligned}$$

dove  $e_1$ ,  $e_2$  ed  $e_3$  sono espressioni (o termini) del linguaggio. In realtà si tratta di due assiomi equazionali (in senso debole), ossia le frecce possono essere invertite.

Questo è uno dei più semplici linguaggi di programmazione universali (si intende rispetto alla classe delle funzioni effettivamente calcolabili); è algebrico puro, senza variabili, con un unico tipo (ovvero *non tipato*). Altre costanti non sono necessarie, in quanto si possono codificare con i due combinatori K ed S. Nel 1930 Curry provò la coerenza di questo sistema, che in seguito contribuì in maniera sorprendente a rendere efficiente la realizzazione di alcuni moderni linguaggi di programmazione funzionali.

Poc’anzi ci è capitato di usare ML, considerato il primo dei moderni, sviluppato nella seconda metà degli anni ’70 presso l’Università di Edimburgo. Qui non possiamo di certo descriverne l’enorme ricchezza; tuttavia, le spiegazioni che hanno accompagnato gli esempi dovrebbero far intuire quali sono gli ingredienti essenziali di un linguaggio funzionale, perlomeno del prim’ordine.

Per trattare di numeri naturali, o di qualsiasi altra entità da essi rappresentabile, bastano una costante `zero` e una funzione unaria `succ` (successore); servono quindi una forma di *espressione condizionale* come `if=_then_else_`<sup>7</sup> e la possibilità di definire funzioni in modo ricorsivo. Ogniqualvolta una funzione sarà applicata, tutte le occorrenze dei parametri saranno sostituite con i corrispondenti argomenti, senza però valutarli subito: gli argomenti saranno valutati se e quando sarà necessario (ML adotta invece una strategia più semplice, che valuta tutti gli argomenti subito prima di un’applicazione, con la sola eccezione del costrutto condizionale). Questo piccolo linguaggio teorico è ispirato dalle idee originali che stanno alla base del LISP, scaturite tra la fine degli anni ’50 e l’inizio dei ’60 dalla mente di uno dei più geniali informatici, John McCarthy (1927-2011), il fondatore dell’intelligenza artificiale.

Nel 1979 David A. Turner sfruttò i combinatori di Curry per ottenere la cosiddetta valutazione *lazy* (“pigra”, contrapposta a *eager* o *greedy*, “ingorda”): poiché non ci sono variabili, nessun meccanismo è richiesto per memorizzarne i valori. Per di più, una volta iniziata, la valutazione di un argomento non deve essere portata a termine in tutti i casi, bensì è interrotta quando non sia più necessario proseguire, vale a dire non appena sia stata calcolata la parte di risultato che interessa.

Già in un articolo del 1976, Daniel P. Friedman e David S. Wise suggerivano che un “costruttore” non dovrebbe valutare i propri argomenti (questi sono valutati qualora vi dovesse accedere una funzione *stretta*), di modo che coi “selettori” (le proiezioni) si possa emulare il costrutto condizionale.

Analogamente, gli elementi di una lista *lazy* non sono valutati fino a che i loro valori non siano richiesti; quindi una lista *lazy* è potenzialmente infinita. Possiamo calcolare una qualsiasi parte finita di una lista infinita, e richiedere a un programma di generare ciascuna parte finita del risultato in un tempo finito.

---

<sup>7</sup> Dal punto di vista semantico, si tratta di un’operazione *non stretta* con quattro operandi: se il primo e il secondo operando sono valutati nello stesso numero allora è valutato il terzo operando, se sono valutati in numeri diversi allora è valutato il quarto. Teniamo tuttavia presente che il processo di valutazione di una espressione può non terminare... Una funzione (espressione o istruzione) non stretta è comunque indispensabile in programmazione!

Più recentemente sono state costruite delle basi formate da un solo combinatore, vale a dire una singola espressione (priva di variabili libere) in grado di generare tutte le espressioni.

Dopo questo brevissimo cenno ai combinatori e alla programmazione funzionale, raccomandiamo la lettura del libro di Smullyan *Fare il verso al pappagallo*, e concludiamo il capitolo con due paragrafi di approfondimento, per chi lo desideri: il primo di logica, il secondo di scacchi.

### I contributi di Gödel alla logica moderna e ciò che ne consegue.

Verso la fine del XIX secolo, il matematico piemontese Giuseppe Peano si propose di formalizzare la teoria dei *numeri naturali* mediante un nucleo di *assiomi*, che oggi noi possiamo scrivere nel seguente modo (ad essi vanno aggiunti quelli dell'usuale predicato binario di uguaglianza, indicato dal simbolo infisso =):

1.  $\forall x (\sim (\text{succ}(x) = 0))$
2.  $\forall x \forall y (\text{succ}(x) = \text{succ}(y) \rightarrow x = y)$
3.  $\forall x (x + 0 = x)$
4.  $\forall x \forall y (x + \text{succ}(y) = \text{succ}(x + y))$
5.  $\forall x (x \cdot 0 = 0)$
6.  $\forall x \forall y (x \cdot \text{succ}(y) = (x \cdot y) + x)$

I primi due assiomi intendono esprimere, rispettivamente, le proprietà “il numero zero non è successore di alcun numero” e “l’operazione successore è iniettiva”; il terzo e il quarto le proprietà dell’addizione, e gli ultimi quelle della moltiplicazione. A questi, Peano aggiunse il *principio di induzione* (un assioma del second’ordine):

7.  $\forall U ((0 \in U \wedge \forall x (x \in U \rightarrow \text{succ}(x) \in U)) \rightarrow \forall x (x \in U))$

in cui la quantificazione universale più esterna è su *insiemi* di numeri naturali; il suo significato è: considerato un qualsiasi insieme  $U$  di numeri naturali, se è vero che 0 appartiene a  $U$  ed è anche vero che se un qualsiasi numero appartiene a  $U$  allora vi appartiene pure il suo successore, allora ne consegue che  $U$  contiene tutti i naturali.

Questa è una *teoria categorica*, cioè ha un solo *modello* (a meno di isomorfismi): l’infinità discreta dei numeri naturali con l’interpretazione consueta dei simboli.

Si potrebbe pensare di sostituire  $x \in U$  con  $P(x)$ , formula ben formata della logica dei predici (del prim’ordine), che esprima la proprietà che deve avere un numero  $x$  per appartenere all’insieme  $U$ , e quindi asserire che:

- 7'.  $(P(0) \wedge \forall x (P(x) \rightarrow P(\text{succ}(x)))) \rightarrow \forall x P(x)$

per ogni  $P$  con una variabile libera  $x$  (eventuali altre variabili libere si intendono quantificate universalmente all’esterno dell’intera formula 7').

Tuttavia 7' è uno *schema* di assioma: sta per un’infinità numerabile di assiomi del prim’ordine (uno per ogni  $P$ ); nell’assioma 7, invece, la variabile  $U$  varia su tutti i possibili sottoinsiemi dei numeri naturali, che sono un’infinità *non* numerabile:

dunque non tutti rappresentabili con formule ben formate. Da ciò, sostanzialmente, discende il fatto che l’aritmetica di Peano del prim’ordine (PA) abbia diversi modelli (mutuamente non isomorfi), anzi – per i teoremi di Gödel – ne ha almeno un’infinità numerabile, e infiniti con dominio di cardinalità numerabile (tra i quali il “modello standard”), appartenenti ad altrettante classi di isomorfismo.

Un risultato “positivo” per la logica predicativa fu raggiunto da Gödel tra il 1929 e il 1930. Se si considera la relazione tra una teoria del prim’ordine e tutti i suoi modelli semantici, allora vi è *completezza*: applicando le inferenze della logica classica, tutto ciò che un automa può dedurre sintatticamente a partire da una teoria (costituita da un certo insieme di enunciati del prim’ordine *coerenti*, cioè tutti veri in *almeno un* “mondo possibile” o, meglio, *modello*) coincide con le *conseguenze logiche* di tale teoria (ossia con quegli enunciati la cui chiusura universale è vera in *ogni* mondo possibile, nel quale siano simultaneamente veri tutti gli enunciati di partenza).

Tuttavia, quando si vuole limitare l’indagine a mondi particolari, come il modello standard dell’aritmetica, le cose cambiano: infatti le formule vere possono essere di più delle conseguenze logiche! Posto che PA sia coerente, Gödel mostrò (nel 1931) che esiste una formula ben formata che non è dimostrabile in PA. Inoltre, si prova che essa non è refutabile in PA, ossia che pure la sua negazione non è dimostrabile in PA: si dice che una tale formula è un enunciato indecidibile in PA o indipendente da PA. Nell’aritmetica esistono enunciati (veri) che non sono dimostrabili: pertanto l’aritmetica non è riducibile alla logica predicativa.

Nel 1936, Turing e Church provarono che non esistono algoritmi per decidere se una formula arbitraria della logica predicativa è un teorema (e dunque conseguenza logica, vera in tutti i mondi possibili), ma soltanto per semi-deciderlo: vale a dire che non vi sono procedure effettive per refutare falsità, in generale.

Nell’ambito dell’aritmetica, invece, Gödel aveva già escluso persino la possibilità di dimostrare tutte le verità, provando che qualsiasi sistema logico-deduttivo corretto del primo ordine è *intrinsecamente incompleto*: non esiste alcuna procedura effettiva per generare (o enumerare) via via *tutti* gli enunciati veri nel modello standard.

In altre parole, non si ha neppure la speranza di poter avere un automa tale che, se qualche proprietà è vera nella teoria dei numeri naturali, allora certamente – prima o poi – la deduce: per fortuna, in generale, non basta un procedimento di calcolo automatico, ma occorrono intuito e ingegno! Vale a dire che l’aritmetica non può essere resa effettiva; e per completare la teoria, bisognerebbe arricchirla con un insieme infinito di assiomi che non è nemmeno effettivamente presentabile.

Questo risultato di incompletezza sintattica per l’aritmetica e le sue estensioni (sì dovuto a Gödel, ma già intuito da Emil Post, che nel 1920-21 provò la completezza della logica proposizionale) rappresenta uno dei passi più significativi della logica dai tempi di Aristotele (IV secolo a. C.), anche nel senso culturale più ampio: ha portato infatti alla comprensione umana la distinzione tra verità e dimostrabilità.

Gödel escogitò un’ingegnosa variazione sul “tema del mentitore” per dimostrare che *ogni* sistema assiomatico coerente (cioè in cui non siano deducibili contraddizioni), abbastanza esteso da permettere lo sviluppo dell’aritmetica, è *incompleto*, nel senso

che esiste sempre una formula chiusa (in particolare, un enunciato universale) che è vera nel modello standard dell'aritmetica, ma che non è deducibile nel sistema assiomatico considerato – e, in conseguenza della sua verità e della correttezza del sistema, non è neppure deducibile la sua negazione (che è falsa): una tale formula è dunque *indimostrabile* nel sistema contemplato. Sebbene si aggiungesse alla teoria questo enunciato indimostrabile, come ulteriore assioma, un nuovo enunciato sarebbe vero senza essere dimostrabile, e così via – bisognerebbe aggiungere un'infinità non effettivamente enumerabile di assiomi, come si è detto.

Disponiamo di vari metodi astratti per provare la coerenza di un tale sistema assiomatico – ma *al di fuori* del sistema stesso! – e quindi per trovare un enunciato indimostrabile. In particolare, una dimostrazione di coerenza (non elementare, semi-costruttiva) della teoria dei numeri naturali fu data da Gerhard Gentzen, nel 1936.

Dunque, per così dire, il “principio del terzo escluso” non vale per la dimostrabilità nell’aritmetica: non è vero che o un enunciato è dimostrabile o lo è la sua negazione. Il tipo di incompletezza di cui abbiamo parlato riguarda la relazione tra un sistema formale (o teoria) dell’aritmetica – chiamiamolo N – e la sua interpretazione nell’insieme dei numeri naturali: vi è incompletezza perché N non dimostra tutto ciò che è legittimamente interpretabile come vero.

La dimostrazione dell’enunciato «la coerenza di N comporta la verità di un certo enunciato universale non dimostrabile in N» è a sua volta formalizzabile in N; pertanto, se N dimostrasse la propria coerenza, allora dimostrerebbe (per *modus ponens*) anche quel certo enunciato, ciò che è escluso dal precedente teorema. Gödel ottenne così un secondo teorema di incompletezza: se N è coerente, l’enunciato che asserisce la coerenza di N non è dimostrabile in N, e dunque dev’essere provato al di fuori di N. Contrariamente alle speranze del matematico tedesco David Hilbert, che al congresso di Parigi del 1900 lo pose come secondo obiettivo nel suo famoso programma d’inizio secolo, non è possibile provare la coerenza di N mediante metodi formali “finitisti”, puramente combinatori, cioè con mezzi riconducibili a procedimenti ricorsivi, accessibili in N stesso.

Dal punto di vista filosofico, l’eventuale successo del programma hilbertiano avrebbe portato a una visione del mondo, della scienza e del pensiero assolutamente meccanicistica, e avrebbe stabilito la completezza (e quindi la chiusura) della matematica, riducendola a procedimenti puramente automatici: tutto sarebbe stato riconducibile a manipolazioni simboliche effettuate da un gigantesco calcolatore e, in particolare, ogni enunciato universale sarebbe stato, oltre che confutabile, anche verificabile con metodi finitisti, e dunque decidibile.

Da Hilbert in poi, e pure dopo i teoremi di Gödel, furono dati comunque notevoli contributi alla *teoria della dimostrazione*, ove l’attenzione si sposta dai risultati (i teoremi) ai metodi: si pensi ai lavori di Jacques Herbrand (interpretazione e minimo modello) e di Gerhard Gentzen (sistema di deduzione naturale). I sistemi formali hanno avuto e svolgono tuttora un ruolo importante in matematica, anche dal punto di vista della ricerca concreta e delle applicazioni, peraltro proprio in informatica. Nel paradigma di programmazione detto *logico*, sviluppato a iniziare dagli anni ’60

e impiegato in tanti progetti di intelligenza artificiale (ad esempio, nei cosiddetti “sistemi esperti”), tipicamente si cerca di dimostrare il risultato da ottenere, partendo da certi assiomi, attraverso deduzioni (corrette). Il capostipite dei linguaggi ispirati da questo paradigma è il PROLOG, acronimo di *PROgrammation en LOGique*, realizzato presso l’Università di Marsiglia nei primi anni ’70. Nella sua accezione pura, un programma è una *teoria* costituita da un insieme finito di formule di forma particolare (che comprendono *fatti* e *regole*) assunte come assiomi; un termine può essere una costante o una variabile, oppure può essere costruito applicando un *funtore* a dei termini. Sicché, nuovamente, con una costante zero e un funtore unario succ, si possono ottenere i termini (senza variabili):

zero, succ(zero), succ(succ(zero)), ...

per rappresentare i numeri naturali. Il modello di riferimento di questo paradigma di programmazione è costituito, in particolare, da un sottoinsieme del *calcolo dei predicati* con poche, opportune *regole di inferenza* – essenzialmente una: il cosiddetto *principio di risoluzione*, introdotto da John A. Robinson nel 1965, per *semi-decidere* se una formula chiusa sia un teorema, e dunque una conseguenza logica, di una teoria del prim’ordine. Nel caso in cui lo sia, aggiungere la sua *negazione* agli assiomi della teoria conduce sintatticamente a una contraddizione (e semanticamente a un insieme di formule *insoddisfattibile*, cioè inconsistente, privo di modelli). Questo principio condensa i tradizionali sillogismi in un’unica regola di inferenza; un insieme di formule è inconsistente se e soltanto se da esso si riesce a derivare il *falso*, in un numero finito di passi, usando soltanto tale regola.

Affinché possa approfondire gioiosamente i concetti accennati in questo paragrafo, segnalo al lettore uno degli ultimi libri pubblicati da Smullyan: *The Gödelian Puzzle Book. Puzzles, Paradoxes and Proofs* (Dover Publications, New York, 2013).

## Dedicato agli scacchisti.

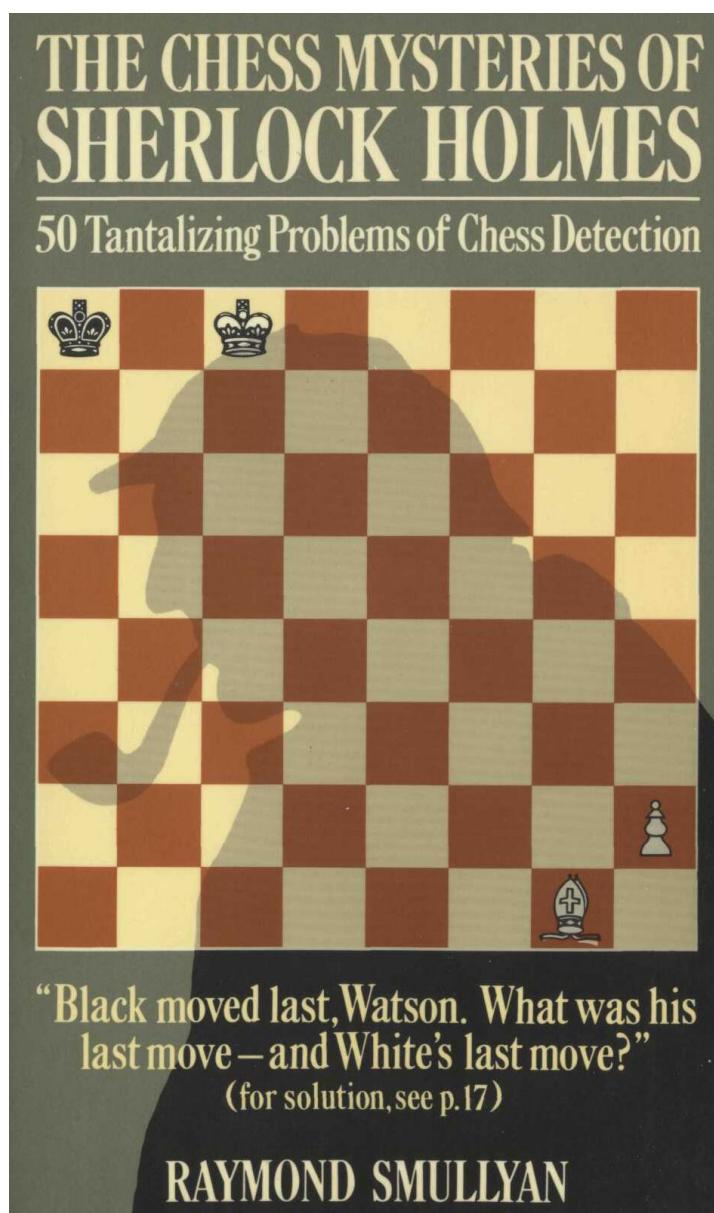
Smullyan raccolse in due libri i suoi migliori problemi di analisi retrograda negli scacchi: *The Chess Mysteries of Sherlock Holmes* e *The Chess Mysteries of the Arabian Knights*, entrambi pubblicati da Knopf (New York, 1979 e 1981) e mai tradotti in italiano.

Già da ragazzo, quando si dilettava a comporre classici problemi di matto in due o tre mosse, egli era affascinato dall’idea di poter ricostruire la genesi di una posizione. La questione non è affatto banale: data una posizione, come si fa a tornare indietro almeno di qualche passo, cioè a sapere quali sono state le ultime mosse di ciascun colore? Innanzi tutto: è possibile? Oppure le risposte potrebbero essere così tante... o forse nessuna?

I problemi che non possono essere risolti sono quelli che Smullyan ha sempre preferito! In effetti, è possibile – anzi, probabile – che si possano presentare numerosissime eventualità e che quindi sia impossibile determinare in modo unico le ultime mosse; d’altra parte, potrebbe darsi il caso che la posizione in esame sia irraggiungibile in una partita ove si rispettino le regole del gioco (e ciò talvolta può

essere arduo da stabilire, sebbene in linea di principio sia sempre possibile), ma nell’analisi retrograda la legalità della posizione è comunque un elemento chiave. I problemi di questo genere che ammettano un’unica soluzione sembrano quasi impossibili; eppure, c’è una ricchissima casistica, interi libri dedicati all’argomento, periodici e siti specializzati, che ne comprendono diverse migliaia. Il fondatore della moderna scuola di retro-analisi scacchistica, l’ingegnere milanese Luigi Ceriani (1894-1969), pubblicò due volumi, il primo nel 1955 e il secondo nel 1961, ritenuti ancor oggi un’autentica bibbia nel settore, contenenti parecchie centinaia di problemi suddivisi per tema, da lui stesso ideati e commentati in maniera sempre precisa e acuta.

Dal canto suo, Smullyan contribuì con i due libri sopra citati, che racchiudono perle di notevole bellezza. Qui sotto è riprodotta la copertina del primo libro, che raffigura – all’ombra del celebre detective – una posizione già proposta dal compositore danese Jan Mortensen nel 1956.

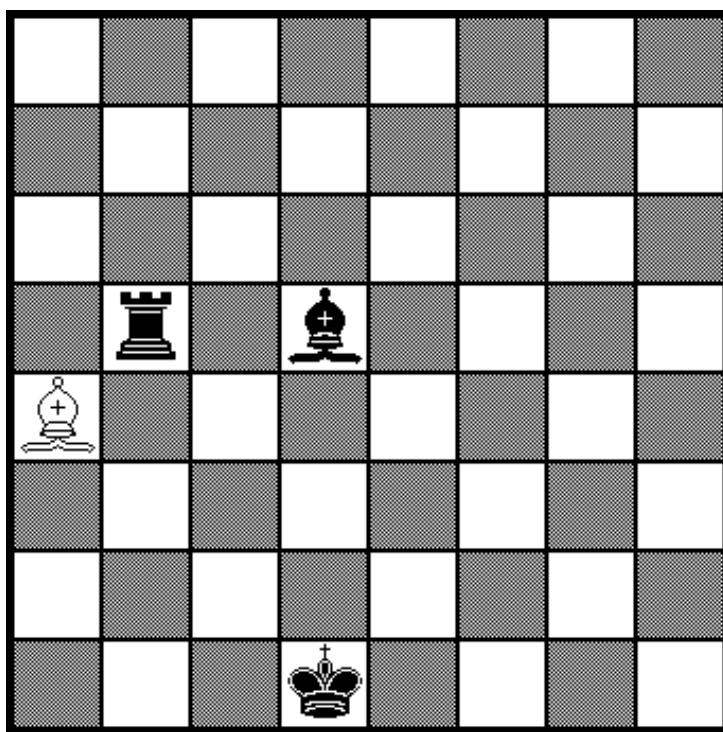


Il problema, che Smullyan definisce “piccolo esercizio”, consiste nel rispondere a due domande: quale mossa ha appena fatto il Nero? E qual è stata la precedente del Bianco? (La scacchiera è orientata correttamente, per cui il Re nero si trova in a8.)

**Soluzione.** Subito prima dell’ultima mossa, il Re nero doveva trovarsi in a7, dov’era sotto scacco (apparentemente impossibile) da parte dell’Alfiere. In effetti, questo scacco fu di scoperta, ma il pezzo bianco che lo provocò ora non c’è più: è stato appena catturato dal Re nero! Quindi deve trattarsi di un Cavallo proveniente da b6: 1. Cb6–(x)a8+, Ra7xa8.

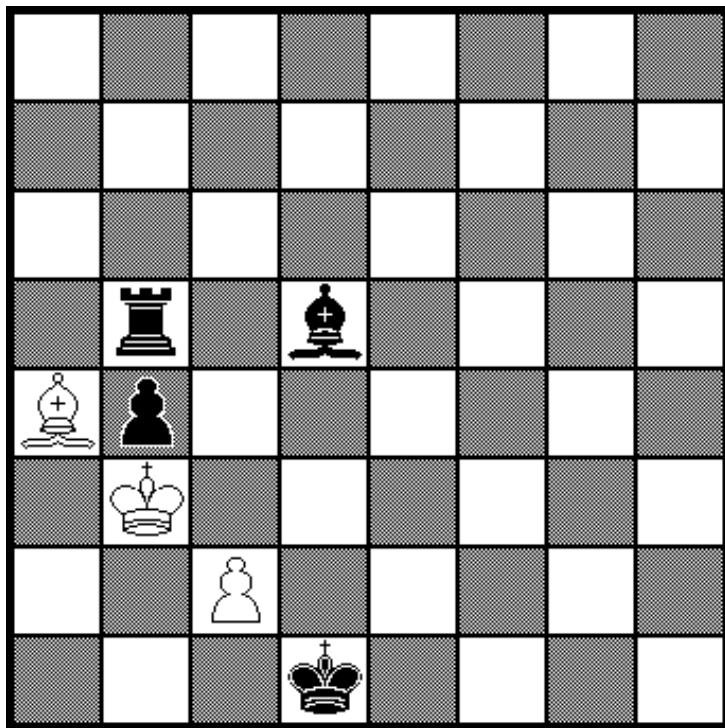
Se però la scacchiera fosse girata, e quindi il Re nero si trovasse in h1, allora sarebbe lecito ipotizzare un’altra ultima mossa per il Bianco: 1. b7–b8 = A+, Rh2–(x)h1.

Vediamo ancora un altro piccolo capolavoro di retro-analisi, in economia di pezzi, in cui si deve pure dedurre quale dei due giocatori abbia fatto l’ultima mossa...



Smullyan pubblicò questo problema nel 1957, sul *Manchester Guardian*; per inciso, a questo quotidiano inglese l’aveva inviato il padre di un suo compagno di studi. Il Re bianco è invisibile. Dove si trova?

**Soluzione.** Se provassimo a collocare il Re bianco in b3, si troverebbe sotto un impossibile doppio scacco. Se lo mettessimo altrove, sarebbe il Re nero a trovarsi sotto scacco d’Alfiere, che si spiegherebbe soltanto se il Re bianco avesse appena lasciato la casa b3... A questo punto, però, non ci troviamo esattamente nello stesso dilemma di prima: il Re bianco deve aver lasciato b3 catturando un pezzo nero che giustifichi in qualche modo il doppio scacco “impossibile”!



Evidentemente, in questa posizione, l'ultima mossa è stata del Nero, che ha spostato il proprio Alfiere in d5, partendo da un'altra casa della diagonale maggiore bianca e mettendo sotto scacco il Re avversario. A partire da qui, il doppio scacco subito dal Bianco è perfettamente spiegabile:

1. c2–c4, b4×c3 *en passant* + ; 2. Rb3×c3+.

Quindi, nel diagramma alla pagina precedente, il Re bianco deve trovarsi in c3, e il Nero ha il tratto.

**Parole chiave:** verità e falsità, contraddizione o paradosso, logica proposizionale e logica predicativa, sistema formale o logico-deduttivo, assioma, teorema, mondo possibile, conseguenza logica, coerenza o consistenza, correttezza e completezza, verità e dimostrabilità, procedura effettiva o algoritmo, ricorsione e iterazione, paradigmi di programmazione funzionale e logico, autoriferimento, problema dell'arresto, enunciati indecidibili e semi-decidibili.

### 3. Piaceri e limiti del calcolo

Da tempi remoti, l'uomo ha mostrato crescente interesse per vari tipi di *macchine* capaci di eseguire automaticamente *procedure* (meccaniche) complesse. Legate a nomi d'illustri matematici quali Pascal e Leibniz, le prime macchine per *calcolare* furono viste più che altro come curiosità scientifiche; ma, già nel corso del XIX secolo, anticiparono – almeno sulla carta – alcune caratteristiche proprie dei moderni elaboratori elettronici.

Infatti, nel 1834, il matematico e inventore inglese Charles Babbage (1791-1871) concepì il progetto di una *macchina analitica*, la cui importanza risiede soprattutto nell'architettura logica sorprendentemente moderna (con un'unità di calcolo “micro-programmata”, *mill*, separata dalla memoria, *store*, efficienti meccanismi di riporto eccetera) e nella sottostante nozione di programmabilità: oggi diremmo un elaboratore meccanico *general-purpose*, cioè programmabile per ogni genere di calcolo.

Due furono le tecnologie che permisero a Babbage di approdare all'idea di memorizzare dati numerici e manipolarli secondo una sequenza possibilmente ripetitiva di operazioni e scelte che, una volta specificate, la macchina potesse compiere in modo del tutto automatico: i cilindri dotati di pioli (già impiegati da Erone di Alessandria, vissuto presumibilmente nel I secolo d. C., e successivamente perfezionati dagli scienziati islamici del Medioevo) e le schede di cartone perforate (usate nell'Ottocento per pianificare i passaggi dei fili colorati nei telai Jacquard, al fine di ottenere sul tessuto la riproduzione di un qualsiasi motivo o disegno, ripetitivo o meno: si tratta del primo impiego certo di una codifica binaria).

Nel 1843 Augusta Ada Byron (1815-1852), contessa di Lovelace, pubblicò le sue celebri *Note* (aggiunte alla traduzione delle memorie di Luigi F. Menabrea sulla presentazione fatta da Babbage all'Accademia delle Scienze di Torino), nelle quali chiaramente intuì la portata innovativa di quel progetto e mise in luce le potenzialità della macchina analitica, proponendone alcune applicazioni, che culminavano nel calcolo dei *numeri di Bernoulli*. La macchina non fu mai interamente costruita – anche per la mancanza di adeguati finanziamenti – nonostante l'impegno profuso dal forse velleitario, ma di certo vulcanico e meticoloso Babbage, per tutto il resto della sua vita.

Il “linguaggio” usato per codificare le operazioni aritmetiche (i cui risultati potevano essere immagazzinati in una o più “locazioni” di memoria) prevedeva, in sostanza, la possibilità di *salti condizionati* (chiamati *experimental operations*) della forma «se il risultato dell'ultima operazione è in una certa relazione con lo zero, allora la macchina torna indietro (o va avanti) di un certo numero di schede, nel nastro delle istruzioni»: ciò è sufficiente per poterlo ritenere un linguaggio *completo* dal punto di vista computazionale. L'esecuzione era pensata a stadi successivi, in *pipeline*, prelevando istruzioni e dati su schede da due unità di lettura distinte, e perforando altre schede con i risultati in uscita – non solo numerici, ma anche consistenti in istruzioni per nuovi programmi!

Il fatto che la macchina analitica, una volta costruita, avrebbe permesso qualsiasi calcolo – e per di più, qualora la memoria non fosse bastata, sarebbero state d’aiuto le schede perforate – era già stato intuito da Babbage; Ada Byron si spinse oltre, affermando che i numeri manipolati dalla macchina possono rappresentare delle *entità* che non siano mere quantità o misure: un’anticipazione del calcolo simbolico! Al tempo stesso, ella ridimensionò l’idea che l’automa fosse “pensante”, affermando: «La macchina analitica non ha alcuna pretesa di *originare* qualche cosa, bensì può fare qualsiasi cosa che *noi sappiamo come ordinarle* di eseguire ». Nel suo saggio *Computing Machinery and Intelligence* del 1950, Turing rese famosa questa affermazione definendola “l’obiezione di Lady Lovelace”.

## Come esprimere un procedimento di calcolo... purché lo si possa fare!

Come abbiamo visto nel precedente capitolo, a partire dagli anni ’30 del Novecento – quindi già prima dell’era dell’elettronica – le possibilità di computazione delle macchine sono divenute oggetto di un’importante branca della logica (matematica): la *teoria della calcolabilità* (e della dimostrazione).

Infatti, in quegli anni, alcuni logici matematici, tra i quali Alonzo Church, Kurt Gödel, Stephen C. Kleene, Alan M. Turing ed Emil L. Post, più o meno indipendentemente l’uno dall’altro, proposero differenti formalizzazioni del concetto di computazione (con notazioni veramente precise e complete, si può dire “a livello macchina”, e in diversi stili), che portarono tutte alla medesima nozione di *funzione effettivamente calcolabile*: “effettivamente” vuol dire mediante un procedimento di calcolo ben specificato, che può essere eseguito da un automa – con “memoria”, ma senza “intelligenza” – in quanto un uomo lo potrebbe compiere in maniera assolutamente deterministica, senza alcuna possibilità di arbitrio né interventi da parte del “caso”.

Ai fini del nostro discorso, è sufficiente pensare a una “funzione” come a una relazione che fa corrispondere a ciascun numero naturale ( $0, 1, 2, \dots$  *ad infinitum*) ancora un numero naturale o altrimenti “nulla”: infatti, qualunque input per un programma può essere riguardato come la rappresentazione (in cifre binarie, ossia *bit*) di un numero naturale, e così pure il suo output, se termina; e proprio il caso di non-terminazione su un input è associato al “nulla”.

La teoria della calcolabilità nacque dunque nell’ambito della logica e, come sappiamo, contribuì presto alla dimostrazione di alcuni fondamentali risultati, anche “negativi”: l’impossibilità di far risolvere certi problemi a un automa...

Uno dei compiti precipui della logica consiste nel cercare di dimostrare come da alcune affermazioni (dette *assiomi*) possano esserne dedotte altre, utilizzando opportune *regole di inferenza*. Ciò si ottiene costruttivamente specificando la sequenza di applicazione delle regole, ossia, in sostanza, indicando un procedimento di computazione, che costituisce la dimostrazione.

In quest’ambito, sorgono questioni del tipo: « Si può dare, descrivendolo in modo finito, un insieme di assiomi e regole da cui sia possibile dimostrare ogni proprietà

che è vera per i numeri naturali?» In effetti, gli enunciati matematici di cui si può dimostrare la verità sono stati spesso ritenuti, per un sistema di pensiero, un fondamento più solido di qualsiasi massima morale o perfino dell’oggettività fisica. Già nel XVII secolo, Leibniz immaginò un sistema di ragionamento in forma di calcolo, grazie al quale si potessero un giorno risolvere tutte le possibili dispute seguendo l’invito: « Signori, calcoliamo! » I progressi fatti dalla logica simbolica diedero fiducia a Hilbert, all’inizio del ’900, circa la possibilità di una risposta affermativa alla questione; il suo sogno fu però infranto da Gödel nel 1931.

Tuttavia, dimostrare che *nessun* metodo puramente meccanico possa risolvere un certo problema presuppone una definizione che comprenda *tutti* i tipi di computazione. Ed è qui che entrano in gioco le formalizzazioni a cui poc’anzi si accennava. Il noto modello computazionale dovuto a Turing<sup>8</sup> equivale – quanto a capacità – a un moderno computer con *memoria potenzialmente illimitata* (oltre che tempo) in grado di eseguire procedure di calcolo, dette *algoritmi*, espresse mediante un appropriato *linguaggio di programmazione*. Esso può eseguire *qualsiasi* computazione, se accettiamo la tesi avanzata da Church nel 1936: ogni funzione che è calcolabile secondo una nozione *intuitiva* di “procedimento di calcolo” è anche calcolabile mediante una macchina di Turing – quindi, secondo una nozione *formalizzata* di procedura di calcolo (ossia, un algoritmo).

In modo ovvio vale il viceversa: pertanto la *tesi di Church* identifica la classe di tutto ciò che è (intuitivamente) calcolabile con la classe di tutto ciò che è (formalmente) calcolabile con le macchine di Turing, o con un moderno computer che avesse il suddetto ideale requisito. Questa tesi non può essere “dimostrata”, proprio perché lega un concetto intuitivo a una nozione formalizzata; al più, potrebbe essere confutata (qualora si trovasse un procedimento di calcolo che possa essere pensato, ma che non si riesca a tradurre in un algoritmo): tuttavia fino ad oggi non è stata confutata, e quindi è comunemente accettata. Se la si accetta, dire “calcolabile” equivale a dire “effettivamente calcolabile”.

Perché abbiamo detto “memoria potenzialmente illimitata”? Soltanto per poter immagazzinare, e in ogni modo elaborare, “numeri” (rappresentanti informazioni) di arbitraria lunghezza, compreso il programma stesso codificato nel *linguaggio macchina* (binario): ogni calcolo che termina userà una quantità di memoria certamente finita, ma in generale non sappiamo a priori quanta ne servirà; invece, un calcolo destinato a non terminare potrebbe richiedere una quantità di memoria infinita. Potremmo anche ragionare così: usiamo computer con risorse sì limitate, ma sempre maggiori, e ci fermeremo se e quando riusciremo a ottenere un risultato...

---

<sup>8</sup> Qui non entriamo nei dettagli: in che cosa consistano le macchine di Turing sarà chiarito nel sesto capitolo. Ricordiamo soltanto che Turing dimostrò l’esistenza di una *macchina universale*, in grado di emulare l’esecuzione di un qualsiasi algoritmo su un qualsiasi input: un interprete ideale! Trascorso neppure un quarto di secolo, John McCarthy e Steve Russell scrissero – in una paginetta di codice LISP – un interprete universale per i programmi in LISP (puro) e lo compilaronon, a mano, nel linguaggio macchina dell’IBM 704...

Ma quali sono gli elementi sintattici che caratterizzano un linguaggio di programmazione *computationalmente completo*, mediante il quale si possa cioè formalizzare un procedimento per calcolare qualsiasi funzione effettivamente calcolabile? Secondo il cosiddetto “paradigma imperativo” (per cui alla macchina sono impartiti dei *comandi*, o *istruzioni*), è essenziale la possibilità di esprimere *cicli* di istruzioni, da ripetersi fino a quando non si verifichi una certa condizione.

Un esempio di linguaggio (imperativo) *completo minimale* è quello che permette la composizione *in sequenza* di istruzioni della forma:

```

 $v = 0$                                 // assegna a una variabile il valore zero;
 $v = v + 1$                              // incrementa di un'unità il valore di una variabile;
while ( $v_1 \neq v_2$ ) {                  // se i valori di due variabili sono diversi,
    sequenza di istruzioni           // allora esegue la sequenza di istruzioni e poi ritorna
}                                         // a valutare la condizione, altrimenti prosegue avanti.

```

Si noti che la struttura sintattica ricorre; la sequenza di istruzioni che costituisce il *corpo* di un ciclo può a sua volta essere composta da istruzioni di tutte le forme ammesse: contenere dunque uno o più cicli, con eventuali altri annidati, e così via.

Se si ha anche la possibilità di scrivere istruzioni di assegnazione come  $v = v - 1$  (decrementa di un'unità il valore di una variabile), allora per i cicli **while** bastano condizioni della forma  $v \neq 0$ . Tuttavia, affinché la “macchina sottostante” possa dirsi davvero *universale*, in grado di calcolare tutto ciò che è calcolabile, è necessario che le variabili – ricordiamo che se ne possono usare un numero finito arbitrario – possano assumere qualsiasi valore naturale, senza limitazioni a priori.

Per usare una metafora, immaginiamo di metterci nei panni di un automa in grado di compiere soltanto poche ma significative operazioni. Supponiamo di avere disponibilità di:

- biglie tutte uguali, a volontà (una biglia rappresenta un'unità);
- sacchetti tutti uguali, a volontà, vuoti ma talmente elastici che vi si possa aggiungere sempre una biglia: il sacchetto si allarga a piacere;
- una bilancia un po' particolare, che ci dica soltanto se due sacchetti hanno lo stesso peso, cioè contengono lo stesso numero di biglie, oppure no: dà un solo bit di informazione!

Le operazioni consentite (quelle “di base”) sono soltanto tre:

- prendere un nuovo sacchetto vuoto e scrivervi sopra un nome (corrisponde all’*allocazione in memoria* di una nuova *variabile*, alla quale è attribuito lo zero come valore iniziale);<sup>9</sup>

---

<sup>9</sup> Poiché si suppone di avere sacchetti vuoti a volontà, nel contesto di questa metafora è inutile prevedere la possibilità di svuotarne uno scelto tra quelli già utilizzati (ciò che corrisponderebbe all’azzeramento di una delle variabili in uso). Si noti comunque che, in ogni momento, il numero di sacchetti utilizzati dall’automa sarà finito, e l’automa potrà sempre attribuire un nome nuovo a ciascun nuovo sacchetto che prenderà, per distinguerlo dagli altri già utilizzati.

- aggiungere una biglia in un sacchetto, individuato tra quelli utilizzati (corrisponde a incrementare di un'unità il contenuto di una certa variabile, assegnandole il successore del valore che conteneva);
- prendere due sacchetti, individuati tra quelli utilizzati, e confrontarne il contenuto mediante la bilancia (corrisponde a confrontare i contenuti di due certe variabili, per sapere se si tratta dello stesso valore oppure no).

Naturalmente, certe variabili hanno come valori iniziali i dati di input, e alla fine (se il processo termina) alcune variabili conterranno i risultati del calcolo.

Primo problema: ci viene dato un sacchetto di biglie; come possiamo fare a calcolare il *predecessore*, cioè a ottenere un sacchetto contenente una biglia in meno rispetto a quello dato?

*Suggerimento:* bisogna prendere un altro sacchetto (vuoto), visto che non è proprio possibile *togliere* biglie; ma uno solo non è sufficiente... E poi, ovviamente, *tutte* le operazioni consentite possono essere ripetute! Ecco l'idea di fondamentale importanza: *ripetere una sequenza di operazioni finché non si verifichi una certa condizione*; ad esempio, si smette di fare qualcosa quando la bilancia ci dice che due sacchetti contengono lo stesso numero di biglie... Rimane tuttavia un problema di fondo: che cosa accadrà (o che cosa dovrà accadere) quando il sacchetto che ci viene dato è vuoto, cioè quando dobbiamo calcolare il predecessore dello zero?

**Soluzione.** Chiamiamo X il sacchetto che ci è stato dato. Prendiamo due sacchetti vuoti, chiamiamoli A e B; aggiungiamo una biglia in B; pesiamo B e X: se hanno peso diverso, aggiungiamo una biglia in A e una in B, e torniamo a pesare B e X... Se a un certo momento osserviamo che hanno lo stesso peso, allora ci fermiamo: in A ci sarà una biglia in meno di quante se ne trovano in X!

Il problema, come si è detto, sorge quando il sacchetto che ci viene dato è vuoto. In tal caso, questa successione di azioni è destinata a continuare per sempre: il *processo* (cioè l'esecuzione di questa sequenza di operazioni, sequenza che è comunque esprimibile in modo finito mediante un *ciclo*) non avrebbe termine... e, a rigore, sarebbe proprio la cosa corretta, poiché la funzione predecessore (nell'aritmetica dei numeri naturali) non è definita sullo zero!

Qualcuno noterà che possiamo prevenire questo pericolo: basta infatti, come prima azione, mettere sulla bilancia un sacchetto vuoto e X e, se hanno lo stesso peso, fermarsi e... gridare: «Aiuto! Non posso calcolare alcun risultato!» In mancanza di un'apposita istruzione condizionale, ce la caviamo ugualmente: anche una *scelta tra due alternative* è esprimibile con le istruzioni elementari della forma vista. Infatti, *in generale*, un'istruzione della forma **if** ( $X \neq 0$ ) { ... } (non prevista nel nostro linguaggio minimale) può essere realizzata introducendo una nuova variabile (diciamo C) riservata a tale scopo, sicché possiamo codificare le azioni da compiere nel modo appresso illustrato (tuttavia, in questo caso, il codice può essere abbreviato, sfruttando B per evitare l'ultimo ciclo, come mostrato in alto a destra):

```

C = 0
while (X != C) {
    A = 0
    B = 0
    B = B + 1
    while (B != X) {
        A = A + 1
        B = B + 1
    }
    // a questo punto, B è uguale a X, mentre A è uguale al predecessore di X;
    // ora bisogna far sì che C giunga ad uguagliare X, per uscire subito dal ciclo esterno:
    while (C != X) {      // la prima volta la condizione sarà vera...
        C = C + 1          // sicché questa istruzione è eseguita almeno una volta!
    }
}

```

Se il dato X è positivo, allora il risultato è A, altrimenti il processo termina senza alcun risultato.<sup>10</sup>

Tuttavia, come abbiamo visto nel precedente capitolo, non è possibile *in generale* programmare una macchina in modo tale che si accorga di essere caduta in un *loop* senza fine!

Passiamo ora a un'altra operazione: ci vengono dati due sacchetti di biglie; come possiamo procedere per calcolare la somma, cioè per ottenere un sacchetto con tante biglie quante ve ne sono complessivamente nei due sacchetti dati?

**Soluzione.** Chiamiamo X e Y i due sacchetti dati; definiamo subito il procedimento con una sequenza di istruzioni:

```

A = 0
B = 0
while (A != X) {
    A = A + 1
}
// adesso A è uguale a X
while (B != Y) {
    A = A + 1
    B = B + 1
}
// a questo punto, B è uguale a Y, mentre A è uguale alla somma di X e Y

```

---

<sup>10</sup> Anticipare l'istruzione A = 0 spostandola al di sopra del primo **while**, e prendere comunque alla fine il valore di A come risultato, equivale a considerare lo zero come predecessore di sé stesso. Ciò permette la definizione della funzione totale chiamata *sottrazione propria*, che estende la sottrazione tra i naturali al caso in cui il primo operando è minore del secondo (e allora il risultato è zero).

Dunque questo calcolo termina sempre con risultato A.  
La traduzione in una funzione nel linguaggio C è immediata:

```
unsigned int sum (unsigned int x, unsigned int y) {
    unsigned int a = 0, b = 0;
    while (a != x) { a++; }
    // a == x
    while (b != y) { a++; b++; }
    // (b == y) && (a == x + y)  (a meno di overflow...)
    return a;
}
```

Il risultato restituito alla fine di un'esecuzione è quello corretto, a meno di *overflow*, cioè di superamento del numero (rappresentabile dalla macchina usata) più lontano dallo zero: ma come può accadere una tale eventualità?

Le *implementazioni* del linguaggio C, essenzialmente costituite dai *compilatori* per le varie macchine, adottano un certo numero prefissato di *byte* (un byte equivale a 8 bit, ossia 8 cifre binarie) per la rappresentazione degli interi senza segno, e di conseguenza un'aritmetica *modulare*. Se, ad esempio, a ciascuna variabile di tipo intero senza segno sono riservati quattro byte, si possono rappresentare i naturali da 0 a  $2^{32} - 1$  (si noti che sono tutti i possibili resti di una divisione per  $2^{32}$ ) e, adottando quindi l'aritmetica *modulo*  $2^{32}$ , una volta superato il più grande, questi numeri si ripetono, come se fossero disposti sul quadrante di un orologio.

Così, ad esempio, se i due numeri da sommare sono 1 e  $2^{32} - 1$ , il risultato restituito è 0: il riporto finale è ignorato; e se ciò costituisse una parte di un calcolo ben più complesso, l'elaborazione continuerebbe, il risultato finale sarebbe probabilmente “sballato”, ma forse ci si dovrebbe pensare un po’ su per accorgersi di che cos’è che non è andato per il verso giusto... Tutto sommato, in molti casi, sarebbe preferibile la segnalazione di un errore (appunto di *overflow*) da parte della macchina!

D'accordo che bisogna evitare errori fatali, incontrollati, che portano all'interruzione (o *aborto*) dell'esecuzione, così come si deve cercare possibilmente di non cadere in *loop* senza fine, ma occorre anche non rischiare di ottenere risultati scorretti che qualcuno potrebbe prendere per buoni e causare guai forse ancora peggiori...

A questo punto non dovrebbe presentare eccessive difficoltà la definizione delle altre operazioni aritmetiche. Ad esempio, per moltiplicare due numeri naturali si dovranno usare tre variabili ausiliarie in due cicli annidati; in linguaggio C possiamo sfruttare la funzione *sum*, abbreviando così la definizione (si veda il codice alla pagina seguente).

Ecco le utilissime idee dell'*astrazione* e dell'*uso* (nonché della *composizione*) di operazioni: definiamo operazioni (che sono funzioni: con i loro parametri-operandi, istanziabili con argomenti che possono essere applicazioni di funzioni) via via più complesse, usando operazioni più semplici definite in precedenza (o la cui definizione, che preciserà *come* calcolarle, è rimandata a un momento successivo).

```

unsigned int prod (unsigned int x, unsigned int y) {
    unsigned int a = 0, b = 0;
    while (b != y) {
        a = sum(a, x);
        b++;
    }
    // (b == y) && (a == x * y) (a meno di overflow...)
    return a;
}

```

Inoltre, quelle della forma vista sono le istruzioni fondamentali (e non soltanto per l’aritmetica, poiché tutto ciò che è codificabile può essere sia rappresentato sia interpretato come numero naturale), i “mattoncini elementari” con i quali possiamo costruire tutte le funzioni che ci servono per risolvere quei problemi la cui soluzione può essere trovata da un programma a calcolatore...

Un’ultima considerazione: qui ci siamo avvalsi di un piccolo linguaggio in stile imperativo. Avremmo potuto usare, invece, un piccolo linguaggio secondo un qualsiasi altro paradigma (funzionale, logico, ...), purché *computazionalmente completo* o, come anche si dice, *universale* (rispetto alla classe delle funzioni effettivamente calcolabili).

I linguaggi di programmazione che comunemente usiamo posseggono una ricchezza di costrutti sintattici che permette definizioni sintetiche (e dunque meglio leggibili) di procedimenti di calcolo pur complessi. Inoltre, essi prevedono la possibilità di creare in memoria oggetti di diversi *tipi*, magari di dimensione prestabilita ma “collegabili” tra loro a formare *strutture di dati* la cui “crescita”, durante una esecuzione del programma, è limitata soltanto dalla memoria fisica della macchina.

## Problemi indecidibili.

Nel precedente capitolo abbiamo esposto la seguente questione: se un programma arbitrario, su un suo input pure arbitrario, prima o poi terminerà... Si tratta di un importante esempio di problema (o, per meglio dire, *predicato*, ossia funzione a *valori di verità*, o *booleani*) *non decidibile* (si intende in modo effettivo, cioè mediante un programma). Noto come “problema dell’arresto”, in realtà è soltanto *semidecidibile*: ciò significa che il massimo che si possa fare equivale a scrivere un programma che prima o poi ci dia la risposta “vero” (o “sì”) tutte le volte che questa è la risposta corretta; invece, in generale, non ci darà alcuna risposta nel caso contrario, ossia quando la risposta giusta dovrebbe essere “falso” (o “no”). Un programma del genere non ci è di grande utilità: se si aspetta e non arriva alcuna risposta, non si sa nulla; il caso in cui non arriverà *mai* risposta non è rilevabile!

Ci sono dei predicati *decidibili*: per ciascuno di essi, si può scrivere un programma che, per ciascun input, prima o poi dia la risposta giusta, “vero” o “falso” che sia. Ciò è indubbiamente utile, ammesso che si abbia il tempo di attendere...

E, come vedremo, l’attesa potrebbe essere assai lunga!

Esempi di questioni decidibili sono: «un dato naturale è un quadrato perfetto», «un dato naturale è un numero primo», «un dato programma scritto in C è sintatticamente corretto» (infatti esistono i compilatori C, che per prima cosa rispondono proprio a quest'ultima domanda).

Si sa che spesso, per risolvere un problema con l'aiuto del computer, esistono metodi diversi che portano a scrivere programmi anche profondamente diversi (che possono pure differire per l'efficienza con cui arrivano alla stessa soluzione); possiamo sempre e comunque cambiare qualcosa, magari qualche nome di variabile, o aggiungere qua e là istruzioni inutili: sicché, in definitiva, i programmi che calcolano una stessa funzione (cioè *funzionalmente equivalenti*) sono un'infinità numerabile... Numerabile sì, ma *non* in modo effettivo: in altre parole, non v'è speranza di programmare un computer affinché sia in grado di dirci se due programmi arbitrari sono funzionalmente equivalenti. Si tratta di una questione che non è nemmeno semidecidibile: si dice anche che è *totalmente indecidibile*.

Un altro esempio di questione totalmente indecidibile è il “problema della totalità”, che concerne la terminazione di un programma arbitrario su *tutti* i suoi possibili input. In effetti, non si può scrivere un programma che prenda in input un programma arbitrario  $p$  e ci dica se  $p$  darà comunque un risultato, qualunque sia il suo input; non vi è neppure speranza, in generale, di ottenere risposta affermativa quando ciò è vero. D'altra parte, è di vitale importanza la possibilità di scrivere programmi che (almeno su certi input) non terminano.

A questo proposito, il lettore ci conceda una breve digressione. Se un programma  $p$  dà sempre un risultato, qualunque sia il suo input, vuol dire che la funzione calcolata da  $p$  è *totale*. Se il linguaggio in stile imperativo introdotto nel precedente paragrafo, al posto del costrutto **while** (che, si noti, permette anche di scrivere programmi che *mai* terminano), disponesse di un comando **for** (presente in certi linguaggi, come il Pascal) per esprimere *soltanto* cicli a numero di iterazioni *sì* variabile in funzione dei dati, ma *precalcolato*, e pure della classica istruzione condizionale **if-then-else**, allora non sarebbe più computazionalmente completo: ci permetterebbe soltanto il calcolo di funzioni totali, ma neppur di tutte le funzioni totali calcolabili!

Un classico controesempio è la *funzione di Ackermann*, scoperta nel 1928, quando Wilhelm F. Ackermann era allievo di Hilbert; una delle sue tante “varianti”, in due variabili naturali, può essere così definita in modo (doppiamente) ricorsivo (A. R. Meyer e D. M. Ritchie, 1967):

$$\begin{aligned} A(0, y) &= 1 && \text{per ogni } y \\ A(1, 0) &= 2 \\ A(x, 0) &= x + 2 && \text{per ogni } x \geq 2 \\ A(x, y) &= A(A(x - 1, y), y - 1) && \text{per ogni } x \text{ e } y \text{ positivi.} \end{aligned}$$

Questa funzione cresce assai rapidamente; infatti si può facilmente provare che (indicando con  $\uparrow$  l'operatore di elevamento a potenza, *con associatività a destra*) dalla definizione data si ottiene:

$A(x, 1) = 2x$	per ogni $x$ positivo
$A(x, 2) = 2 \uparrow x$	per ogni $x$
$A(x, 3) = 2 \uparrow 2 \uparrow \dots \uparrow 2$	dove 2 compare $x$ volte,
$A(1, 4) = 2$	per ogni $x$ positivo
$A(2, 4) = 2 \uparrow 2 = 4$	
$A(3, 4) = 2 \uparrow 2 \uparrow 2 \uparrow 2 = 2 \uparrow 2 \uparrow 4 = 2 \uparrow 16 = 65536$	
$A(4, 4) = 2 \uparrow 2 \uparrow \dots \uparrow 2$	dove 2 compare 65536 volte,

numero che già non possiamo esplicitare: occorrerebbero infatti, per la precisione,  $1 + 2 \uparrow 2 \uparrow \dots \uparrow 2$  cifre binarie, dove questa volta 2 compare 65535 volte!

Ebbene, per calcolare – seppur sulla carta, s'intende! – questa funzione totale, non bastano i cicli **for**.

Un predicato decidibile è, a maggior ragione, anche semidecidibile.

La *negazione* di un predicato decidibile è ancora decidibile: sarà sufficiente invertire le uscite dell'algoritmo che decide il predicato in questione. Ad esempio, se prendiamo l'algoritmo che decide la primalità di un numero e vi aggiungiamo un'istruzione che inverta il risultato, otteniamo un algoritmo che decide la non-primalità, ossia se il numero dato è il prodotto di almeno due fattori primi.

La negazione di un predicato semidecidibile, ma non decidibile, è totalmente indecidibile. Ciò può essere facilmente provato per assurdo: se infatti fosse semidecidibile, potremmo far eseguire “in parallelo” (anche da una sola macchina: un passo l'uno e un passo l'altro) entrambi gli algoritmi, quello che semi-decide il predicato in questione e quello che semi-decide la sua negazione; prima o poi uno dei due darebbe risposta affermativa: se si trattasse del secondo, muteremmo la risposta in negativa. Otterremmo in tal modo un algoritmo per *decidere* il predicato in questione!

Pertanto, ad esempio, il “problema della non-terminazione” (stabilire se un programma arbitrario su un suo input pure arbitrario *non* terminerà) è totalmente indecidibile.

Come può essere la negazione di un predicato totalmente indecidibile? (Ad esempio, si può dimostrare che il “problema della non-totalità” è anch’esso totalmente indecidibile. Provate pure a riflettere su quest’altro problema: decidere se un programma arbitrario dà sempre risultato 0, e poi sulla sua “negazione”.)

Ricordiamo che, quando parliamo di decidibilità, intendiamo sempre in senso algoritmico. In particolare, dunque, i problemi indecidibili sono quelli irrisolvibili mediante un programma.

Dal tempo di Turing in poi sono stati scoperti altri problemi indecidibili, spesso derivati proprio dal problema dell’arresto. Sebbene, di solito, i problemi che sorgono nella pratica abbiano una soluzione (eventualmente di impossibilità), ed esista – almeno in linea di principio – un modo per trovarla, si può dimostrare che i problemi indecidibili costituiscono l’assoluta maggioranza fra tutti quelli che, teoricamente, si potrebbero presentare.

Un esempio famoso di problema provato indecidibile consiste nello stabilire se un'equazione a coefficienti interi con numero di variabili e grado arbitrari ammetta soluzioni intere (Ju. V. Matijasevič, 1970); un altro riguarda la possibilità di ricoprire una superficie rettangolare qualsiasi (di lati interi, grandi a piacere) con piastrelle quadrate di lato unitario scelte in un dato insieme di tipi, con lati di vari colori per ciascun tipo, ma non ruotabili, in modo che non vi siano mai due lati adiacenti di colore diverso (“problema del domino”); ancora un paio di problemi indecidibili saranno menzionati alla fine di questo capitolo, mentre di un altro parliamo adesso.

Scegliamo un linguaggio, a nostro piacere, tra quelli computazionalmente completi. Nel seguito del paragrafo, quando diremo “il linguaggio”, intenderemo proprio quello prescelto, e quando ci riferiremo a un generico programma supporremo – senza perdita di generalità – che sia scritto in tale linguaggio, e che dunque sia codificabile con una sequenza finita di bit, interpretabile da una macchina; anche l’input per il programma sarà rappresentato da una sequenza finita di bit.

Una quarantina d’anni dopo il risultato di Turing – siamo negli anni ’70 – Gregory J. Chaitin si pose un’interessante domanda: qual è la *probabilità* che la macchina si arresti quando le si fa eseguire un programma scelto *a caso* su un input pure scelto *a caso* (pur in infinità numerabili)? Chiamò  $\Omega$  questa probabilità, e dimostrò che  $\Omega$  è un numero (ovviamente maggiore di 0 e minore di 1) *irrazionale* e *non calcolabile*: esso esiste, nel senso che può essere definito in modo preciso, ma nessuna sottosequenza delle sue cifre può essere computata in tempo finito (neanche “a mano”, altrimenti ecco che la tesi di Church sarebbe confutata). Quindi  $\Omega$  non si conosce!

Anche  $\pi$  è irrazionale (è stato provato nella seconda metà del ’700), ma costituisce un esempio di numero irrazionale che si può calcolare, mediante un algoritmo (di lunghezza finita), con una precisione arbitraria (finita), in un tempo finito (che dipende, com’è ovvio, dalla velocità del “calcolatore”):

$$\pi = 3.141592653589793238462643383279502884197\dots$$

Queste quaranta cifre bastano a calcolare la lunghezza di una circonferenza che abbia un diametro dell’ordine di quello dell’universo conosciuto ( $92 \cdot 10^9$  anni luce), supposto precisamente noto, con un errore che non supera il diametro di un atomo!

Non si sa però se la frequenza relativa delle cifre decimali tenda all’uniformità all’aumentare del numero di cifre considerate nello sviluppo di  $\pi$ , e quindi se – in questo senso – la successione possa ritenersi “casuale”. Si pensa di sì, perlomeno quando lo si esprime in cifre binarie, ma il punto focale della questione è un altro: tutta l’informazione contenuta in questa successione infinita di cifre è “condensata” nell’algoritmo che la calcola, e quindi chi riuscisse a capire questa legge sottesa potrebbe scommettere con successo sulle cifre successive. In questo senso, allora, la successione non può più essere ritenuta casuale!

Il “caso”, in effetti, è uno dei temi più difficili ed elusivi, sia in matematica sia in fisica (e in filosofia); raramente, prima di Pascal, si trovano tentativi di affrontare in maniera razionale e scientifica questo argomento, che attualmente occupa una posizione centrale nella *teoria dell’informazione*. Lo stesso concetto di probabilità

ha pochi secoli di vita ed è tuttora spesso bistrattato dal senso comune: si pensi, ad esempio, a ciò che si sente dire sui numeri “in ritardo” nelle estrazioni del lotto! Una sequenza di bit può dirsi casuale se non può essere “compressa” in modo rilevante, vale a dire se non può essere generata (come output) da un programma sensibilmente più corto della sequenza stessa. Questa definizione “algoritmica” di casualità come incomprimibilità fu avanzata negli anni ’60, da Andrej N. Kolmogorov e dallo stesso Chaitin, all’epoca giovanissimo.

In breve, Chaitin ha mostrato come alcuni fatti matematici non abbiano ridondanza e siano troppo complicati (o meglio, infinitamente complessi) per poter essere compressi in una teoria. In seguito ha scoperto che queste idee affiorano già da un saggio di Gottfried Wilhelm Leibniz del 1686, *Discours de métaphysique*, dove il filosofo tedesco si chiede come possiamo distinguere tra fatti che seguono una legge e fatti irregolari, casuali. Per inciso, sette anni prima Leibniz aveva introdotto l’aritmetica binaria e menzionato la possibilità di meccanizzarla.<sup>11</sup> L’idea di Leibniz è semplice e profonda a un tempo: un insieme di dati non casuali deve poter essere descritto da una legge più semplice dei dati stessi. Da cui: una teoria è utile quando la sua dimensione in bit è inferiore a quella dei dati che essa spiega. E oltre tre secoli prima di Leibniz, Guglielmo di Occam aveva asserito che la teoria più semplice è la migliore. In altre parole, dobbiamo cercare il programma più breve che generi quei dati: la sua lunghezza è il contenuto di *informazione algoritmica* dei dati.

“Moltissimi” numeri reali sono casuali in senso algoritmico; ma pur limitandosi – per così dire – agli interi, la maggior parte dei numeri risultano casuali – troppo pochi sono infatti i programmi “brevi”! – sebbene non si possa dimostrare per via automatica che un numero arbitrario lo sia! Più precisamente: soltanto di un numero finito di essi può essere dimostrata la casualità in un sistema formale coerente; infatti, se assiomi e regole possono essere descritti con  $n$  bit complessivi, allora non è possibile dimostrare la casualità di un numero rappresentato da una quantità di bit abbastanza più grande di  $n$ . E questo equivale alla non decidibilità del problema dell’arresto.

Dalle proprietà di  $\Omega$  che abbiamo enunciato, discende dunque che questo numero “cabalistico” è *casuale* in senso forte: non è “ridondante” e non si può scommettere meglio che alla pari sulla successione delle sue cifre (almeno sul lungo periodo).

Chaitin, con un affascinante ragionamento, ci mostra che, qualora si potessero conoscere anche poche migliaia delle prime cifre di  $\Omega$ , si potrebbe trovare – almeno in linea di principio – una maniera per risolvere la maggior parte delle questioni notoriamente rimaste aperte in matematica, in particolare quelle concernenti le proposizioni che, se fossero false, potrebbero essere refutate in un numero finito di passi: ad esempio la *congettura di Goldbach* o l’asserzione che un qualche enunciato è *indipendente* da un dato insieme di assiomi, cioè che non può essere né dimostrato né confutato a partire da essi.

---

<sup>11</sup> Tuttavia, la macchina calcolatrice da lui progettata nel 1673 – ma finita successivamente, nel 1694 – operava su base decimale.

La nota congettura rimasta legata al nome di Christian Goldbach, semplicissima da enunciare (ogni numero pari maggiore di 2 è somma di due numeri primi), resiste dal 1742 ai tentativi di dimostrazione o di demolizione. Ammesso che sia vera, non è detto tuttavia che ne esista una dimostrazione a partire dagli assiomi dell'aritmetica; d'altra parte, se si riuscisse a dimostrare che è indecidibile, ecco che si sarebbe dimostrata la sua verità: infatti, ciò vorrebbe dire che non esiste un controesempio che un automa potrebbe prima o poi trovare!

Vi sono esempi famosi di enunciati indipendenti da un insieme di assiomi. Mentre comunemente è accettata la necessità del platonico principio di non-contraddizione o di consistenza (un enunciato e la sua negazione non possono essere entrambi veri, altrimenti qualsiasi affermazione sarebbe vera: *ex falso quodlibet*), nella logica intuizionista si fa a meno di uno schema di assioma più generale, il principio del terzo escluso (o un enunciato è vero o è vera la sua negazione: *tertium non datur*).

Nella matematica costruttivista non si dà per scontata l'esistenza di una funzione che ad ogni insieme di una famiglia non vuota di insiemi non vuoti faccia corrispondere un suo elemento (*assioma di scelta*, che Gödel provò essere indipendente dalla teoria degli insiemi di Zermelo-Fraenkel), così come nelle geometrie non euclidee si nega che, sul piano, per ogni punto non appartenente a una retta data passi una e una sola parallela a tale retta (*postulato delle parallele*).

## Algoritmi del passato... ma sempre attuali.

Come furono rappresentati gli algoritmi prima dell'avvento della teoria della calcolabilità e dei linguaggi di programmazione? I più antichi procedimenti di calcolo a noi noti risalgono alla Mesopotamia di circa una quarantina di secoli fa: però si tratta soltanto di sequenze di operazioni su particolari dati, prive di una descrizione della procedura astratta da eseguirsi in generale.

Bisogna giungere alla civiltà greca del III secolo a. C. per trovare parecchi algoritmi non banali definiti in astratto, senza riferirsi a casi specifici, ma ancora in linguaggio naturale, cioè informalmente: per fare soltanto due esempi, ricordiamo il calcolo del massimo comun divisore descritto da Euclide e il crivello di Eratostene per lasciare i soli numeri primi, partendo dalla successione dei naturali maggiori di 1.

Dal lontano passato emergono metodi per fare certe operazioni che talvolta sono più efficienti di quelli che di solito usiamo o che si insegnano a scuola (invito il lettore a fare una ricerca, ad esempio sulla moltiplicazione); spesso di tali tecniche si sono serviti i progettisti delle macchine per calcolare, dalle prime calcolatrici agli elaboratori elettronici.

Erone di Alessandria fu uno dei più grandi ingegneri e sperimentatori dell'antichità; i suoi progetti di macchine azionate da pesi o da mezzi idraulici o pneumatici (che sfruttavano il vapore o il vento), nonché i risultati da lui ottenuti in varie scienze (quali ottica, geometria e matematica), ci sono pervenuti attraverso manoscritti arabi o versioni latine.

Ai fini del nostro discorso, ci interessano le *formule ricorrenti* che egli impiegò per calcolare radici quadrate e cubiche, plausibilmente non originali ma riprese da quella civiltà babilonese che, insieme con i lavori di Ctesibio di Alessandria e Filone di Bisanzio e le opere di Euclide e Archimede, fu certamente fonte d’ispirazione per le sue idee.

Queste formule troveranno la loro giustificazione, molti secoli più tardi, nel “metodo delle tangenti”, concepito da Newton e poi generalizzato da Raphson e da Simpson, per calcolare – in maniera approssimata – gli zeri di una funzione derivabile.

Sia  $q$  il numero (reale) positivo di cui si vuole calcolare la radice quadrata (approssimata). Si parte con  $x_0 > 0$ , preferibilmente prossimo al risultato, ma ciò non è essenziale; ad esempio si può scegliere  $x_0 = 1$  oppure  $x_0 = q/2$ . Si applica quindi la formula ricorrente:<sup>12</sup>

$$x_{n+1} = (x_n + q/x_n)/2$$

Quando ci si può fermare? In generale, un criterio di arresto sensato consiste nel valutare quanto ci si è scostati con l’ultimo passo dal valore precedente, fermandosi se tale scostamento è inferiore a una certa (piccola) percentuale del valore precedente. In altre parole, si accetta una certa *tolleranza relativa*; la condizione che decreta la fine del calcolo ha dunque la forma:

$$|(x_{n+1} - x_n)/x_n| < \varepsilon$$

Per evitare troppi calcoli, si potrebbe pensare di iterare la formula per un numero prestabilito di volte: è un criterio altrettanto plausibile? Prima di rispondere a questa domanda, diciamo qualcosa sul formato dei valori numerici manipolati da un elaboratore elettronico, e illustriamo nei dettagli un altro metodo iterativo per calcolare la radice quadrata.

Se in un computer è previsto un certo numero di bit per memorizzare un valore “reale”, in verità è rappresentabile soltanto un sottoinsieme finito dei razionali, noto a priori. Per fissare le idee, supponiamo di prevedere 22 bit, di cui: uno per il segno del numero ( $0 = +$ ,  $1 = -$ ), 7 per l’esponente (intero) di 2, e 14 per la parte frazionaria (detta *mantissa*). L’esponente potrà assumere valori da  $-64$  a  $+63$  (magari adottando per i negativi l’utile codifica in “complemento a 2”), mentre prima delle cifre della mantissa può essere sottintesa la cifra 1 (“bit nascosto”) seguita dal punto. Tutte le configurazioni con esponente  $-64$  rappresentano lo zero; tutte quelle con esponente  $+63$  sono considerate  $+\infty$  o  $-\infty$  a seconda del segno del numero, e comunque assimilate all’effetto di un *overflow*.

(In queste ipotesi, quanti sono i numeri positivi, o i negativi, rappresentabili? E come risultano distribuiti sulla retta dei reali?)

<sup>12</sup> Questa è la formula per la radice quadrata; per ottenere un’approssimazione della radice cubica basta cambiare l’espressione alla destra del segno di uguaglianza: al doppio di  $x_n$  si somma il rapporto tra  $q$  e il quadrato di  $x_n$ ; poi si divide il risultato per 3.

Questo è proprio ciò che fu fatto per la prima volta nello Z3, il primo computer digitale (a relè) controllato da programma (esterno, a sola lettura), messo in funzione a Berlino nel 1941 dal suo creatore, l'ingegnere civile tedesco Konrad Zuse (1910-1995); purtroppo andò distrutto durante un bombardamento dopo nemmeno tre anni. Zuse chiamò questa rappresentazione “semilogaritmica” (il motivo è spiegato dalla risposta all’ultima domanda posta in fondo alla pagina precedente, che lasciamo scoprire al lettore); noi oggi parliamo di “virgola mobile” (*floating point*) e gli attuali standard IEEE sono del tutto simili, ma l’idea di codificare in questo modo i dati numerici da elaborare, allo scopo di mantenere il più possibile costante l’errore relativo commesso nelle approssimazioni, risale al 1914 ed è dovuta a un altro ingegnere civile, l’inventore spagnolo Leonardo Torres y Quevedo (1852-1936).

Tra le tante cose notevoli, Zuse realizzò nello Z3 un algoritmo efficiente proprio per calcolare la radice quadrata, un vero gioiellino basato su un altro metodo che viene dal passato: lo descrisse infatti, intorno alla metà del XVI secolo, il matematico bolognese Rafael Bombelli per i numeri nell’usuale notazione decimale, ed è noto come “completamento del quadrato”. Vediamo come Zuse lo adattò al suo computer binario.

Senza perdita di generalità, possiamo assumere che il radicando  $q$  sia  $\geq 1$  e  $< 4 = 100_2$ ; qualsiasi numero positivo può infatti essere scritto nella forma  $q \cdot 2^p$ , con  $p$  pari e  $q$  (come si è detto)  $\geq 1$ , e quindi alla fine basterà moltiplicare il risultato ottenuto per il fattore  $2^{p/2}$ . Dunque, poiché la parte intera di  $q$  è 1 o 2 o 3, la radice quadrata di  $q$  avrà 1 come parte intera, cioè sarà  $\geq 1$  e  $< 2 = 10_2$ .

Fissato  $n$ , numero di bit della mantissa, e dato  $q$  come sopra, l’idea per calcolare la radice quadrata di  $q$  consiste nel costruire il risultato un bit alla volta, partendo dal più significativo, in  $n$  passi (al più). La variabile in cui è calcolato il risultato  $r$  è inizializzata al valore 1, corrispondente al bit nascosto che precede il punto (sottinteso) e la mantissa (con tutti i suoi bit a 0). Ecco l’algoritmo:

```

 $r \leftarrow 1$ 
 $e \leftarrow q - 1$ 
se  $e = 0$  allora FINE
per  $k = 1, 2, \dots, n$ :
   $b \leftarrow 2 \cdot e - (2 \cdot r + 2^{-k})$ 
  se  $b \geq 0$  allora // il  $k$ -esimo bit della mantissa è messo a 1
     $r \leftarrow r + 2^{-k}$ 
     $e \leftarrow b$ 
  altrimenti // il  $k$ -esimo bit della mantissa è lasciato a 0 ...
     $e \leftarrow 2 \cdot e$  // ... e comunque  $2^{-k} \cdot e$  non aumenta!
  se  $e = 0$  allora FINE

```

Ogni volta che  $e$  è assegnato (sia prima di entrare nel ciclo, sia al termine di ciascuna iterazione), se il suo valore è 0 allora l’algoritmo può terminare: qualora si verifichi una tale eventualità, il risultato  $r$  sarà esatto, altrimenti sarà approssimato per difetto.

Si noti che, al termine della  $k$ -esima iterazione, il “resto”  $2^{-k} \cdot e$  è minore o uguale a quello che si aveva all’inizio della medesima iterazione. Alla fine,  $e$  è l’errore “scalato”, poiché si ha:

$$r^2 + 2^{-n} \cdot e = q$$

Nello Z3, come si è detto,  $n$  valeva 14 e comunque il sequenziatore di micro-operazioni era in grado di compiere il necessario numero di iterazioni.

Vediamo un esempio, assumendo per brevità  $n = 8$ . Vogliamo calcolare la radice quadrata di  $54.125 = 110110.001_2 = 1.10110001 \cdot 2^5$ . Tale numero sarà dunque rappresentato precisamente da questi 16 bit (uno per il segno, 7 per l’esponente e 8 per la mantissa, il bit nascosto non compare): 0 0000101 10110001.

L’esponente di 2 è 5; essendo dispari si deve portare all’unità inferiore, cioè 4:  $11.0110001 \cdot 2^4$ , e quindi:  $q = 11.0110001$ ,  $r = 1.00000000$ ,  $e = q - 1 = 10.0110001$ . Qui i passi si fanno tutti e otto, perché non accade che  $e$  si azzeri:

$k$	$2 \cdot e - (2 \cdot r + 2^{-k}) = b$ [in neretto: il nuovo valore di $e$ ...]	$r$ [... e i bit calcolati]
1	$100.110001 - 10.1 = \mathbf{10.010001} (> 0)$	1.10000000
2	$100.10001 - 11.01 = \mathbf{1.01001} (> 0)$	1.11000000
3	$\mathbf{10.1001} - 11.101 (< 0)$	1.11000000
4	$101.001 - 11.1001 = \mathbf{1.1001} (> 0)$	1.11010000
5	$\mathbf{11.001} - 11.10101 (< 0)$	1.11010000
6	$110.01 - 11.101001 = \mathbf{10.100111} (> 0)$	1.11010100
7	$101.00111 - 11.1010101 = \mathbf{1.1000111} (> 0)$	1.11010110
8	$\mathbf{11.000111} - 11.10101101 (< 0)$	1.11010110

Il risultato finale, cioè la radice quadrata di 54.125 (approssimata per difetto), è dunque  $r \cdot 2^2$  (rappresentato con 0 0000010 11010110), quello esatto sarebbe:

$$2^2 \cdot \sqrt{q} = 2^2 \cdot \sqrt{(r^2 + 2^{-8} \cdot e)}$$

Si noti che in questo caso sarebbe stata più precisa l’approssimazione per eccesso, con l’ultimo bit di  $r$  messo a 1: varrebbe infine la pena di fare un controllo (previo calcolo di un bit in più) e, se opportuno, un aggiustamento del risultato.

Il procedimento che abbiamo esposto è assai economico, poiché usa soltanto le operazioni di sottrazione, moltiplicazione per 2 (corrispondente a uno *shift* a sinistra con inserimento di uno 0), cambiamento di un bit da 0 a 1, e confronti con lo zero.

Anche per l’operazione di divisione tra due numeri, in genere piuttosto costosa, Zuse realizzò un algoritmo simile, noto come *non-restoring division*. Sicché il tempo richiesto per una radice quadrata era di circa 4 secondi, quello per una divisione poco meno (lo Z3 era temporizzato a una frequenza di circa 5 Hz); tanto per fare un paragone, gli odierni *smartphone* sono 300 milioni di volte più veloci – ovvero, mentre il primo percorre un metro o poco più, il secondo va dalla Terra alla Luna!

Torniamo alla formula di Erone, mantenendo le stesse convenzioni. Nel caso preso ad esempio, partendo con  $x_0 = q/2$  bastano due iterazioni per giungere allo stesso risultato:

$$\begin{aligned}x_1 &= (x_0 + 2)/2 &= 1.11011000 \\x_2 &= (x_1 + q/x_1)/2 &= 1.11010110\end{aligned}$$

(anche il quoziente  $q/x_1$  è stato approssimato per difetto). *Idem* se si parte con  $x_0 = 3/2$ , centro dell'intervallo in cui cadrà il risultato.

Ma siamo certi che due iterazioni siano sempre sufficienti a raggiungere la massima precisione consentita dalle 8 cifre della mantissa? In caso contrario dovremmo continuare, fino ad accorgerci che un'iterazione non ha cambiato nulla, ossia che la condizione  $x_{k+1} = x_k$  è vera... Potrebbe nascondere qualche insidia questo criterio d'arresto?

Adattiamo al nostro problema un altro metodo ben noto per trovare un punto di zero di una funzione in un intervallo dove essa sia continua e cambi segno: il metodo di *bisezione* (o *dicotomico*) che ad ogni iterazione dimezza l'intervallo di incertezza.

Nella fattispecie ci interessa la funzione  $x^2 - q$  nell'intervallo  $[1, 2]$ , ov'è crescente; infatti, escluso il caso banale  $q = 1$ , la radice di  $q$  è  $> 1$  e  $< 2$ , e la funzione è negativa in 1, positiva in 2.

Al fine di ottenere la massima accuratezza, adottiamo questo algoritmo:

```
se  $q = 1$  allora  $r \leftarrow 1$ . FINE
 $x \leftarrow 1$ 
 $h \leftarrow 1/2$ 
 $r \leftarrow 3/2$ 
ciclo continuo:
  se  $r \cdot r - q < 0$  allora  $x \leftarrow r$ 
   $h \leftarrow h/2$ 
   $r \leftarrow x + h$ 
  se  $r = x$  allora FINE
```

Se lo applichiamo al caso dell'esempio, arriviamo di nuovo allo stesso risultato in 8 iterazioni del ciclo, come con l'algoritmo di Zuse; là tuttavia le operazioni per ciascun passo erano meno costose in termini di tempo.

Affinché il procedimento abbia termine,  $h$  deve ridursi a un valore tanto piccolo da non alterare quello di  $x$  quando vi è sommato. Ciò prima o poi accade, dato che ad ogni passo  $h$  si dimezza, purché il formato dei numeri in virgola mobile permetta di scendere a un esponente relativamente basso o preveda di associare il valore 0 alle configurazioni con esponente più piccolo, come comunemente avviene; altrimenti sì che potrebbe sorgere un problema!

Se ad esempio il minore degli esponenti fosse  $-8$ , e  $2^{-8}$  non rappresentasse lo zero, bensì venisse approssimato con sé stesso quando è diviso per 2, allora all'inizio dell'ottava iterazione del ciclo si avrebbe:

$$x = 1.11010110, \ h = 2^{-8}, \ r = 1.11010111$$

e durante questa iterazione... nessuno dei valori di tali variabili cambierebbe! Il calcolo cadrebbe così in un *loop* senza fine. Si può dunque intuire l'importanza della scelta delle configurazioni numeriche, del loro significato e dei criteri per la loro manipolazione.

La *convergenza* alla radice quadrata è lenta, sia per l'algoritmo di Zuse, sia per il metodo di bisezione; infatti è *lineare*: ad ogni iterazione del ciclo si guadagna una cifra binaria. La formula di Erone ha invece convergenza *quadratica*: ad ogni passo il numero di cifre corrette tende a raddoppiare. Tuttavia l'algoritmo di Zuse rimane attraente per l'efficienza con cui si riescono a realizzare, all'interno di un computer, le semplici operazioni coinvolte.

Vorrei concludere questo paragrafo aggiungendovi qualche ulteriore, importante considerazione sullo Z3, giacché se n'è parlato.

Come già concepito da Charles Babbage un secolo prima, e con anni di anticipo rispetto alle raccomandazioni di John von Neumann sull'architettura ideale per un computer, lo Z3 era dotato di unità di calcolo e di controllo separate dalla memoria. Pure di Babbage erano già state le idee di un particolare meccanismo per accelerare la propagazione dei riporti nell'esecuzione delle addizioni, e soprattutto del controllo dei calcoli mediante un programma esterno. Zuse riuscì inoltre a realizzare i formati di istruzione con indirizzi di memoria numerici (a 6 bit), in origine proposti dall'irlandese Percy E. Ludgate, nel 1909.

In Germania, Zuse è reputato l'inventore del computer, e certamente uno degli inventori lo è stato; tanto più che, almeno in linea di principio, lo Z3 può essere considerato la prima macchina funzionante *Turing-completa*. Infatti: leggeva le istruzioni da una pellicola perforata (ciascuna istruzione era costituita da 8 bit, disposti su due linee sfasate) che – incollandone le estremità – avrebbe potuto essere chiusa a formare un nastro continuo; inoltre possedeva un hardware (per inciso, realizzato in modo elegante ed economico, con pochi relè) dedicato al rilevamento di operazioni con risultato indeterminato e alla conseguente fermata della macchina. Ebbene, è stato dimostrato (R. Rojas, 1998) che si può simulare una qualsiasi macchina di Turing con un unico ciclo di sole operazioni aritmetiche (su operandi presi dalla memoria, e trasferimento del risultato in memoria), ammesso che si disponga di un'operazione che eccezionalmente causi l'arresto (0/0 nello Z3) e di un sufficiente numero di locazioni di memoria indirizzabili in modo diretto.

## Grammatiche, linguaggi e automi.

Nel prosieguo del capitolo accenniamo alla formalizzazione dei linguaggi (s'intende “artificiali”, come quelli di programmazione), alle macchine che li riconoscono e ad alcuni problemi, talvolta indecidibili, legati sia alle loro caratteristiche intrinseche sia al loro trattamento automatico.

Quando diciamo *alfabeto* intendiamo un insieme finito e non vuoto di simboli, detti *terminali*.

Se  $T$  è un alfabeto, indichiamo con  $T^*$  l'insieme delle sequenze di lunghezza finita costituite da elementi di  $T$  (dette *stringhe su  $T$* ). Gli elementi di  $T^*$  sono dunque un'infinità numerabile. Un qualsiasi sottoinsieme di  $T^*$  è un *linguaggio su  $T$* .

Una domanda assai interessante è questa: se consideriamo un linguaggio infinito (cioè costituito da un'infinità di stringhe) – ad esempio: i romanzi che possono essere scritti rispettando la grammatica della lingua italiana oppure i programmi sintatticamente corretti che possono essere scritti in C, lunghi a piacere e a prescindere dal loro significato – riusciamo a descriverlo per mezzo di un insieme finito (e ragionevolmente piccolo) di *regole grammaticali*? Proprio questa è l'idea di “grammatica”... ma la risposta a tale domanda è in molti casi negativa!

Cominciamo con un semplice esempio. Come simboli dell'alfabeto prendiamo le cifre binarie; vogliamo descrivere l'insieme dei numeri naturali scritti in notazione binaria, senza zeri non significativi in testa. A questo scopo ci possiamo servire di una *espressione regolare*:

$$0 \mid 1(0|1)^*$$

che appunto rappresenta l'insieme costituito dalla stringa 0 e da tutte le stringhe di bit che iniziano con 1 (seguito da una sequenza di bit lunga a piacere, anche vuota).

Le espressioni regolari sono basate su un formalismo definito da Kleene intorno alla metà degli anni '50 del secolo scorso.

Abbiamo assunto che l'operatore  $*$  (che denota la cosiddetta *chiusura di Kleene*) abbia priorità su quello di concatenazione (omesso), e che questo a sua volta abbia priorità sull'operatore di scelta alternativa  $\mid$ . È per questa ragione che abbiamo dovuto usare le parentesi.

Allora l'espressione  $0|1^*$  rappresenta il linguaggio

$$\{ 0, \varepsilon, 1, 11, 111, \dots \} = \{ 0 \} \cup \{ 1^n \mid n \geq 0 \},$$

mentre l'espressione  $0^*|1^*$  descrive l'insieme

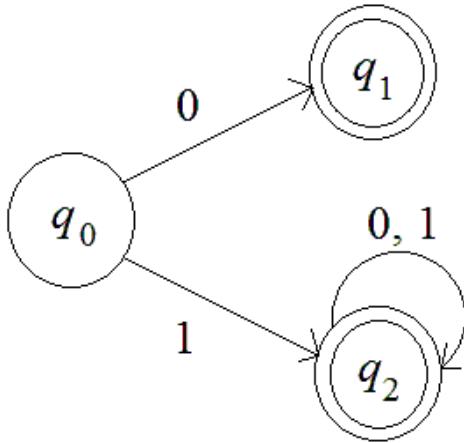
$$\{ \varepsilon \} \cup \{ 0^n \mid n > 0 \} \cup \{ 1^n \mid n > 0 \},$$

dove  $\varepsilon$  denota la stringa vuota. Invece l'espressione  $0^*1^*$  rappresenta l'insieme delle stringhe formate da una sequenza di 0 seguita da una sequenza di 1, anche di lunghezza nulla:

$$\{ 0^m 1^n \mid m \geq 0 \text{ e } n \geq 0 \}.$$

Un *automa a stati finiti* (deterministico) che riconosce l'insieme dei numeri naturali scritti in binario è rappresentato dal diagramma disegnato alla pagina seguente.

Possiamo convenire di indicare sempre con  $q_0$  lo stato *iniziale*. In generale, quando la stringa è terminata, lo stato corrente dev'essere uno degli stati *finali* (doppiamente cerchiati), altrimenti il riconoscimento fallisce. Fallisce pure se la stringa non è terminata, ma dallo stato corrente non esce alcun arco etichettato col prossimo simbolo letto nella stringa.



Provate a disegnare un automa che riconosca le stringhe con un numero dispari di 1; poi provate a disegnare un automa che riconosca le stringhe con un numero pari di 0 e un numero pari di 1.

A un'espressione regolare può sempre essere associato un automa a stati finiti (deterministico) che riconosca il linguaggio (regolare) da essa descritto.

Un'espressione regolare può anche essere trasformata in una *grammatica*, ossia in un insieme finito di regole di forma particolare. Nel caso dei numeri in binario:

$$\begin{aligned} S &\rightarrow 0 \mid 1D \\ D &\rightarrow \epsilon \mid 0D \mid 1D \end{aligned}$$

La prima linea contiene l'abbreviazione di *due* regole grammaticali: una stringa del linguaggio è formata da 0, oppure da 1 seguito da D, dove (si vedano le *tre* regole in seconda linea) D è costituito dalla stringa vuota, oppure da 0 seguito da D, oppure da 1 seguito da D, dove D... Le ultime due regole sono ricorsive (a destra). Tale grammatica può essere riformulata eliminando la regola  $D \rightarrow \epsilon$  e aggiungendone altre tre: quali?

I simboli S e D, che non appartengono all'alfabeto, si dicono *non terminali*; S, quello da cui si parte per l'analisi sintattica (dall'alto al basso) di una stringa, è detto *distintivo* o *iniziale*.

Le grammatiche che hanno soltanto regole di questo tipo, in cui è previsto che ciascun simbolo non terminale debba essere sostituito dalla stringa vuota, oppure da un terminale, oppure da un terminale seguito da un non terminale, sono dette *regolari* e sono in grado di descrivere la stessa classe di linguaggi rappresentati dalle espressioni regolari e riconosciuti dagli automi a stati finiti.

Le regole grammaticali sono anche dette “di produzione”, o semplicemente *produzioni*: infatti possono essere viste come *regole di riscrittura*. Il linguaggio *generato* da una grammatica è costituito esattamente dalle stringhe di soli simboli terminali che si possono ottenere (o, come si dice, *derivare*) a partire dal simbolo distintivo applicando le regole di riscrittura. Una stringa sì derivabile, ma contenente qualche non terminale, si dirà *forma di frase*.

Vediamo altri due esempi di grammatiche regolari, prive di regole che producono  $\epsilon$ :

$$\begin{array}{l} S \rightarrow 0A \\ A \rightarrow 0A \mid 1B \mid 1 \\ B \rightarrow 1B \mid 1 \end{array}$$

$$\begin{array}{l} S \rightarrow 0A \mid 0B \\ A \rightarrow 1A \mid 1 \\ B \rightarrow 2B \mid 2 \end{array}$$

Nel secondo caso, l'alfabeto comprende anche la cifra 2. Nel primo caso, lasciando inalterato il linguaggio, si possono eliminare due regole e aggiungerne una che produce  $\epsilon$ : di quali si tratta?

Quali sono i linguaggi generati? Come possono essere disegnati due automi per riconoscerli?

### Le grammatiche libere dal contesto.

Estendiamo la classe delle grammatiche regolari, non ponendo alcuna restrizione alla forma della parte di regola a destra della freccia (a sinistra abbiamo sempre un solo simbolo non terminale): otteniamo così le grammatiche *libere dal contesto* (o semplicemente “libere”), le quali devono il loro nome proprio al fatto che ciascuna regola impone che il simbolo non terminale a sinistra della freccia possa sempre essere sostituito dalla parte a destra, quali che siano gli eventuali simboli (terminali o non) che lo precedono o lo seguono in una forma di frase in cui compare.

Queste grammatiche sono davvero più potenti: sono in grado di generare dei linguaggi che non possono essere descritti da grammatiche regolari.

Un automa a stati finiti non ha memoria ausiliaria; può tener traccia dei simboli letti soltanto passando da uno stato a un altro, ma il numero degli stati è finito! Ad esempio, non riesce a riconoscere il linguaggio costituito da una sequenza di 0 di lunghezza arbitraria, seguita da una sequenza di 1 *di uguale lunghezza*, cioè  $\{ 0^n 1^n \mid n \geq 0 \}$ , cosa che invece potrà fare agevolmente se lo si doterà di una *pila* potenzialmente illimitata, con accesso al solo oggetto posto sulla cima (cioè l'ultimo che vi è stato inserito). Procederà seguendo queste semplici direttive: ogni 0 letto è inserito sulla pila, quando inizierà la sequenza di 1 non si potranno più leggere 0, e dovrà essere possibile togliere dalla pila uno 0 per ciascun 1 letto. Una stringa è *accettata* (cioè riconosciuta come appartenente al linguaggio) se la pila è vuota quando non ci sono più simboli da leggere.

È immediato constatare che una macchina del genere, detta *automa a pila*, possa fare di più. Ad esempio, accettare una stringa in cui dopo un 1 possano esservi altri 0: ogni 0 letto è inserito sulla pila, e ogni volta che un 1 è letto si dovrà poter togliere uno 0 dalla cima della pila. Soltanto alla fine la pila dovrà restare vuota. Se cambiamo 0 con la parentesi aperta e 1 con la parentesi chiusa, ecco che abbiamo un riconoscitore di sequenze di parentesi ben bilanciate, e non solo della forma  $()$  oppure  $((()))$ , ma anche, tanto per esemplificare,  $(())()$  oppure  $(((()))()$ . Stringhe della forma  $)()$  oppure  $(())()$  non sono invece accettate, sebbene le parentesi aperte siano tante quante le chiuse.

Le seguenti grammatiche, con un solo simbolo non terminale (il distintivo), descrivono i due linguaggi visti:

$$S \rightarrow \varepsilon | 0S1$$

$$S \rightarrow \varepsilon | (S)S$$

Se dal linguaggio si vuole escludere la stringa vuota, la grammatica si allunga soltanto un poco; rispettivamente:

$$S \rightarrow 01 | 0S1$$

$$S \rightarrow () | ()S | (S) | (S)S$$

Servendoci di quest'ultima grammatica, possiamo derivare dal simbolo distintivo la stringa

$$((())())()$$

nel seguente modo:

$$S \xrightarrow{3} (S) \xrightarrow{4} ((S)S) \xrightarrow{2} (((S)S)S) \xrightarrow{1} (((())S)S) \xrightarrow{1} (((())())S) \xrightarrow{1} (((())())())$$

(per ogni passo di riscrittura, accanto alla freccia è riportato il numero d'ordine della regola applicata, avendo numerato le regole da 1 a iniziare da sinistra). Abbiamo deciso di *riscrivere ad ogni passo il non terminale più a sinistra*: così facendo, non ci siamo mai trovati davanti a scelte alternative che ci consentissero di arrivare ugualmente al nostro obiettivo. Tuttavia, se nel linguaggio ci fosse anche una sola stringa derivabile in almeno due modi diversi, sempre seguendo questa prassi, allora la grammatica si direbbe *ambigua*. La grammatica da noi considerata non lo è, ma se – ad esempio – vi aggiungessimo come quinta regola

$$S \rightarrow S()$$

*senza che il linguaggio ne sia alterato*, allora per la stringa poc'anzi considerata si avrebbero altre tre possibili derivazioni, una delle quali è

$$S \xrightarrow{3} (S) \xrightarrow{4} ((S)S) \xrightarrow{5} ((S())S) \xrightarrow{1} (((())S)S) \xrightarrow{1} (((())())S) \xrightarrow{1} (((())())())$$

(le altre due si ottengono anticipando al secondo passo di riscrittura l'applicazione della nuova regola).

Non si pensi che l'ambiguità sia sempre causata da qualche regola ridondante; pur non contemplando regole inutili, la grammatica

$$S \rightarrow () | (S) | SS$$

genera ancora lo stesso linguaggio ed è ambigua: verificarlo è assai facile.

Questo esempio rivela l'esistenza di linguaggi descrivibili tramite grammatiche diverse, alcune ambigue e altre no.

Esistono anche linguaggi *inherentemente ambigi*, che non possono essere descritti da alcuna grammatica non ambigua, ad esempio  $\{ 0^i 1^j 2^k \mid i = j \text{ o } j = k \}$ .

Sorge spontanea una domanda: si può scrivere un programma che, data una grammatica libera, ci dica se essa è ambigua oppure no? La risposta è negativa: questo è un ulteriore esempio di problema indecidibile. Come quello dell'arresto, è soltanto semidecidibile: infatti, se una grammatica è ambigua, un automa è in grado di esibire prima o poi una stringa derivabile in due modi diversi (e quindi, in definitiva, con due strutture sintattiche diverse) a partire dal simbolo distintivo e sempre riscrivendo ad ogni passo il non terminale più a sinistra.

Le grammatiche libere rivestono grande importanza nella definizione dei costrutti sintatticamente corretti dei più comuni linguaggi di programmazione. Consideriamo, ad esempio, il semplice linguaggio introdotto all'inizio del capitolo, cambiando però la sintassi dell'istruzione di incremento: scriveremo  $v++$  anziché  $v = v + 1$  (il motivo sarà presto chiarito). Input/output (acquisire valori iniziali ed esplicitare valori finali per certe variabili) a parte, un programma è descritto dalla seguente grammatica:

Programma	$\rightarrow$	Sequenza
Sequenza	$\rightarrow$	$\epsilon$   Istruzione Sequenza
Istruzione	$\rightarrow$	Variabile = 0   Variabile ++   <b>while</b> (Condizione) { Sequenza }
Condizione	$\rightarrow$	Variabile != Variabile

La categoria (lessicale) dei nomi di variabile può essere definita con l'espressione regolare

$$\text{Lettera} (\text{Lettera} \mid \text{Cifra})^*$$

per denotare l'insieme delle parole, composte da lettere e cifre, che iniziano con una lettera.

Si noti che il corpo di un ciclo o persino un intero programma possono essere vuoti! Sotto l'aspetto sintattico, un usuale linguaggio di programmazione è assai più ricco; ciononostante, la sua sintassi è di solito definita da una grammatica piuttosto contenuta, comprensiva di un centinaio di categorie o poco più.

### Una grammatica per le espressioni aritmetiche col suo *parser*.

Nei più comuni linguaggi di programmazione imperativi, una *istruzione di assegnazione* ha la forma  $v = e$ , dove  $e$  è un'espressione e  $v$  il nome di una variabile; al momento dell'esecuzione, il valore di  $e$  sarà calcolato e memorizzato nella locazione che corrisponde a  $v$  (se tutto va bene).

Semplificando assai, un'espressione aritmetica – il cui valore potrà essere assegnato a una variabile di tipo numerico – è descritta da questa grammatica:

$$\text{Espressione} \rightarrow \text{Variabile} \mid \text{Numero} \mid \text{Espressione Operatore Espressione} \mid (\text{Espressione})$$

$$\text{Operatore} \rightarrow + \mid - \mid * \mid /$$

In tal modo è sì descritta la forma astratta di un'espressione (con operatori binari, in notazione *infissa*), però questa grammatica è ambigua, poiché ad esempio la stringa  $2 + 3 * 4$  è derivabile in due modi diversi, pur riscrivendo ad ogni passo il non terminale più a sinistra; abbreviando:

$$E \rightarrow_3 E O E \rightarrow_3 E O E O E \rightarrow_2 N O E O E \rightarrow \dots \rightarrow 2 + 3 * 4$$

$$\begin{aligned} E &\rightarrow_3 E O E \rightarrow_2 N O E \rightarrow \dots \rightarrow 2 + E \rightarrow_3 2 + E O E \rightarrow_2 2 + N O E \rightarrow \dots \\ &\quad \rightarrow 2 + 3 * 4 \end{aligned}$$

La prima derivazione sottintende l'associatività a sinistra degli operatori; la seconda, invece, quella a destra: le due strutture sintattiche corrispondono, rispettivamente, alle espressioni  $(2 + 3) * 4$  e  $2 + (3 * 4)$ .

Di solito, inoltre, a differenti strutture sintattiche sono associati significati diversi: nel nostro caso 20 e 14, rispettivamente.

Se, in assenza di parentesi, vogliamo mantenere l'associatività a sinistra, ma dare priorità agli operatori moltiplicativi,  $*$  e  $/$ , su quelli additivi,  $+$  e  $-$ , allora bisogna riscrivere la grammatica, introducendo nuove categorie sintattiche:

Espressione	$\rightarrow$	Termine   Espressione Additivo Termine
Termine	$\rightarrow$	Fattore   Termine Moltiplicativo Fattore
Fattore	$\rightarrow$	Variabile   Numero   (Espressione)
Additivo	$\rightarrow$	$+$   $-$
Moltiplicativo	$\rightarrow$	$*$   $/$

Si noti che, ad esempio, all'espressione  $12 / 2 * 3$  sarà attribuito lo stesso significato di  $(12 / 2) * 3$ , cioè 18, diverso da quello attribuito all'espressione  $12 / (2 * 3)$ , che è 2. Ciò è messo in evidenza dalla ricorsione a sinistra di Termine; analogamente per Espressione.

Si possono sempre eliminare le ricorsioni sinistre dirette, senza alterare il linguaggio generato dalla grammatica; nel nostro caso, basta sostituire le prime due linee con le seguenti quattro:

Espressione	$\rightarrow$	Termine SeguitoE
SeguitoE	$\rightarrow$	$\epsilon$   Additivo Termine SeguitoE
Termine	$\rightarrow$	Fattore SeguitoT
SeguitoT	$\rightarrow$	$\epsilon$   Moltiplicativo Fattore SeguitoT

introducendo dunque altri due non terminali, entrambi riducibili al vuoto: SeguitoE e SeguitoT, per descrivere come può proseguire, rispettivamente, un'espressione (dopo un termine) o un termine (dopo un fattore).

Facendoci guidare da questa grammatica (non ambigua), vediamo come si fa a costruire un analizzatore sintattico (in inglese *parser*) che legga un'espressione un simbolo alla volta e, qualora essa sia sintatticamente corretta, la trasformi nella equivalente *postfissa*.

Ad esempio, l'analisi dell'espressione

$$((a + 5) * b - (3 + c)) / d$$

produrrà la sequenza di simboli

$$a\ 5 + b * 3\ c + - d /$$

Processa\_espressione:

    Processa\_termine

    Processa\_seguito\_espressione

Processa\_seguito\_espressione:

    se il simbolo corrente è un operatore additivo **allora**

*azione semantica*: salva in OP il simbolo corrente

    Processa\_termine

*azione semantica*: produce OP

    Processa\_seguito\_espressione

Processa\_termine:

    Processa\_fattore

    Processa\_seguito\_termine

Processa\_seguito\_termine:

    se il simbolo corrente è un operatore moltiplicativo **allora**

*azione semantica*: salva in OP il simbolo corrente

    Processa\_fattore

*azione semantica*: produce OP

    Processa\_seguito\_termine

Processa\_fattore:

    legge il prossimo simbolo

    se il simbolo corrente è un nome di variabile o un numero **allora**

*azione semantica*: produce il simbolo corrente

    altrimenti se il simbolo corrente è la parentesi tonda aperta **allora**

        Processa\_espressione

        se il simbolo corrente non è la parentesi tonda chiusa **allora**

*segna errore*: attesa parentesi tonda chiusa

    altrimenti

*segna errore*: atteso nome di variabile o numero o

            parentesi tonda aperta

    legge il prossimo simbolo

Basterà dunque invocare la procedura Processa\_espressione; questa chiamerà Processa\_termine, che a sua volta invocherà Processa\_fattore, la quale leggerà il primo simbolo dell'espressione, che diventerà quello corrente...

Si noti che una decisione può essere presa in modo deterministico ogni volta che l'esecutore esamina il simbolo corrente, cioè l'ultimo che ha letto.

Se tutto va bene, alla fine del processo è stato già letto il simbolo *successivo* all'espressione; allora conviene imporre un simbolo *terminatore*: ad esempio, facciamo seguire all'espressione un punto e virgola, e magari – dopo la chiamata esterna di Processa\_espressione – controlliamo che il simbolo corrente sia proprio il punto e virgola, altrimenti sarà segnalato il relativo errore.

Abbiamo convenuto che una segnalazione di errore interrompa il processo di analisi; si tenga tuttavia presente che, nella pratica dei compilatori, sorge l'esigenza di recuperare la situazione di errore, per scoprire eventuali successivi errori, dipendenti o meno da quello rilevato.

Poiché tutti gli operatori sono binari, se si dispone di una pila che possa ospitare valori numerici, per la valutazione della forma postfissa si potrà procedere nel seguente modo, leggendo i simboli nello stesso ordine in cui sono stati prodotti dal processo di analisi sopra illustrato:

- quando si trova un operando, si mette il suo valore in cima alla pila;
- quando invece si trova un operatore, si prelevano due operandi dalla pila, si applica la corrispondente operazione alla coppia di operandi costituita dal *secondo* e dal *primo* prelevati dalla pila, nell'ordine, e infine si mette il risultato dell'operazione in cima alla pila.

Alla fine, sulla cima della pila si troverà il valore dell'espressione.

Ovviamente, per la valutazione di un'espressione con variabili, occorrerà riferirsi a un *ambiente* in cui ciascun nome di variabile è associato al suo tipo e all'indirizzo della corrispondente locazione di memoria, in cui è memorizzato il valore corrente di tale variabile. A seconda dei tipi degli operandi sarà applicato l'appropriato algoritmo per eseguire l'operazione; ad esempio, se gli operandi sono entrambi interi allora l'operatore / denoterà la divisione intera.

Si noti che nessuna delle grammatiche, date sopra per le espressioni, contempla l'operatore “– unario”; ciò non comporta alcun problema: ad esempio, per cambiare segno al contenuto di una variabile numerica  $v$  scriveremo  $v = 0 - v$ , esattamente come già aveva previsto Babbage!

(Provate ad arricchire il linguaggio delle espressioni con l'operatore binario  $\uparrow$  per l'elevamento a potenza, con priorità più alta dei moltiplicativi e associatività a destra.)

Come già accennato, il compilatore di un linguaggio di programmazione non si limita però all'analisi sintattica. Se consideriamo gli aspetti semanticci, pur contemplando le espressioni aritmetiche secondo una definizione semplificata come quella proposta nel presente paragrafo, basta che il linguaggio sia tipato affinché la libertà dal contesto sia perduta; ad esempio, per sapere se è corretta l'assegnazione  $b = a + 7$ , bisogna ricordare se le variabili  $a$  e  $b$  sono state dichiarate di un tipo numerico e, possibilmente, se alla variabile  $a$  è già stato assegnato un valore!

Se sottintendessimo il tipo intero per tutte le variabili e limitassimo le assegnazioni alla forma  $v = v + 1$  (come nel linguaggio minimale introdotto all'inizio del capitolo), oltre al problema di ricordare se alla variabile coinvolta in una

assegnazione sia stato assegnato in precedenza un valore, se ne porrebbe un altro già a livello sintattico: ricordare il nome della variabile che precede il simbolo = per verificare se è lo stesso che occorre subito dopo il simbolo =. Nemmeno questo requisito, in fondo banale, può essere espresso con una grammatica libera: occorrono regole *contextuali*, come vedremo nel prossimo paragrafo.

Ciononostante, ribadiamo l'importanza delle grammatiche libere (e non ambigue) per definire precisamente come deve essere scritto un programma e per guidarne l'analisi sintattica automatica: tant'è vero che sono state le più studiate e applicate.

Il genere di problemi ai quali abbiamo ora accennato (da non dimenticare è anche quello della corrispondenza in numero e tipo degli argomenti di una chiamata di funzione con i parametri formali stabiliti nella sua dichiarazione) può essere risolto via via costruendo e consultando opportune tabelle o demandando il compito a un modulo indipendente del compilatore. Teniamo presente che, una volta portata a termine (con successo) l'analisi sintattica di un programma, le cose da fare per avere infine il codice eseguibile sono ancora moltissime – e non parliamo nemmeno di ciò che ci aspetta per verificare se il programma “funzionerà” davvero!

Inoltre, se consideriamo i linguaggi naturali, e ricordiamo i “pensierini” che si scrivevano alle scuole elementari, ci accorgiamo del fatto che la parte di grammatica concernente la struttura di una frase è pure formulabile in modo libero dal contesto.

Senza qui entrare nei dettagli – altrimenti non basterebbe un intero volume! – diciamo soltanto che, delle grammatiche libere, sono state individuate particolari sottoclassi per le quali gli algoritmi di analisi (dei linguaggi da esse generati) risultano più efficienti di quanto ci si possa aspettare per una generica grammatica libera.<sup>13</sup> Nei casi peggiori, infatti, la complessità non è proporzionale alla lunghezza (in simboli) della stringa da analizzare, bensì al suo quadrato: inaccettabile per le applicazioni pratiche. Ciò accade quando si deve tornare indietro frequentemente a ripercorrere parti di stringa, facendo di volta in volta scelte alternative rispetto a quelle che evidentemente hanno portato a un fallimento.

Per fare un esempio, si pensi al linguaggio generato dalla grammatica libera

$$S \rightarrow \epsilon \mid 0S0 \mid 1S1$$

che è costituito da tutte le stringhe palindrome di lunghezza pari sull'alfabeto delle cifre binarie. Esso fa parte dei cosiddetti linguaggi *non deterministic*, proprio perché un automa a pila, per riconoscerlo, deve impilare i simboli letti, “indovinare” quando è stata raggiunta la metà della stringa e controllare che i rimanenti simboli letti siano uguali a quelli via via rimossi dalla pila. In sostanza, l'automa a pila deve avere un comportamento non deterministico: per simularlo in modo deterministico si devono tentare tutte le possibili alternative.

Osserviamo che, sostituendo  $S \rightarrow \epsilon$  con  $S \rightarrow 2$ , cambia l'alfabeto e cambia il linguaggio, che ora è deterministico, costituito da stringhe (di lunghezza dispari)

---

<sup>13</sup> Ovviamente, sono state le più utili nel progetto dei compilatori; un piccolo esempio di caso particolarmente felice, e importante, è stato qui trattato: la grammatica delle espressioni aritmetiche.

accettabili in una sola scansione e dunque con complessità lineare: il simbolo 2, che si trova al centro di ognuna delle stringhe da accettare, determina appunto il cambiamento di stato dell'automa, che dovrà quindi iniziare a svuotare la pila, controllando progressivamente che i simboli tolti siano uguali a quelli letti nel seguito della stringa.

### Le classi delle grammatiche meno restrittive.

Se vogliamo definire per mezzo di una grammatica un linguaggio come  $\{ 0^n 1^n 2^n \mid n > 0 \}$ , non ci può bastare una grammatica libera, bensì dobbiamo ricorrere a una grammatica *monotona*, le cui produzioni hanno, nella parte a sinistra della freccia, una qualsiasi sequenza di simboli (terminali o non) nella quale compaia almeno un non terminale e, nella parte a destra, una sequenza di simboli *non più corta* di quella a sinistra.<sup>14</sup>

I linguaggi generati da questa classe di grammatiche sono riconoscibili dagli automi che dispongono di una memoria di dimensione limitata – di volta in volta proporzionale alla lunghezza della stringa da analizzare – ma non soggetta al vincolo della gestione a pila: i simboli vi possono essere scritti e rivisti, ed eventualmente rimossi, in qualsiasi ordine.

Nel caso specifico del linguaggio  $\{ 0^n 1^n 2^n \mid n > 0 \}$ , si può ragionare così: se si dispone di una memoria sufficiente a contenere i due terzi dei simboli in input, la stringa sarà accettata se e soltanto se si riescono a memorizzare prima tutti i simboli 0 letti da input, poi tutti i simboli 1, e successivamente a cancellare dalla memoria uno 0 e un 1 per ogni simbolo 2 letto, rimanendo infine con la memoria vuota.

Una grammatica (monotona) che descriva questo linguaggio consta di sette produzioni:

1. S → 0AB	5. 1B → 12
2. S → 0SAB	6. 2B → 22
3. 0A → 01	7. BA → AB
4. 1A → 11	

Poniamoci ora dalla parte di una macchina che riceva un insieme di regole, come potrebbe essere questo, e una stringa, ad esempio 001122. In generale, quale metodo la macchina potrebbe applicare per *decidere* (poiché è possibile farlo) se la stringa data appartiene o meno al linguaggio generato dalla grammatica monotona data?

Un'idea è quella di esplorare “a tappeto” tutte le possibili derivazioni, iniziando dal simbolo distintivo; per ogni forma di frase che troveremo, proveremo a riscriverla in tutti i modi consentiti dall'applicazione di una qualche regola a qualche sua parte.

---

<sup>14</sup> L'unica eccezione è questa: se al linguaggio deve appartenere la stringa vuota, è ammessa la produzione  $S \rightarrow \epsilon$ , purché il simbolo distintivo S non compaia nella parte destra di nessun'altra regola.

Vediamo come si può procedere per garantire la terminazione con la risposta giusta, partendo dall'esempio proposto.

Elencate in un qualche ordine le regole – noi assumiamo quello della numerazione da 1 a 7 data sopra – iniziamo dal simbolo distintivo S e tentiamo di applicarle appunto in quest'ordine, finché è possibile e promettente:

$$S \rightarrow_1 0AB \rightarrow_3 01B \rightarrow_5 012$$

Siamo giunti a una stringa di soli terminali – che dunque appartiene al linguaggio generato dalla grammatica – però non è quella cercata; tornando indietro sui nostri passi di riscrittura, ci accorgiamo che non sono possibili altre applicazioni, fino a S; dopodiché:

$$S \rightarrow_2 0SAB \rightarrow_1 00ABAB \rightarrow_3 001BAB \rightarrow_5 0012AB$$

A questo punto nessuna regola è più applicabile: la forma di frase 0012AB non può essere riscritta; per trovare un'altra strada, questa volta basta tornare indietro di un passo:

$$001BAB \rightarrow_7 001ABB \rightarrow_4 0011BB \rightarrow_5 00112B \rightarrow_6 001122$$

Trovata! Se ora volessimo *generare* tutte le stringhe del linguaggio lunghe, al massimo, 6 simboli, dovremmo continuare la ricerca; procedendo a ritroso, troviamo un'alternativa al momento di riscrivere la forma di frase 00ABAB:

$$00ABAB \rightarrow_7 00AABB \rightarrow_3 001ABB$$

Ricordando le forme di frase già espansse, sappiamo che qui ci dobbiamo fermare (vedi sopra). Per trovare altre strade inesplorate occorre risalire a 0SAB:

$$0SAB \rightarrow_2 00SABAB$$

e qui dobbiamo di nuovo fermarci, perché la lunghezza di questa forma di frase è 7 e, per le caratteristiche della grammatica, non può diminuire nei successivi passi di riscrittura.

(Si noti che ora si potrebbe applicare la regola 1, ma così pure la 2; anzi, a partire da S la regola 2 è applicabile *ad libitum*.)

In conclusione, abbiamo provato – per verifica diretta – che la sola stringa del linguaggio lunga 6 è 001122; non soltanto: l'unica più corta è 012.

Per realizzare il procedimento che abbiamo ora esemplificato, useremo una pila (o una coda) in cui memorizzare le forme di frase ancora da espandere. Nel caso peggiore, la stringa data non è accettata e per ottenere tale responso tutte le possibilità saranno comunque esplorate.

Una forma di frase non è più riscritta se si verifica almeno una di queste eventualità:

- la sua lunghezza supera quella della stringa data;
- nessuna produzione è ad essa applicabile;
- è già stata generata e sviluppata precedentemente.

Per riconoscere quest'ultima circostanza, dobbiamo ricordare (in un apposito “dizionario”<sup>15</sup>) le forme di frase precedentemente generate, onde evitare di rimetterle sulla pila (o nella coda): ciò non soltanto è utile, perché ci evita di sprecare tempo a ripetere indagini già fatte, ma è anche necessario, per non incorrere nell'esplorazione di sequenze di derivazioni che si ripetono all'infinito. Infatti, potrebbe darsi il caso in cui, dopo un certo numero di passi di riscrittura che mantengono costante la lunghezza, si ritorni a una stessa forma di frase.

Nella peggiore delle ipotesi tale lunghezza è proprio quella, chiamiamola  $L$ , della stringa da riconoscere; un limite superiore al numero di passi è dato allora dal numero di forme di frase di lunghezza  $L$  (andrebbero escluse le stringhe costituite da soli terminali, non ulteriormente riscrivibili), dopodiché si arriverà per forza a una forma di frase già trovata. Se  $n$  è il numero di simboli (terminali o non) allora le forme di frase di lunghezza  $L$  sono  $n^L$ .

Questo ragionamento ci permette di affermare che, nel caso peggiore, la complessità rispetto al tempo per l'accettazione o il rifiuto, adottando il procedimento delineato, è *esponenziale* nella lunghezza della stringa.

In conclusione, data una grammatica monotona e data una stringa di terminali, è *decidibile* se la stringa appartiene al linguaggio generato dalla grammatica: si può scrivere un programma che ci dica sì o no.

Sfortunatamente, non sono stati finora trovati algoritmi di complessità inferiore all'esponenziale, né è stato dimostrato che non ne esistono: cosicché non si sa se la complessità intrinseca del problema sia esponenziale o meno nella lunghezza della stringa da analizzare.

Comunque, circa la complessità di alcuni problemi “difficili”, torneremo in argomento nel prossimo capitolo.

È stato invece provato che la classe dei linguaggi descritti non cambia limitando la forma delle produzioni al caso particolare delle *regole contestuali*:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

dove  $A$  è un non terminale, mentre  $\alpha$ ,  $\beta$  e  $\gamma$  sono sequenze di simboli (terminali o non), con  $\gamma$  non vuota. Detto in parole:  $A$  può essere riscritto come  $\gamma$  se è preceduto da  $\alpha$  e seguito da  $\beta$ , cioè nel contesto  $(\alpha, \beta)$ .

Nella precedente grammatica, l'unica regola non contestuale è l'ultima,  $BA \rightarrow AB$ , che può essere sostituita dalle quattro regole contestuali

$$BA \rightarrow CA, \quad CA \rightarrow CD, \quad CD \rightarrow AD, \quad AD \rightarrow AB$$

(nella prima e nella terza il contesto sinistro è vuoto, nelle altre due lo è il destro), senza che il linguaggio cambi.

I linguaggi generati dalle grammatiche monotone sono perciò detti contestuali.

---

<sup>15</sup> Il dizionario è costituito da una struttura di dati destinata a crescere costantemente: infatti, da esso non saranno mai cancellate “parole”! Le soluzioni consigliabili sono un albero binario di ricerca o una struttura *trie* o una *hash table* (aperta).

Una classe più ampia delle grammatiche monotòne, che le comprende in senso stretto, si ottiene rimuovendo il vincolo sulla parte destra delle produzioni. La sola prescrizione che rimane riguarda la parte a sinistra della freccia, in cui deve comparire almeno un non terminale: queste sono le grammatiche *a struttura di frase*, o *illimitate*.

Se una stringa appartiene al linguaggio generato da una di esse, l'accettazione può essere fatta da un automa con memoria potenzialmente illimitata, ossia da una macchina di Turing (o equivalente).

In generale, il problema del riconoscimento è soltanto semidecidibile: se una stringa non appartiene al linguaggio, non è detto che il processo abbia termine. Infatti, se la grammatica non è monotona, non si può stabilire a priori un limite alla lunghezza delle espansioni generate nei passi di riscrittura, che poi potrebbero ridursi proprio in virtù dell'applicazione di regole non monotone.

Esistono linguaggi – anzi, sono la stragrande maggioranza – che non si possono generare nemmeno con grammatiche illimitate: per essi il problema del riconoscimento è del tutto indecidibile. Ce ne possiamo facilmente convincere considerando tutti i linguaggi sull'alfabeto binario, che sono tutti i possibili sottoinsiemi dell'insieme (infinito) delle stringhe formate da un numero finito di bit. Quindi i linguaggi costituiscono un'infinità di ordine superiore rispetto all'infinità delle macchine di Turing, e di tutti i programmi scrivibili in un linguaggio di programmazione, che hanno una lunghezza finita. Possiamo affermare che tra i linguaggi e gli algoritmi in grado di riconoscerli sussiste lo stesso rapporto che vi è tra l'insieme dei numeri reali e quello dei naturali.

Come abbiamo visto, partendo dalle grammatiche regolari e salendo su, fino a quelle illimitate, le regole diventano più complesse e gli automi più potenti, mentre i linguaggi sono soggetti a vincoli più rigidi.

Un automa a stati finiti privo di memoria ausiliaria non può neppure “contare”: infatti non riesce a riconoscere le stringhe formate da un numero arbitrario di 0 seguiti da altrettanti 1; con l'ausilio di una pila può contare due cose, ma non tre; con una memoria limitata dalla lunghezza della stringa, ma accessibile a piacere, può contare tre o più cose, ma non può “calcolare”; infine, con una memoria potenzialmente illimitata può calcolare qualsiasi funzione... purché sia calcolabile!

Concludiamo con una breve nota storica.

Uno dei primi sistemi “grammaticali”, introdotto dal matematico norvegese Axel Thue agli inizi del '900, consiste nei cosiddetti *semi-sistemi di Thue*: evitando di definire simboli non terminali, si usano un alfabeto e regole di riscrittura della forma  $\alpha \rightarrow \beta$ , dove  $\alpha$  e  $\beta$  sono sequenze di simboli dell'alfabeto, con  $\alpha$  non vuota. Per mezzo di questi sistemi si riescono a esprimere tutte le funzioni calcolabili: un'altra definizione di algoritmo! E anche qui, com'è logico aspettarsi, si ritrovano problemi indecidibili: ad esempio, determinare se da una sequenza arbitraria se ne possa ottenere un'altra, applicando le regole di un sistema pure arbitrario, è un problema soltanto semidecidibile.

La classificazione delle grammatiche che abbiamo delineato in questo capitolo è stata formalizzata da Noam Chomsky, del Massachusetts Institute of Technology (MIT), nella seconda metà degli anni '50. Volendo tentare un accordo tra lo studio dei linguaggi naturali e le ricerche sui sistemi formali, Chomsky partì dal problema di come specificare l'insieme delle regole, affinché un automa potesse generare frasi sintatticamente corrette, e quindi basò la sua “gerarchia delle grammatiche” sia sulla loro adeguatezza a descrivere aspetti significativi del fenomeno linguistico, sia sulla possibilità di una loro trattazione di natura matematica.

Il formalismo che abbiamo usato per scrivere le grammatiche è noto come Backus Normal Form (infatti, nel 1959, per la descrizione dell'ALGOL 60, fu introdotto da John W. Backus della IBM, già coordinatore del gruppo che sviluppò il FORTRAN) o Backus-Naur Form (per i contributi dati da Peter Naur dell'Università di Copenhagen). Si noti che la forma delle regole è simile a quella successivamente usata in alcuni linguaggi di programmazione *dichiarativi*, come il PROLOG.

### **Alcuni quesiti.**

I linguaggi formali e i sistemi di riscrittura di termini hanno ispirato diversi quesiti proposti dal *Kangourou dell'Informatica*: “Tretterobòt”, “Analisi grammaticale” e “Reazioni chimiche” (gara di marzo 2009), “Elementi pericolosi” (finale di maggio 2010), “Parola d'ordine” (gara di marzo 2011), “Parole crociate regolari” (libretto del 2014), “Scelte... climatiche” e “Strane parole” (libretto del 2015), che, alla luce di quanto sopra esposto, potrebbe essere interessante classificare.

Di pile e code trattano invece due problemi del 2009: “In pizzeria” (gara di marzo) e “I gettoni e i piattini” (finale di maggio); altri si trovano nei libretti del 2014 (“Pila” e “Il take away”) e del 2015 (“I conigli dispettosi”).

Infine, alle domande disseminate lungo l'*excursus* che ha occupato la seconda parte del capitolo, aggiungo alcuni quesiti, complementari agli esempi illustrati e assai istruttivi, che volutamente lascio senza risposta.

i) Disegnate un automa riconoscitore (col minimo numero di stati) e scrivete una grammatica (regolare) per il linguaggio sull'alfabeto { 0, 1, 2 } costituito dalle stringhe che non contengono due simboli consecutivi uguali.

ii) Le due grammatiche

$$\begin{aligned} S &\rightarrow AB \mid BA \\ A &\rightarrow 0 \mid AAB \mid ABA \mid BAA \\ B &\rightarrow 1 \mid ABB \mid BAB \mid BBA \end{aligned}$$

$$\begin{aligned} S &\rightarrow AB \mid ASB \\ A &\rightarrow 0 \\ B &\rightarrow 1 \\ AB &\rightarrow BA \\ BA &\rightarrow AB \end{aligned}$$

generano lo stesso linguaggio. A quali classi appartengono? Quale linguaggio generano? Come può essere riconosciuto questo linguaggio, in modo deterministico, da un automa che disponga di *due* pile?

iii) Consideriamo le seguenti grammatiche libere:

$$S \rightarrow 0S | 0S1S | 2$$

$$\begin{aligned} S &\rightarrow 0SA | 2 \\ A &\rightarrow \epsilon | 1S \end{aligned}$$

È immediato constatare che generano lo stesso linguaggio e che sono ambigue: ad esempio, la stringa 00212 ha due strutture sintattiche diverse.

La sintassi del linguaggio di programmazione minimale, definita in questo capitolo, impone che il corpo di un ciclo **while** sia delimitato da una coppia di parentesi graffe. Tuttavia, nei più comuni linguaggi, quando il corpo è costituito da una sola istruzione le parentesi sono opzionali. Lo stesso vale anche per le parti **then** ed **else** di un'istruzione condizionale, che per giunta può essere priva della parte **else**; sicché un'istruzione (composita) della forma

$$\mathbf{if} (c_1) \mathbf{then} \mathbf{if} (c_2) \mathbf{then} i_1 \mathbf{else} i_2$$

è ambigua perché la parte **else** può essere associata al primo **then** oppure al secondo.

Lo stesso identico problema è riconoscibile nelle due grammatiche date sopra.

Se si vuole privilegiare il secondo **then**, adottando la norma generale di associare ogni **else** al **then** precedente più vicino, al quale non sia stato ancora associato un **else**, allora si hanno due possibilità: o si dà una grammatica non ambigua che strutturi le istruzioni condizionali nel modo desiderato, o si assume una grammatica analoga alla seconda delle due date sopra, privilegiando – ogni volta che si presenta l'alternativa – la riscrittura di A come 1S rispetto alla stringa vuota (si rifletta sul fatto che, seguendo questa seconda idea, si richiede l'introduzione di una regola semantica in sede di analisi sintattica).

Può essere un utile esercizio provare a scrivere una grammatica equivalente alle due date, ma non ambigua, che strutturi le stringhe come suggerito.

Per inciso, il problema dell'equivalenza di due grammatiche – stabilire cioè se generano lo stesso linguaggio oppure no – è totalmente indecidibile.

iv) Consideriamo l'alfabeto  $\{a, b, =\}$  e la grammatica

$$\begin{aligned} S &\rightarrow = | aAS | bBS \\ Aa &\rightarrow aA \\ Ab &\rightarrow bA \\ A= &\rightarrow =a \\ Ba &\rightarrow aB \\ Bb &\rightarrow bB \\ B= &\rightarrow =b \end{aligned}$$

A quale classe di grammatiche appartiene? Qual è il linguaggio da essa generato?

v) Consideriamo l'alfabeto  $\{ a \}$  e la grammatica

$$\begin{array}{lcl} S & \rightarrow & ABaC \\ aD & \rightarrow & Da \\ aE & \rightarrow & Ea \\ AD & \rightarrow & AB \\ AE & \rightarrow & \epsilon \\ Ba & \rightarrow & aaB \\ BC & \rightarrow & DC \mid E \end{array}$$

A quale classe di grammatiche appartiene? Qual è il linguaggio da essa generato?

**Parole chiave:** funzione effettivamente calcolabile, tesi di Church, linguaggio completo, operazioni di base, astrazione, uso e composizione di operazioni, problema indecidibile, teoria dell'informazione algoritmica, calcolo della radice quadrata (con i metodi di Erone, di Bombelli e di bisezione), regole sintattiche o di riscrittura, grammatiche (regolari, libere, contestuali, monotone, illimitate), automi riconoscitori di linguaggi, ambiguità, *parsing* di espressioni aritmetiche, espressioni aritmetiche in forma postfissa, regole contestuali, semi-sistemi di Thue.

## 4. Problemi intrattabili o quasi

Nel capitolo precedente abbiamo accennato alle operazioni fondamentali del “calcolo” e a come possono essere espresse in un linguaggio interpretabile da un automa. Abbiamo poi visto alcuni esempi sia di problemi risolvibili (e di algoritmi per risolverli) sia di questioni indecidibili mediante una procedura effettiva.

Ora affronteremo un breve viaggio nel mondo dei problemi complessi: problemi intrinsecamente esponenziali o super-esponenziali, commessi viaggiatori, imballaggi e zaini, numeri primi, fattorizzazione e crittoanalisi. Accenneremo ai diversi livelli di “difficoltà” di tali problemi, e descriveremo degli algoritmi esatti o approssimati per risolverne qualcuno.

Certe loro istanze trovarono spazio nelle gare *Kangourou dell'Informatica*.

### Problemi esponenziali e super-esponenziali.

Se le questioni indecidibili non hanno speranza di essere risolte in modo algoritmico – poiché appunto *nemmeno in teoria* esiste un automa che le possa, in generale, risolvere – bisogna riconoscere che spesso, nella realtà, si incontrano questioni *decidibili* (per le quali esiste un algoritmo che ne individua la risposta giusta, in tempo finito, per ogni possibile valore dei dati in ingresso) che devono essere considerate *di fatto intrattabili*, perché il “tempo” richiesto per risolverle cresce in modo *esponenziale* (o peggio ancora) con la “dimensione del problema” (locuzione con cui si intende il numero di bit necessario a rappresentare i dati in ingresso). Anziché di tempo, per essere più precisi, si dovrebbe parlare di passi compiuti da una macchina di Turing, o di operazioni di base eseguite da una macchina di riferimento equivalente.

Per un problema in sé, un conto è essere risolvibile in modo algoritmico, un conto è esserlo in un tempo accettabile. Tuttavia, alla luce delle odierne conoscenze, non è tracciabile un confine netto tra i problemi che possono essere risolti in maniera *efficiente* (per i quali esiste *almeno un* algoritmo risolutivo esatto che, pure nei casi sfavorevoli, richieda tempo, *al più, polinomiale*, cioè limitato superiormente da un polinomio nella lunghezza dei dati in ingresso codificati in bit) e quelli praticamente *intrattabili* al crescere della dimensione dei dati: per questi ultimi, che chiamiamo “intrinsecamente (almeno) esponenziali”, *qualsiasi* algoritmo risolutivo richiede un tempo di esecuzione che, in definitiva, dipende *almeno esponenzialmente* dalla lunghezza dei dati in ingresso. Anzi, come vedremo nei prossimi paragrafi, proprio nell’esplorazione della regione di confine tra queste due classi di problemi si sono avventurati tanti ricercatori in *teoria della complessità computazionale* nel corso dell’ultimo mezzo secolo.<sup>16</sup>

---

<sup>16</sup> Anche nella classe dei problemi indecidibili sono state definite gerarchie di sottoclassi; alcune questioni menzionate nel precedente capitolo vi si trovano ai livelli più alti: ad esempio, i problemi della totalità, della non-totalità e dell’equivalenza funzionale per i programmi, che rimarrebbero indecidibili anche qualora, per pura ipotesi, fosse decidibile il problema dell’arresto.

Al momento, l'indagine è giunta a un livello tale da distinguere oltre 500 classi di complessità, in base alle sottigliezze matematiche dei metodi che vi potrebbero essere impiegati (si veda: [https://complexityzoo.uwaterloo.ca/Complexity\\_Zoo](https://complexityzoo.uwaterloo.ca/Complexity_Zoo)). Però a certe questioni, la cui importanza sarebbe fondamentale, non è ancora riuscita a dare una risposta: saranno risolvibili? E, se sì, saranno mai risolte? (Non da una macchina, s'intende, ma dalla mente umana!)

Diamo comunque uno sguardo, dapprima, ai problemi esponenziali.

Un esempio banale di problema intrinsecamente esponenziale consiste nella *stampa* di tutti gli anagrammi di una data parola; da ROMA si ottengono: AMOR, AMRO, AOMR, ..., ROMA. In questo caso gli anagrammi sono 24; con una parola di 10 lettere (tutte diverse) sarebbero più di tre milioni e mezzo! Se le lettere della parola sono  $n$  e sono distinte, allora i possibili anagrammi sono  $n! = 1 \cdot 2 \cdot 3 \cdots n$  (il fattoriale di  $n$ ), che si approssima – sempre meglio al crescere di  $n$  – con  $\sqrt{2\pi n} \cdot (n/e)^n$  ( $e$  è la base del logaritmo naturale); questa approssimazione è nota come *formula di Stirling*, risale al 1730, ed è ancor più precisa se moltiplicata per  $e$  elevato all'esponente  $1/(12n)$ . Altri esempi ovvi sono l'elencazione di tutti i sottoinsiemi di un dato insieme finito (ad esempio di nuovo le  $n$  diverse lettere di una parola: i sottoinsiemi sono  $2^n$  e corrispondono a tutti i possibili modi di scegliere un gruppo di lettere, anche tutte o nessuna) o delle mosse per spostare una *torre di Hanoi* (che sono  $2^n - 1$ , se la torre è formata da  $n$  dischi).

Delle torri di Hanoi torneremo a parlare nel seguito. Adesso, affinché non sorgano confusioni, è meglio chiarire la nozione di rapporto tra il tempo impiegato per produrre il risultato in output e la dimensione in bit dell'input.

Riflettiamo su un semplicissimo problema: dato un intero positivo  $n$ , scrivere tutti i numeri interi da 1 a  $n$ . Poiché si deve supporre  $n$  codificato in binario, allora la dimensione dell'input, in bit, è  $d = \text{floor}(\log_2 n) + 1$ ; se  $n$  fosse codificato in decimale, cambierebbe soltanto la base del logaritmo. Partendo da 0, la macchina dovrà applicare  $n$  volte la funzione successore, stampando ogni volta il risultato ottenuto. Se la macchina stampa i numeri usando esattamente  $d$  bit per ciascuno di essi (quindi aggiungendo eventuali 0 non significativi in testa), allora non v'è dubbio che il tempo necessario non può essere inferiore a quello impiegato per stampare un bit moltiplicato per  $n \cdot d$ , che è circa  $2^d \cdot d$ : dunque *esponenziale* nella dimensione dell'input! (Alla stessa conclusione si perviene anche nel caso in cui non siano stampati 0 non significativi, ma sia usato un simbolo apposito per separare un numero dal successivo.) Questo genere di complessità rientra comunque in quella degli algoritmi *pseudo-polinomiali*, di cui parleremo successivamente.

Se invece fosse usata una codifica *unaria*, quale sarebbe il numero di simboli da stampare in funzione del numero dato in input? (Ad esempio: dato 11111, l'output prodotto dovrebbe essere 1#11#111#1111#11111, ... e così “pseudo” è giustificato). Non bisogna lasciarsi fuorviare da un ragionamento (assolutamente errato) del tipo: dato  $n$ , si devono calcolare e stampare  $n$  numeri (consecutivi, che si sanno calcolare facilmente), e quindi la complessità è *lineare*...

Ciò infatti significherebbe che il tempo impiegato per produrre il risultato in output è in definitiva *proporzionale* alla dimensione dell'input!

Ma allora ritorniamo un attimo al problema della torre di Hanoi e cerchiamo di essere più precisi. Le mosse da fare sono circa  $2^n$  e ciascuna può essere codificata con un numero fissato di bit (ne bastano tre, per denotare la coppia ordinata di pioli coinvolti): quindi possiamo dire che il numero di bit da stampare sia proporzionale a  $2^n$ , dove  $n$  è (circa)  $2^d$ , e perciò la complessità è *doppiamente esponenziale* nella lunghezza dell'input!

Quelli testé illustrati sono esempi ovvi, perché richiedono la produzione in uscita di un risultato la cui lunghezza è esponenziale nella dimensione del problema, e di conseguenza il tempo richiesto non può essere di ordine inferiore.

Tuttavia ci sono problemi (detti *decisionali*) per i quali la risposta è molto breve, “sì” o “no” (un solo bit!), ma il tempo necessario per calcolarla non può essere meno che esponenziale al crescere della loro dimensione, almeno per certe istanze (i cosiddetti “casi peggiori”).

Abbiamo già incontrato alcuni importanti problemi decisionali, anche non risolvibili mediante un algoritmo. Riprendiamo le espressioni regolari introdotte nel capitolo precedente e aggiungiamo l'operatore “quadrato”: se  $R$  è un'espressione regolare, lo è anche  $R^2 = RR$ , che rappresenta l'insieme delle stringhe ottenute concatenando due stringhe qualsiasi (diverse o uguali che siano) descritte da  $R$ . Ciò non aumenta la potenza espressiva del formalismo: abbiamo soltanto aggiunto la possibilità di scrivere in maniera abbreviata certe espressioni. In queste ipotesi, stabilire se due espressioni regolari descrivono lo stesso insieme di stringhe, oppure no, richiede uno spazio di memoria (e quindi un tempo) esponenziale. Questo fu il primo problema non banale del quale è stato dimostrato il carattere esponenziale (A. R. Meyer e L. J. Stockmeyer, 1972).

(A tal proposito è d'uopo un'osservazione: è assai rischioso tentare di risolvere i problemi che hanno un limite inferiore di spazio esponenziale, come in realtà lo hanno tutti quelli che richiedono un tempo doppiamente esponenziale. Basti pensare che anche qualora si arrivasse a memorizzare un bit in una decina di atomi – e il grosso problema sarebbe poi come fare a leggerli! – per contenere tutto ciò che esiste di stampato sulla Terra sarebbe sufficiente una piccola biglia, ma per formare una memoria di  $2^{164}$  bit occorrerebbero gli atomi dell'intero pianeta.)

Da allora sono stati scoperti parecchi altri problemi intrinsecamente esponenziali, riguardanti soprattutto la teoria dei linguaggi formali, alcune logiche e l'analisi di certi giochi (il blocco stradale, la dama generalizzata): per questi non v'è speranza di trovare algoritmi risolutivi efficienti.

La dama generalizzata è giocata su una scacchiera  $n \times n$ , con le regole della dama classica e un numero di pedine adeguato.

Il blocco stradale è giocato, sempre tra due avversari, su un grafo arbitrariamente grande, dove i nodi e gli archi rappresentano, rispettivamente, gli incroci e i tratti di strada; il grafo può essere non planare, dunque può succedere che un tratto di strada debba passare sopra un altro, tramite un cavalcavia; ciascun arco è colorato con uno

fra tre colori. Alcuni nodi sono contrassegnati da “vince A”, altrettanti altri da “vince B” (A è il giocatore che muove per primo). Ciascuno dei due giocatori possiede un certo numero di automobili, inizialmente distribuite su altrettanti nodi; su uno stesso nodo non possono mai trovarsi due automobili, indipendentemente da chi le possegga. Il giocatore di turno muove una delle sue automobili, fino a portarla su un altro nodo (libero) percorrendo archi tutti di uno stesso colore e attraversando nodi liberi; se non può muovere, perde. Se invece arriva in un nodo contrassegnato dalla propria vittoria, vince.

Ebbene, è stato dimostrato che non esiste in assoluto alcun algoritmo efficiente per stabilire se il primo giocatore a muovere abbia una strategia vincente oppure no. Il tempo richiesto aumenta in modo esponenziale rispetto alla dimensione  $n$  dell’input: ciò significa che, al crescere di  $n$ , qualsiasi algoritmo – anche il migliore – in grado di decidere in generale la questione potrà sempre imbattersi in configurazioni iniziali – nel caso del blocco stradale ce ne possono essere davvero molte! – che, per essere analizzate, richiederanno un tempo dell’ordine di una costante ( $> 1$ ) elevata alla potenza  $n$ . Poiché un algoritmo esponenziale risolutivo esiste, si può affermare che il problema in sé ha complessità esponenziale. E non solo: è altrettanto costosa anche la *verifica* di una strategia vincente, ammesso che ne sia in qualche modo prodotta una! (Provate voi stessi a disegnare diversi tabelloni e a fare poi delle partite!)

Un altro esempio di questione intrinsecamente esponenziale è la *soddisfattibilità* di una arbitraria proposizione della *logica proposizionale dinamica* (o *algoritmica*) che coinvolga l’operatore **after** su programmi “non raffinati”. Vediamo che cosa significa.

Nel precedente capitolo abbiamo introdotto un linguaggio di programmazione “minimale”, che consta di sequenze di istruzioni (particolari forme di assegnazioni e cicli) composte a piacere. Nei brevi programmi riportati abbiamo aggiunto dei commenti che esprimono certe condizioni, sicuramente soddisfatte subito dopo l’esecuzione dell’istruzione che le precede. Tali condizioni sono dette *postcondizioni*, e forniscono un aiuto a chi esamina il programma, sia per capire che cosa avviene, sia per (tentare di) provarne la correttezza. Un’analoga funzione hanno le *precondizioni*, inserite prima di un’istruzione; di solito sono scritte all’inizio di una procedura, per affermare ciò che dev’essere vero affinché l’esecuzione della procedura abbia successo.

Agli operatori della logica proposizionale classica  $\wedge$ ,  $\vee$ ,  $\sim$ ,  $\rightarrow$  (rispettivamente: “e”, “o”, “non”, “implica”) aggiungiamo il costrutto

**after** ( $s, c$ ) col significato: al termine dell’esecuzione della sequenza  
di istruzioni  $s$ , la proposizione  $c$  è vera.

Ad esempio, abbreviando l’istruzione **while** (C1) { S1 } con W1 e l’istruzione **while** (C2) { S2 } con W2, e indicando con W1 W2 la sequenza da esse formata, possiamo scrivere la seguente proposizione:

$$((C3 \rightarrow \text{after}(W1, C4)) \wedge (C4 \rightarrow \text{after}(W2, C5))) \rightarrow \\ (C3 \rightarrow \text{after}(W1 W2, C5 \wedge \neg C2))$$

Ricordando il significato dell'implicazione materiale ( $A \rightarrow B$  equivale a  $B \vee \neg A$ ) e che la condizione (o *guardia*) di un ciclo **while** è falsa se e quando l'esecuzione del ciclo termina, il lettore non dovrebbe trovare difficoltà a capire che qui si tratta di un'asserzione *comunque vera*, indipendentemente dalle condizioni rappresentate dalle meta-variabili  $C_i$  e dalle sequenze di istruzioni rappresentate dalle meta-variabili  $S_1$  e  $S_2$  (ammesso che l'esecuzione dei cicli termini).

Le asserzioni di questo tipo appartengono alla logica proposizionale dinamica: "dinamica" proprio perché parla di qualcosa che si verificherà una volta eseguito un programma.

Ebbene, è stato provato che il problema di decidere se una data proposizione di questa logica *possa essere vera* è intrinsecamente esponenziale (M. J. Fischer e R. E. Ladner, 1979).

Se pensiamo invece a formalismi logici "statici" ma un po' più ricchi – ad esempio con i quantificatori "esiste" e "per ogni", che legano variabili sui numeri naturali, e la possibilità di parlare di numeri naturali e di uguaglianza e addizione tra essi, come nell'aritmetica PA a pagina 54, ma senza gli assiomi 5 e 6 – allora il problema di determinare il valore di verità di una formula (ben formata e chiusa) diventa almeno *doppiamente esponenziale*, dell'ordine di  $2 \uparrow 2 \uparrow (c \cdot n)$ , con  $c$  costante positiva e  $n$  lunghezza dell'enunciato (M. J. Fischer e M. O. Rabin, 1974). A questa logica, più debole di PA, non finitamente assiomatizzabile ma completa, detta "aritmetica di Presburger" (dal nome del matematico ebreo polacco Mojżesz Presburger, allievo di Tarski, che la introdusse nel 1929), possiamo aggiungere qualcos'altro, ad esempio:

- ◆ o la possibilità di parlare di *insiemi* di numeri naturali: in tal caso, la determinazione del valore di verità rimane possibile, ma diventa veramente proibitiva, perché nella migliore delle ipotesi il numero dei livelli di esponenti 2 cresce con  $n$  (ricordiamo la funzione di Ackermann), e dunque rientra nei cosiddetti problemi "altamente intrattabili" (A. R. Meyer, 1975);
- ◆ o la *moltiplicazione* tra numeri naturali: e qui ricadiamo nell'indecidibilità! Infatti, nell'*aritmetica del prim'ordine* la verità è totalmente indecidibile, in generale (ricordiamo i teoremi di Gödel e Tarski);
- ◆ o entrambe le cose sudette: si ha allora l'*aritmetica del second'ordine*, nella quale l'indecidibilità si eleva a un grado ancora più alto...

Abbandoniamo i voli pindarici, e scendiamo a problemi più comuni nella pratica e risolvibili in modo algoritmico... ma per i quali non è stato finora trovato alcun procedimento *efficiente in generale* (e si dubita che vi sia), nonostante non sia stato ancora dimostrato che non esiste.

In altre parole, *si sospetta fortemente* che i problemi di cui ora ci occuperemo siano intrattabili...

## Le classi P e NP.

Dalla metà degli anni '60, la nozione di *efficienza* è associata all'esistenza di un procedimento eseguibile in *tempo polinomiale* (e, possibilmente, limitato da un polinomio di grado non troppo elevato, altrimenti, all'atto pratico, l'efficienza ne sarebbe compromessa). La classe di tutti i problemi decisionali per i quali esiste un algoritmo risolutivo tempo-polinomiale è stata chiamata P, che sta appunto per “polinomiale”: dunque, un problema in P è *risolvibile* in tempo polinomiale.

Sono esempi di problemi in P: decidere se due numeri interi maggiori di 1 sono primi tra loro (Euclide, circa 300 a. C.), decidere se un sistema di equazioni lineari ammette una soluzione (con una versione del “metodo di eliminazione di Gauss” proposta da Jack Edmonds nel 1967), decidere se un numero è primo (M. Agrawal e altri, 2002). I dati del problema devono essere sempre intesi arbitrari e codificati in binario, in maniera conveniente.

La classe dei problemi decisionali che sono *verificabili* in tempo polinomiale è detta NP, che sta per “non-deterministico polinomiale”. Cerchiamo di intuire il significato di questa terminologia, aiutati da una spiegazione informale e qualche esempio.

In sostanza si richiede che, a prescindere dal metodo impiegato (purché esatto), tutte le volte che il problema in questione ha risposta “sì” il solutore possa produrre anche una *prova* della veridicità della risposta, che sia appunto controllabile mediante un algoritmo (“certificatore”) tempo-polinomiale.

Consideriamo, ad esempio, il problema di decidere se un numero *non* è primo. Per ogni sua istanza che ammetta risposta “sì” esiste un “testimone”, in questo caso un numero intero maggiore di 1 e minore del numero dato, che divide esattamente il numero dato; e la verifica può essere fatta eseguendo una divisione, quindi in tempo polinomiale. Questo problema è “funzionalmente equivalente” (nel senso che a stesso dato numero corrisponde stessa risposta) al seguente: decidere se un numero ha un fattore maggiore di 1 e minore o uguale alla sua radice quadrata (troncata); se la risposta è “sì” ed è prodotto un testimone, il controllo può essere fatto altrettanto semplicemente, con una divisione seguita da un confronto (il quoziente della divisione deve essere maggiore o uguale al testimone). Si noti che, infatti, non è richiesto che il fattore sia primo.

Un altro esempio: decidere se un grafo ha un *circuito hamiltoniano*, cioè un cammino chiuso che includa tutti i nodi una e una sola volta ciascuno. In caso di risposta affermativa, un testimone (o “certificato”, come pure si dice) potrà essere costituito da una lista contenente tutti i nodi una sola volta, ad eccezione dell'ultimo che replicherà il primo, in modo tale che vi sia un arco da ciascun nodo al successivo presente nella lista.

In generale, un grafo (orientato, ossia con archi dotati di freccia che indica il verso di percorrenza) con  $n$  nodi (messi in una corrispondenza biunivoca con i numeri da 1 a  $n$ ) può essere rappresentato per mezzo di una matrice  $n \times n$  di bit, detta *matrice di adiacenza*, tale che il bit di riga  $i$  e colonna  $j$  sia 1 se e soltanto se esiste l'arco dal nodo  $i$  al nodo  $j$ . (In questo genere di problemi rimangono esclusi gli archi da un

nodo a sé stesso – i cosiddetti *cappi* – sicché si può supporre che sulla diagonale principale della matrice vi siano tutti 0; inoltre, qualora il grafo sia non orientato, cioè con archi senza freccia percorribili nei due sensi, la matrice sarà simmetrica, e in tal caso si preferisce parlare di *ciclo hamiltoniano*.) Si intuisce quindi, per il problema del circuito hamiltoniano, come procedere per un controllo in tempo polinomiale del testimone (la lista dei nodi nell’ordine in cui sono stati raggiunti).

In alternativa, e in maniera equivalente, si può ragionare così: *immaginiamo* che il problema possa essere risolto da un algoritmo *non deterministico*, in grado di fare sempre una scelta giusta ogni volta che si trovi davanti a due o più alternative, in modo tale da: 1) non escludere mai, se c’è, la possibilità di arrivare a dare risposta “sì”, ma anzi di giungervi per la via più breve; 2) prendere altrimenti la via più lunga che arrivi a un fallimento (laddove invece un algoritmo deterministico “ovvio” enumera le alternative ed eventualmente ritorna sui propri passi, arrivando a valutare negativamente *tutte* le possibili scelte, nei casi peggiori). O, se si preferisce, si può anche pensare che – non meno magicamente – un algoritmo non deterministico sia dotato di un grado di parallelismo illimitato: può compiere *simultaneamente* tutte le scelte, pure quelle sbagliate, e soltanto se nessuna giunge a soddisfare la richiesta risponde “no”. Ebbene, se un tale ipotetico algoritmo riesce a risolvere il problema in tempo polinomiale (impiegato alla fin fine a valutare *in parallelo* gli esiti di tutte le alternative, tempo che dunque corrisponde all’alternativa più lunga), allora il problema appartiene alla classe NP.

Poiché un algoritmo deterministico può essere visto come caso particolare di un algoritmo non-deterministico – o anche, più semplicemente, poiché un problema risolvibile in tempo polinomiale è evidentemente verificabile in tempo polinomiale – la classe P è inclusa nella classe NP. Sul viceversa ci sono forti dubbi, sebbene nessuno sia ancora riuscito a dimostrare che P sia inclusa *strettamente* in NP, ossia che per almeno uno dei problemi in NP *non esistano* algoritmi risolutivi tempo-polynomiali. In effetti, si tratta di una delle più importanti questioni aperte nella teoria della complessità computazionale.

Dagli anni ’70 del secolo scorso, sono stati individuati parecchi tra i problemi “più difficili” della classe NP, detti *NP-completi*: ciò significa che *qualunque* problema in NP può essere “trasformato”, in tempo polinomiale, in uno di essi. In particolare, i dati in ingresso per il primo problema sono presi da un algoritmo tempo-polinomiale, che produce i dati in ingresso per il secondo problema, in modo tale che la risposta del secondo problema sia esattamente quella che darebbe il primo.

Dunque, tutti i problemi appartenenti alla sottoclasse degli NP-completi presentano lo stesso livello di difficoltà (intesa come laboriosità richiesta per risolverli) e ognuno di essi può essere trasformato efficientemente in ciascun altro appartenente alla stessa sottoclasse.

Tutto ciò implica che, qualora si trovasse un algoritmo tempo-polinomiale per risolvere uno qualsiasi dei problemi NP-completi, ecco che si sarebbe trovata una soluzione efficiente per *tutti* i problemi in NP, e quindi l’intera classe NP “collasserebbe” nella classe P.

Per concludere che le due classi sono diverse, sarebbe invece sufficiente provare la non esistenza di un algoritmo tempo-polinomiale per uno qualsiasi dei problemi in NP.

In un fondamentale articolo del 1971, Stephen A. Cook dimostrò che qualsiasi problema in NP può essere trasformato nel problema della *soddisfattibilità*, e quindi fu questo il primo problema NP-completo ad essere individuato.

1. Problema della *soddisfattibilità* (SAT): data una proposizione formata dalla congiunzione ( $\wedge$ ) di *clausole*, ciascuna costituita da una disgiunzione ( $\vee$ ) di *letterali* (cioè di variabili *booleane* o di loro negazioni), stabilire se esiste una assegnazione di valori di verità alle variabili che vi occorrono, tale che la proposizione risulti vera.

Lo stesso si può ancora affermare se ci limitiamo alle clausole con, al più, tre letterali, oppure con esattamente tre letterali (in quest'ultimo caso si parla di problema 3SAT).

Se invece consideriamo proposizioni formate da clausole con *due* letterali (2SAT), allora il problema è risolvibile in tempo lineare, e dunque è in P.

Senza perdita di generalità, possiamo supporre che in ciascuna clausola la stessa variabile non occorra più di una volta. (Perché?)

Ad esempio, data la proposizione  $(\sim x_1 \vee x_2 \vee \sim x_3) \wedge (x_1) \wedge (\sim x_2 \vee x_3)$ , la risposta sarà “sì”; in questo caso ci sono due assegnazioni che la rendono vera:  $(x_1 = \text{vero}, x_2 = \text{falso}, x_3 = \text{falso})$  e  $(x_1 = \text{vero}, x_2 = \text{vera}, x_3 = \text{vera})$ . Tale istanza del problema può essere trasformata nell’istanza di 3SAT che prende in ingresso la proposizione

$$\begin{aligned} &(\sim x_1 \vee x_2 \vee \sim x_3) \wedge \\ &(x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \sim y_2) \wedge (x_1 \vee \sim y_1 \vee y_2) \wedge (x_1 \vee \sim y_1 \vee \sim y_2) \wedge \\ &(\sim x_2 \vee x_3 \vee y_3) \wedge (\sim x_2 \vee x_3 \vee \sim y_3). \end{aligned}$$

Ciascuna clausola costituita da un solo letterale comporta l’introduzione di due nuove variabili, mentre ciascuna clausola con due letterali comporta l’introduzione di una nuova variabile.

D’altra parte, se nella proposizione originaria comparisse ad esempio  $(\sim x_1 \vee x_2 \vee \sim x_3 \vee x_4 \vee x_5)$ , che è una clausola con cinque letterali, la potremmo sostituire con

$$\begin{aligned} &(\sim x_1 \vee x_2 \vee y_4) \wedge (x_3 \vee y_4) \wedge (\sim x_4 \vee y_4) \wedge (\sim x_5 \vee y_4) \wedge \\ &(\sim x_3 \vee x_4 \vee y_5) \wedge (x_5 \vee \sim y_4 \vee \sim y_5) \wedge (\sim x_5 \vee y_5) \wedge (y_4 \vee y_5) \end{aligned}$$

introducendo due nuove variabili.

(Sulla scorta di questi esempi, si dovrebbe riuscire a generalizzare senza difficoltà... Come si prova che questo genere di trasformazione è tempo-polinomiale?)

Un’interessante variante di SAT, che rimane NP-completa, è detta CIRCUIT-SAT: dato un circuito combinatorio, costituito da porte logiche AND e OR (a due ingressi e un’uscita) e NOT (a un ingresso e un’uscita), con un numero arbitrario di ingressi

binari e una sola uscita binaria, stabilire se esiste una assegnazione di valori binari agli ingressi tale che il valore in uscita sia 1.

Durante il successivo anno 1972, Richard M. Karp allargò la famiglia, individuando una ventina di altri problemi, appartenenti alla classe NP, nei quali poteva essere trasformato efficientemente (anche per transitività) il problema “capostipite” della soddisfattibilità: quindi anch’essi erano NP-completi. Alcuni li abbiamo già incontrati o li ritroveremo nel corso del capitolo, magari in forma un po’ diversa ma sostanzialmente equivalente. Le prove di Karp seguono questa linea: si mostra come SAT possa essere trasformato in un altro problema, quindi si mostra come questo possa essere trasformato in un altro ancora, e così via; l’ultimo anello della catena sarà trasformabile in SAT grazie al teorema di Cook, e così è provata la sostanziale equivalenza dei problemi coinvolti in questa dimostrazione.

In seguito furono individuati tanti altri problemi NP-completi, mediante “trasformazioni” talvolta assai ingegnose; l’importante testo *Computers and Intractability: A Guide to the Theory of NP-Completeness* di Michael R. Garey e David S. Johnson, pubblicato nel 1979, ne riportava già più di 300!

Seguitiamo con il nostro breve elenco, invitando il lettore a riflettere su ciascun problema, per cercare di intuire in che cosa possa consistere un testimone controllabile in tempo polinomiale, da produrre in caso di risposta affermativa. Salvo avviso contrario, quando diremo “grafo” intenderemo non orientato e privo di cappi, e quando diremo “numero” intenderemo intero positivo.

2. Problema del *sottografo completo* (CLIQUE): dati un grafo  $G$  e un numero  $k$ , stabilire se in  $G$  esiste un insieme di  $k$  nodi a due a due adiacenti (cioè: un sottografo di  $k$  nodi *completo*, in cui ciascuna coppia di nodi sia collegata da un arco).

(Non è difficile trasformare un’istanza di SAT in un’istanza di CLIQUE...)

3. Problema dell’*insieme indipendente* (o *stabile*): dati un grafo  $G$  e un numero  $k$ , stabilire se in  $G$  esiste un insieme di  $k$  nodi a due a due *non* adiacenti (cioè: nessuna coppia di nodi appartenenti a tale insieme è collegata da un arco).

Rimane NP-completo anche imponendo che i nodi abbiano, al più, quattro archi incidenti (un arco è incidente su un nodo quando ha un estremo in tale nodo).

È immediato constatare che un insieme indipendente è costituito dai nodi di un sottografo completo (*clique*) nel grafo *complementare*, ossia nel grafo che si ottiene da quello dato togliendo tutti gli archi e mettendo nuovi archi (tra coppie di nodi distinti) esattamente dove prima non c’erano.

4. Problema del *ciclo hamiltoniano*: già enunciato.

*Idem* per il *cammino hamiltoniano*: si deve stabilire se esiste un percorso che tocchi tutti i nodi una e una sola volta ciascuno, ma non è richiesto che sia chiuso, cioè non è detto che vi sia un arco dal nodo di arrivo a quello di partenza. La più famosa versione del problema del ciclo hamiltoniano, ugualmente impegnativa, è nota come problema del *commesso viaggiatore* (*travelling-*

*salesman problem* o TSP): fa riferimento a un dato grafo *pesato*, ove sia assegnato un costo intero positivo a ciascun arco (ad esempio, la distanza tra i due nodi – le città – che collega), e richiede di stabilire se esiste un ciclo hamiltoniano che abbia costo minore o uguale a un dato numero  $C$ . Come costo del ciclo s'intende, ovviamente, la somma dei costi degli archi che lo compongono.

Questi problemi rimangono NP-completi anche nell'ipotesi (più generale) in cui il grafo dato in ingresso sia orientato.

Possibili varianti prevedono come dato aggiuntivo un numero  $k$  e chiedono se esista un ciclo costituito da almeno  $k$  nodi (senza ripetizioni).

Un altro interessante problema NP-completo ha due nodi come ulteriori dati: bisogna decidere se tra questi vi sia un cammino costituito da almeno  $k$  archi (senza ripetizioni); o ancora, quando il grafo è pesato, se vi sia un cammino di costo *maggior* o uguale a  $k$ . Si ricordi che trovare invece i *cammini di costo minimo* per tutte le coppie di nodi ha complessità polinomiale.

Determinare i cammini di costo o minimo o massimo a partire da un dato nodo, in un grafo orientato aciclico, richiede tempo lineare nella somma di nodi e archi.

5. Problema della *colorabilità dei nodi*: dati un grafo  $G$  e un numero  $k > 2$ , stabilire se i nodi di  $G$  possono essere colorati usando  $k$  colori, in maniera tale che non vi siano due nodi adiacenti di stesso colore.

Anche la seguente variante del problema è NP-completa: dato un grafo  $G$  *planare* (cioè che possa essere disegnato su un piano in modo da non intersecare archi), stabilire se  $G$  è colorabile con *tre* colori (L. J. Stockmeyer, 1973); *idem* se pur si vincola ciascun nodo del grafo planare ad avere al massimo quattro archi incidenti su di esso.

Un qualsiasi grafo planare è colorabile con (al più) *quattro* colori. (Di questo celebre “teorema dei quattro colori” abbiamo già parlato nel libretto del 2010, a proposito del quesito “Colora la mappa”.)

Decidere se un grafo arbitrario possa essere colorato usando soltanto *due* colori è questione risolvibile in tempo lineare, e dunque ascrivibile nella classe P. Equivale infatti a chiedersi se esso *non* ha cicli formati da un numero dispari di archi (D. König, 1916). L’idea consiste nel colorare un nodo e, di conseguenza, assegnare l’altro colore a tutti i nodi ad esso adiacenti; poi si ripete l’operazione per ciascuno dei nodi colorati... In sostanza, si visita il grafo *in ampiezza* (con l’algoritmo di *breadth-first search*), finché è possibile continuare la colorazione. Se consideriamo grafi arbitrari, allora non v’è limite al numero di colori che possono essere richiesti (si pensi ai grafi completi); tuttavia ci sono grafi non planari i cui nodi possono essere colorati usando due soli colori: sapete trovarne uno?

Giacché abbiamo parlato di grafi planari, ricordiamo che, restringendo l’input a tale classe, i problemi del ciclo hamiltoniano e dell’insieme indipendente rimangono NP-completi, mentre CLIQUE si semplifica assai, poiché in un grafo planare non possono esservi sottografi completi comprendenti più di quattro

nodi, e il compito di trovarvi tutti quelli di quattro nodi può essere svolto da un algoritmo tempo-lineare...

Inoltre, poiché un grafo planare con  $n$  nodi può avere, al più,  $3n$  archi, si può stabilire in tempo lineare se un dato grafo è planare (J. E. Hopcroft e R. E. Tarjan, 1974). Come si caratterizza un grafo non planare? (Si veda il teorema formulato nel 1930 da K. Kuratowski.)

6. Problema delle *equazioni diofantee* (o *diofantine*) *quadratiche*: dati tre numeri  $a$ ,  $b$  e  $c$ , stabilire se esistono due numeri (sempre interi e positivi)  $x$  e  $y$  tali che  $ax^2 + by = c$ .

In generale, il problema di stabilire se un'equazione polinomiale indeterminata a coefficienti interi (per essere certi: di quarto grado e in 11 variabili, o più) ammetta una soluzione intera è provatamente *indecidibile*: una procedura effettiva potrà soltanto semi-deciderlo (Ju. V. Matijasevič, 1970).

7. Problema dello *zaino* (KNAPSACK): dati due numeri,  $P$  e  $V$ , e date due liste di  $n$  numeri ciascuna,  $(p_1, \dots, p_n)$  e  $(v_1, \dots, v_n)$ , che rappresentino rispettivamente i pesi e i valori di  $n$  oggetti, stabilire se esiste un insieme di oggetti il cui peso complessivo sia, al più,  $P$  (il peso massimo sopportato dallo zaino) e il cui valore complessivo sia almeno  $V$  (il valore minimo che si desidera trasportare).

La formulazione di Karp era più semplice, ma ugualmente impegnativa da risolvere: parlando soltanto dei pesi, chiedeva di stabilire se lo zaino può essere riempito esattamente col suo peso massimo (si noti che ciò corrisponde a fissare  $V = P$  e ciascun  $v_i$  uguale al rispettivo  $p_i$ ). È pure utile osservare che questo equivale a chiedersi se l'equazione lineare

$$p_1 x_1 + p_2 x_2 + \dots + p_n x_n = P$$

abbia soluzioni in cui ciascuna incognita  $x_i$  valga 0 o 1 (l'oggetto  $i$ -esimo è escluso o è messo nello zaino, rispettivamente). Questa formulazione è nota col nome di SUBSET-SUM e si può ritrovare, ad esempio, in crittografia. In generale, porsi la stessa domanda per un *sistema* di equazioni lineari a coefficienti *interi* non cambia la complessità del problema, che viene detto di *programmazione (lineare) intera 0-1*.

8. Problema della *partizione*: data una lista di numeri (interi positivi o interi qualsiasi) la cui somma sia pari, stabilire se tali numeri possono essere ripartiti in due liste di uguale somma. (Il problema rimane NP-completo se si aggiunge «e contenenti lo stesso numero di elementi», ammesso che gli elementi della lista data siano in numero pari.)

Si verifica immediatamente che si tratta di un caso particolare del problema di SUBSET-SUM, e quindi non potrebbe comunque rivelarsi più laborioso di quest'ultimo.

D'altra parte, la generica istanza di SUBSET-SUM può essere trasformata nell'istanza del problema della partizione che prende la seguente lista:

$$(p_1, \dots, p_n, P+1, s+1-P),$$

dove  $s$  è la somma dei  $p_i$  (perché?). Quindi i due problemi presentano uguale difficoltà.

9. Problema dell'*imballaggio* (BIN-PACKING): dati due numeri,  $k =$  numero di contenitori (o *bin*) disponibili, tutti uguali, e  $P =$  peso massimo sopportato da ciascun contenitore, e data una lista di  $n$  numeri  $(p_1, \dots, p_n)$  che rappresentano i pesi di  $n$  oggetti, stabilire se tutti questi  $n$  oggetti possono essere ripartiti nei  $k$  contenitori in modo tale che la somma dei pesi in ciascun contenitore sia, al più,  $P$ .
10. Problema di *sequencing* (JOB-SEQUENCING): dato un numero  $P$  e date tre liste di  $n$  numeri ciascuna:  $(t_1, \dots, t_n)$ ,  $(d_1, \dots, d_n)$  e  $(p_1, \dots, p_n)$ , le quali rappresentino rispettivamente: le durate di  $n$  lavori, gli istanti di tempo desiderati per la loro conclusione (l'istante 0 coinciderà con l'inizio del primo lavoro) e le penalità da pagare per un eventuale loro ritardo (anche minimo) sui tempi desiderati, stabilire se esiste un ordine di esecuzione degli  $n$  lavori (in sequenza), tale che la somma delle penalità relative ai lavori terminati in ritardo sia, al più,  $P$ .

È interessante fare un paio di considerazioni su questo problema. La prima è che rimane NP-completo anche nel caso particolare in cui i  $d_i$  siano imposti tutti uguali, il che significa dare un solo tempo, desiderato per la conclusione di tutti i lavori, e di conseguenza le penalità da pagare saranno quelle relative ai lavori che termineranno dopo tale scadenza. Si può cogliere l'analogia con il problema dello zaino: in effetti, è assai facile trasformare KNAPSACK in JOB-SEQUENCING. La seconda osservazione è che se invece si impongono tutte uguali le penalità (ad esempio a 1, il che significa dover stabilire se i lavori terminati in ritardo potranno essere, al più,  $P$ ), allora il problema è risolvibile in tempo polinomiale. (Come?)

Una variante che resta NP-completa prevede, al posto delle penalità, gli istanti di tempo a partire dai quali i lavori possono essere iniziati: la decisione riguarda la possibilità o meno di rispettare *tutti* i tempi.

11. Problema del *sottografo isomorfo*: dati due grafi, stabilire se il primo è isomorfo a un sottografo del secondo (un sottografo si ottiene eventualmente cancellando alcuni archi e/o alcuni nodi con tutti gli archi su di essi incidenti). Due grafi sono isomorfi se risultano identici, eventualmente dopo aver rinumerato in maniera opportuna i nodi di uno dei due, sicché le loro rappresentazioni grafiche saranno equivalenti dal punto di vista topologico.

Per inciso, il problema dell'*isomorfismo tra grafi* (dati due grafi arbitrari, stabilire se sono isomorfi) è uno dei pochi problemi noti che appartengono a NP ma per i quali non è stato dimostrato né che siano NP-completi, né che appartengano a P.

(Si veda il quesito “L’amico sconosciuto”, nel libretto del 2015.)

Per grafi planari, il problema del sottografo isomorfo rimane NP-completo, mentre quello dell'isomorfismo tra grafi è risolvibile con un algoritmo tempo-lineare (J. E. Hopcroft e J. K. Wong, 1974).

Un elenco di problemi NP-completi, scelti e commentati, si può trovare alla pagina [http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated\\_np.html](http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html).

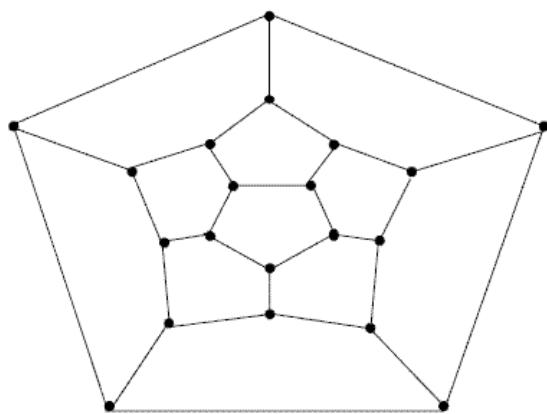
## Cicli hamiltoniani e torri di Hanoi.

Nel precedente paragrafo abbiamo parlato di cicli hamiltoniani, i quali sono così chiamati in onore di William Rowan Hamilton (1805-1865), fisico e astronomo irlandese, che sviluppò un'interpretazione algebrica dei numeri complessi e la cosiddetta “teoria dei quaternioni”, anticipatrice della moderna analisi vettoriale.

In tale contesto, egli ideò un'algebra basata sulle proprietà di simmetria dell'icosaedro regolare, il solido platonico le cui 20 facce sono triangoli equilateri (dal greco antico εἴκοσι, ossia “venti”), e nel 1857 propose un gioco, che denominò *icosian game*: a partire da un vertice di un dodecaedro regolare (altro solido platonico, con 12 facce pentagonali) si dovevano percorrere 20 (dei 30) spigoli, passando una e una sola volta per ciascun vertice e ritornando infine al vertice di partenza.

Se si numerano i vertici, vi si possono trovare 30 cicli hamiltoniani distinti (a prescindere dal senso di percorrenza del ciclo), ottenibili per una qualche simmetria (anche speculare) da un unico ciclo di base.

Dal punto di vista topologico, il grafo qui sotto disegnato è equivalente al dodecaedro.



Il problema di trovare un ciclo del genere, in un grafo specifico, se lo era già posto persino il grande Eulero (Leonhard Euler, 1707-1783), compiendo un'astrazione mentre indagava sui giri chiusi del cavallo sulla scacchiera: di questo tratteremo in un capitolo successivo.

Nel 1880, Peter G. Tait (anche di lui parleremo in seguito) congetturò che ogni poliedro in cui ciascun vertice è a capo di tre spigoli fosse hamiltoniano (cioè ammettesse un ciclo di tal genere attraverso i suoi vertici). Ad esempio, i solidi

platonici e archimedei sono hamiltoniani, ma non tutti hanno esclusivamente vertici sui quali incidono tre spigoli. Soltanto nel 1946 il matematico e crittoanalista inglese William T. Tutte trovò il primo controesempio che demoliva la congettura di Tait (il grafo corrispondente, con 46 nodi, è oggi conosciuto come *grafo di Tutte*); il controesempio più piccolo è stato trovato nel 1965: il grafo di Barnette-Bosák-Lederberg, che ha 38 nodi. Per inciso, Joshua Lederberg, di professione biologo, nel 1958 era stato insignito del premio Nobel per la medicina!

Il matematico francese Édouard Lucas (1842-1891) – che ritroveremo più volte nella nostra esposizione, e non soltanto nelle occasioni a tema ludico – tratta del gioco di Hamilton e di sue trasformazioni nel secondo volume delle *Récréations mathématiques*, concludendo con la proposta di giocare un solitario concepito nel seguente modo. Si dispongono inizialmente venti pedine sui nodi del grafo sopra riportato, se ne toglie una da un certo nodo e poi, a una a una, si “mangiano” le altre: una pedina ne può mangiare un’altra ad essa adiacente, saltandovi sopra, per finire in un altro nodo libero adiacente a quella mangiata. L’ultima pedina deve rimanere in un nodo prestabilito. Lucas dimostra che la riuscita di questo solitario è sempre possibile, qualunque sia la coppia di nodi – quello libero iniziale e quello da occupare alla fine con l’ultima pedina rimasta – che è stata fissata.

È proprio Lucas l’inventore della torre di Hanoi, venduta come *puzzle* dal 1883. Ci sono tre pioli, che possiamo immaginare disposti ai vertici di un triangolo; su uno di essi sono impilati  $n$  dischi forati, di diverso diametro, dal più grande alla base al più piccolo in alto. Si tratta di ricostruire la torre su uno degli altri due pioli, spostando un disco alla volta e senza mai posarne uno sopra un altro di diametro inferiore.

Al duraturo successo di questo puzzle ha in larga parte contribuito la predilezione degli informatici! Il tempo che occorre per eseguire tutti gli spostamenti cresce esponenzialmente con  $n$  – e non v’è rimedio – ma per farlo risolvere al computer basta scrivere una procedura ricorsiva di poche righe, che richiama due volte sé stessa (e lo spazio di memoria interna che richiede, output escluso, aumenta linearmente con  $n$ ): si può trovarla in molti testi di programmazione.

A noi qui interessa un altro procedimento risolutivo: nelle mosse di ordine dispari (la prima, la terza, ...) si trasferisce di un piolo il disco più piccolo procedendo sempre in uno dei due sensi (ad esempio quello orario); nelle mosse di ordine pari si fa invece l’unico trasferimento possibile che non coinvolga il disco più piccolo.

Ad esempio, numerando i dischi da 1 (il più piccolo in alto) a 3 (il più grande alla base) di una torre di tre dischi, si muovono nell’ordine: 1-2-1-3-1-2-1; se ci fosse un disco ancora più grande, numerato 4, la sequenza risolutiva sarebbe quella di prima, seguita da 4 e poi ancora una volta dalla precedente: 1-2-1-3-1-2-1-4-1-2-1-3-1-2-1. E così via.

Che cosa c’entra la torre di Hanoi con i cicli hamiltoniani lungo un solido? Come riporta Martin Gardner nella sua raccolta *Hexaflexagons, Probability Paradoxes, and the Tower of Hanoi*, Donald W. Crowe ha osservato che la sequenza di mosse che risolve la torre di Hanoi con  $n$  dischi corrisponde esattamente a un cammino hamiltoniano sugli spigoli di un ipercubo nello spazio a  $n$  dimensioni: basta mettere

in corrispondenza biunivoca i dischi con le dimensioni dello spazio lungo le quali ci si dovrà muovere di volta in volta.

Ad esempio, nel caso  $n = 3$ , la sequenza 1-2-1-3-1-2-1 può essere messa in corrispondenza con i seguenti movimenti lungo gli assi dello spazio tridimensionale:  $x-y-x-z-x-y-x$ . D'altra parte, se il cubo considerato ha un vertice nel punto di coordinate  $(0, 0, 0)$  e tre lati unitari disposti lungo il verso positivo dei tre assi, e se si inizia dal vertice  $(0, 0, 0)$ , il suddetto cammino toccherà, nell'ordine, i vertici  $(1, 0, 0), (1, 1, 0), (0, 1, 0), (0, 1, 1), (1, 1, 1), (1, 0, 1)$  e  $(0, 0, 1)$ . Se li riguardiamo come terne di bit, poiché da un vertice al successivo cambia un solo bit, essi costituiscono un *codice di Gray* (Frank Gray, 1953). I codici con questa proprietà hanno trovato applicazione nelle telecomunicazioni, nelle codifiche a correzione di errore e in crittografia.

Gardner sosteneva che per un matematico poche esperienze sono più eccitanti dello scoprire che due strutture apparentemente estranee l'una all'altra sono in realtà strettamente collegate!

### **Problemi di ottimizzazione combinatoria: il commesso viaggiatore.**

Ancor più frequentemente di quelli *decisionali*, ai quali abbiamo accennato, nella pratica si incontrano problemi di *ottimizzazione combinatoria* (o *discreta*), che consistono nella massimizzazione di una funzione obiettivo, o nella minimizzazione di una funzione di costo, rispetto ai modi alternativi in cui diversi elementi di un insieme finito possono essere scelti, disposti e ordinati, nello spazio o nel tempo.

Non ci si accontenta di decidere se il problema ammette almeno una soluzione, bensì si cerca la migliore, o una delle migliori (nei casi in cui più soluzioni siano ugualmente ottime), secondo certi criteri, di solito facilmente specificabili. Ciò che invece non è dato esplicitamente è l'insieme (finito) delle soluzioni ammissibili, che dipende dalla struttura del problema; e, per esplorare con “intelligenza” quell'insieme, un algoritmo dovrà sfruttare in modo opportuno tale struttura.

Dalla metà del Novecento, i ricercatori nel campo dell'ottimizzazione combinatoria hanno trovato algoritmi sempre più raffinati e di più elevate prestazioni. I problemi di questo tipo, però, si rivelano spesso non meno impegnativi di quelli che abbiamo elencato nel precedente paragrafo: quand'è così, se si vuole risolverli in generale e in maniera esatta, si deve ricorrere ad algoritmi “onerosi”, che sostanzialmente operano per enumerazione (o *esaurimento*) di tutti i casi possibili, escludendone, al più, una parte non preponderante. Si tratta di algoritmi che avviano una ricerca sistematica della soluzione desiderata, fino a trovarla o a rilevarne la non esistenza (qualora la particolare istanza del problema in esame non ammetta alcuna soluzione), e in definitiva risultano esponenziali o comunque super-polynomiali (o magari, per problemi particolari, pseudo-polynomiali, come vedremo nel caso dello zaino).

Un celebre esempio è il problema di ottimizzazione del *commesso viaggiatore* (TSP): a partire da un nodo (città) scelto a piacere, si deve trovare un ciclo hamiltoniano di

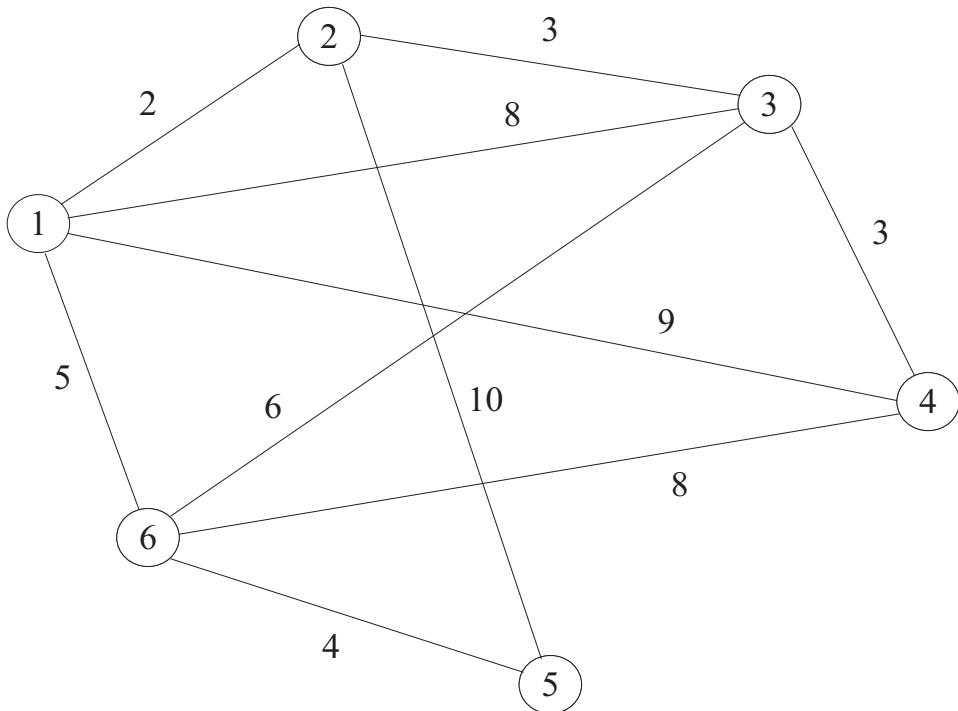
costo *minimo* (cioè minore o uguale al costo associato a qualsiasi altro ciclo hamiltoniano nel grafo dato). Dunque, rispetto al corrispondente problema decisionale, è richiesto un dato di input in meno: il numero  $C$ , ossia il costo massimo del ciclo di cui si doveva stabilire l'esistenza o meno.

Il quesito “Città d'Europa”, che trovate nel libretto della gara *Kangourou dell'Informatica* di marzo 2009, ne proponeva un'istanza, facile ma istruttiva.

Qualcuno potrebbe obiettare: ma se interessasse sapere soltanto qual è il costo minimo? Bisognerebbe ugualmente arrivare a conoscere un ciclo che abbia tale costo!

Il problema decisionale non può essere più laborioso di quello di ottimizzazione: il costo minimo fornito da quest'ultimo può essere subito confrontato con  $C$ , per dare di conseguenza la risposta al problema decisionale. In realtà, sebbene per via più complicata, si arriva a dimostrare anche il viceversa, e cioè che il problema di ottimizzazione non è sostanzialmente “più difficile”, sotto l'aspetto computazionale, del problema decisionale.

Il nodo di partenza può essere prefissato, senza perdita di generalità. Il grafo può essere orientato oppure no; nell'esempio sotto riportato, il grafo (che, per inciso, è planare) non è orientato: ciò significa che gli archi sono percorribili nei due sensi (con lo stesso costo), e quindi ciascun ciclo può essere percorso in due modi diversi, l'uno rovesciato rispetto all'altro (con lo stesso costo). (Di solito, se il grafo è orientato è perché qualche arco di ritorno manca oppure presenta un costo differente rispetto a quello di andata: si tratta allora dell'altrettanto interessante e più generale TSP *asimmetrico*.)



Qui vi sono soltanto quattro cicli hamiltoniani:

- [1, 2, 5, 6, 3, 4, 1] con costo 34;
- [1, 4, 3, 2, 5, 6, 1] con costo 34;
- [1, 2, 5, 6, 4, 3, 1] con costo 35;
- [1, 3, 2, 5, 6, 4, 1] con costo 42.

Uno dei primi due indifferentemente (o il suo “rovescio”) costituisce la soluzione di questa istanza del TSP.

Vediamo come si può specificare un semplice algoritmo enumerativo esatto, valido in generale, anche nel caso asimmetrico, che usi come “variabili globali” i seguenti oggetti:

- una descrizione del grafo, ad esempio sotto forma di *matrice dei costi* (qui simmetrica)

0	2	8	9	$+\infty$	5
2	0	3	$+\infty$	10	$+\infty$
8	3	0	3	$+\infty$	6
9	$+\infty$	3	0	$+\infty$	8
$+\infty$	10	$+\infty$	$+\infty$	0	4
5	$+\infty$	6	8	4	0

dove  $+\infty$  indica l’assenza di arco, ossia di collegamento diretto (potrebbe essere rappresentato da un numero molto grande, prefissato o comunque maggiore della somma dei costi di tutti gli archi, ma la cosa più sicura è servirsi anche della matrice di adiacenza, considerando come costi effettivi i soli valori che corrispondono a un 1 in tale matrice);

- una variabile *costo\_minimo* di tipo intero positivo, con valore iniziale  $+\infty$  (qui sì che indica un numero sufficientemente grande);
- una variabile *cammino\_minimo* di tipo lista di interi positivi, se decidiamo di numerare i nodi a iniziare da 1, in accordo con le matrici che descrivono il grafo; se conveniamo di partire dal nodo 1, assegniamo inizialmente il valore [1] alla variabile *cammino\_minimo*.

L’esecuzione dell’algoritmo non modificherà le matrici che descrivono il grafo; tutte le volte che esiste almeno un circuito hamiltoniano, alla fine le variabili *cammino\_minimo* e *costo\_minimo* conterranno rispettivamente la soluzione trovata e il relativo costo, altrimenti non saranno modificate.

Nel seguito indicheremo con  $C(i, j)$  il costo dell’arco diretto dal nodo  $i$  al nodo  $j$ , ossia il valore dell’elemento di riga  $i$  e colonna  $j$  nella matrice dei costi.

Il calcolo è innescato dalla chiamata TSP ([1], 0) della procedura ricorsiva TSP, con due parametri (in cui sono trasmessi i valori degli argomenti), così specificata:

TSP (*percorso\_corrente*, *costo\_corrente*):

*i*  $\leftarrow$  ultimo nodo di *percorso\_corrente*  
*non\_visitati*  $\leftarrow$  lista dei nodi del grafo eccetto quelli elencati in *percorso\_corrente*  
se *non\_visitati* contiene almeno un nodo **allora**  
    **per ogni** *j* contenuto in *non\_visitati* e tale che esista l'arco da *i* a *j*  
        *c*  $\leftarrow$  *costo\_corrente* + C(*i*, *j*)  
    (\*)   **se** *c* < *costo\_minimo* **allora**  
        *tentativo*  $\leftarrow$  *percorso\_corrente* + *j* (aggiunto come ultimo)  
        TSP (*tentativo*, *c*) // chiamata ricorsiva

**altrimenti**

// *non\_visitati* è vuota, ossia *percorso\_corrente* contiene tutti i nodi del grafo  
    **se** esiste l'arco da *i* a 1 e *costo\_corrente* + C(*i*, 1) < *costo\_minimo* **allora**  
        *cammino\_minimo*  $\leftarrow$  *percorso\_corrente* + 1 (aggiunto come ultimo)  
        *costo\_minimo*  $\leftarrow$  *costo\_corrente* + C(*i*, 1)

Se nel grafo dato non esiste alcun circuito hamiltoniano, la ricerca si arresta ugualmente, dopo aver esplorato tutti i possibili percorsi (aciclici) a partire dal nodo 1, e in *cammino\_minimo* rimane il valore iniziale [1].

La chiamata ricorsiva è subordinata al test (\*): se l'arco che stiamo per aggiungere al percorso corrente determina un costo corrente che già raggiunge o supera il minimo costo finora trovato di un intero circuito hamiltoniano, allora è inutile proseguire su quella strada. Secondo la nostra specifica, è questo l'unico motivo che può evitarcì l'esplorazione di *tutti* i possibili percorsi; ma tale eventualità non capiterà mai nei casi più sfortunati, e neanche mediamente possiamo attenderci un beneficio tale da migliorare in ordine di grandezza le prestazioni della procedura rispetto alla versione senza la linea (\*). Il test (\*) costa un solo confronto tra i contenuti di due variabili intere, e può migliorare le cose almeno in certi casi, perciò senza dubbio conviene prevederlo piuttosto che no. In effetti, è proprio questa la prima, ovvia tecnica "di interruzione" (dell'esplorazione di un percorso) applicabile al TSP, così come agli altri problemi di natura combinatoria, quando si vuol procedere con la semplice enumerazione delle soluzioni ammissibili.

Supponendo che nella lista (locale) *non\_visitati* i numeri che rappresentano i nodi siano ordinati in senso ascendente, quali passi farà l'algoritmo nell'esempio proposto, e quale delle due soluzioni di costo 34 infine produrrà? Prima del ciclo "per ogni", potrebbe risultare conveniente un ordinamento della lista *non\_visitati* secondo un qualche criterio?

Suggeriamo inoltre di riflettere sulle modifiche da apportare alla procedura TSP affinché trovi, se c'è, un *cammino* hamiltoniano ottimo a partire da un nodo dato.

Ci sono idee alternative, e semplici, su come affrontare in generale il TSP? Una regola *euristica* di facile applicazione si basa su una tecnica "ingorda" (*greedy*): ad ogni passo, percorriamo l'arco (o uno degli archi) di costo minore, tra quelli che dal nodo corrente portano a un nodo non ancora visitato. Tuttavia questo algoritmo, chiamato *nearest neighbour*, non garantisce affatto di trovare una soluzione!

Applicato al nostro esempio, troverebbe il ciclo di costo 35 (che non è l'ottimo), ma soltanto se partisse dal nodo 3 e scegliesse al primo passo di portarsi nel nodo 4,

oppure se partisse dal nodo 6; negli altri casi non troverebbe alcun ciclo. Si possono dare esempi in cui una soluzione esiste, ma non è trovata a partire da alcun nodo, a meno di dotare l'algoritmo della capacità di tornare sui propri passi per tentare strade alternative. Ovviamente, la completezza del grafo (cioè l'esistenza di un arco tra ogni coppia di nodi distinti) è una condizione sufficiente per il reperimento di una soluzione, a prescindere dal nodo di partenza; tuttavia, la bontà della soluzione trovata può dipendere anche sensibilmente dalla scelta del nodo di partenza.

Supponiamo pure, in generale, di studiare il problema di ottimizzazione su grafi completi e non orientati, per cui il costo da  $i$  a  $j$  è uguale al costo da  $j$  a  $i$  per ogni coppia di nodi  $i$  e  $j$ , e inoltre una soluzione esiste sempre. Se il grafo ha  $n$  nodi, i possibili cicli hamiltoniani distinti (ma non per senso di percorrenza) sono  $(n-1)!/2$ ; quando  $n$  è 20, questo numero è circa  $6 \cdot 10^{16}$  (60 milioni di miliardi), e ad esaminarli al frenetico ritmo di un miliardo al secondo s'impiegherebbe un paio d'anni! Ciò significa che un algoritmo di “forza bruta” non ci è di alcuna utilità.

Possiamo almeno sperare di cavarsela impiegando un *algoritmo di approssimazione* tempo-polonomiale che ci garantisca di trovare sempre un ciclo di costo non superiore a quello minimo moltiplicato per un fattore costante? (Questo fattore sarà  $\geq 1$ ; ad esempio, 1.25 significa ottenere una soluzione il cui costo sarà al massimo il 25% in più dell'ottimo, mentre 1 corrisponde al risultato fornito dall'algoritmo esatto.) Purtroppo, per questo problema davvero “difficile”, la risposta è negativa (S. Sahni e T. Gonzales, 1976), a meno che, s'intende, sia  $P = NP$ .

Dobbiamo gettare la spugna? Assolutamente no! Già nel lontano 1954, George B. Dantzig, Delbert R. Fulkerson e Selmer M. Johnson pubblicarono un articolo in cui descrivevano un metodo per risolvere all'ottimo il problema e ne illustravano la potenza su un esempio che comprendeva ben 42 città degli Stati Uniti: una dimensione impressionante per quell'epoca! Il loro metodo prese avvio da quello del *simplesso*, ideato da Dantzig nel 1947 per risolvere problemi di *programmazione lineare* “nel continuo” e già collaudato con successo in numerose e varie applicazioni.<sup>17</sup>

---

<sup>17</sup> Il termine “programmazione” qui ha davvero poco a che fare con la stesura di codici per computer: si riferisce piuttosto alla *pianificazione* degli impieghi di certe risorse al fine di rendere massimo il profitto che ne deriva. L'etimologia di “programmazione dinamica” è simile: tale locuzione fu concepita per l'ottimizzazione di processi multi-stadio, in particolare per quelli la cui evoluzione possa essere descritta da un grafo orientato privo di cicli. In verità, la programmazione lineare e la programmazione dinamica sono due tra le più generali tecniche algoritmiche.

Per inciso, l'algoritmo del simplesso è tempo-esponenziale nei casi peggiori, ma nella quasi totalità delle applicazioni pratiche funziona assai meglio di un algoritmo tempo-polonomiale trovato nel 1979 dall'armeno Leonid G. Khachiyan, al quale spetta comunque il merito di aver provato che il problema generale della programmazione lineare *nel continuo* appartiene alla classe P (“nel continuo” qui significa che le incognite, ossia gli impieghi delle risorse, possono assumere valori razionali o reali). Pochi anni dopo, nel 1984, l'indiano Narendra K. Karmarkar sviluppò un'idea completamente diversa, e ideò un nuovo algoritmo tempo-polonomiale: si scoprì che questo funzionava bene nella pratica, e da allora ebbe origine un'accanita competizione col metodo del simplesso, che portò allo sviluppo di software sempre più efficienti per risolvere problemi di programmazione lineare.

In particolare, il TSP (anche asimmetrico) può essere impostato come problema di programmazione intera 0-1: ad ogni arco è associata un'incognita binaria (che assumerà valore 1 se l'arco farà parte della soluzione, 0 altrimenti) e la funzione (lineare) da minimizzare è la sommatoria di tutte le incognite, ciascuna pesata col costo del corrispondente arco. Gli ulteriori vincoli esprimono il fatto che una soluzione ammissibile deve prevedere per ciascun nodo un arco entrante e uno uscente e, inoltre, l'inesistenza di sotto-circuiti. Questi ultimi vincoli possono essere formalizzati efficacemente nel caso simmetrico, sfruttando certe proprietà dei sottografi.

Se si rimuovesse il vincolo di interezza (qui addirittura di “binarietà”) sulle incognite, si avrebbe il cosiddetto *problema rilassato*, risolvibile col metodo del simplex.

### Un algoritmo di *branch-and-bound*.

Nel 1960, nell’ambito della programmazione lineare a incognite tutte o in parte discrete, A. H. Land e A. G. Doig proposero una tecnica generale, poi detta *branch-and-bound* (“ramifica e limita”), per simulare una completa enumerazione di tutte le soluzioni ammissibili senza doverle considerare a una a una. Per molti problemi di ottimizzazione combinatoria, apportandovi specifici adattamenti ed eventuali modifiche in sede di realizzazione, il *branch-and-bound* rimane tuttora uno dei metodi più soddisfacenti per ottenere una soluzione *ottima*, magari abbinandolo a diverse tecniche di programmazione lineare e di *ricerca locale* (allo scopo di migliorare localmente una soluzione trovata con una regola euristica: sebbene quasi mai suffragate dalla teoria, tali tecniche hanno dimostrato una grande efficacia).

In generale, non è detto che un approccio euristico offra di per sé garanzie; per certi problemi può però accadere che uno specifico metodo euristico fornisca molto probabilmente un risultato vicino all’ottimo. Nella fattispecie, ad esempio, si può anche pensare a ottimi locali, combinati in un unico ciclo mediante un algoritmo *greedy* simile a quelli per trovare un albero ricoprente di costo minimo, di cui parleremo più avanti: se non altro, si spera di ottenere in tal modo una buona maggiorazione del costo ottimo.

Un algoritmo di *branch-and-bound* fu applicato per la prima volta al TSP da J. D. C. Little e altri, nel 1963. Senza scendere nei dettagli, riteniamo però utile esporre l’idea di fondo e illustrare un piccolissimo esempio.

Immaginando di disporre inizialmente dell’insieme di tutte le possibili soluzioni (rappresentato in modo implicito) e di voler individuarne una che *minimizzi* una data funzione di costo, com’è nel nostro caso, si procede per passi successivi. Ogni passo considera un insieme di soluzioni e consiste di due fasi: si ripartisce l’insieme in due o più sottoinsiemi non vuoti (*branching*); quindi, per ciascuno dei sottoinsiemi, si calcola un limite inferiore (*lower bound*), o meglio una *minorazione*, che sia sicuramente minore o uguale al minimo dei costi associati alle soluzioni del sottoinsieme considerato.

Si iterà il procedimento, di solito considerando uno dei sottoinsiemi con *lower bound* più basso (sì da avere buona probabilità che contenga una soluzione ottima), fino a giungere – secondo la via più breve – a un sottoinsieme costituito da una sola soluzione: sarà ottima? Ne avremmo la certezza soltanto se a *tutti* i sottoinsiemi sui quali dovremmo ancora iterare il procedimento fosse stato attribuito un *lower bound* maggiore o uguale al costo della soluzione individuata.

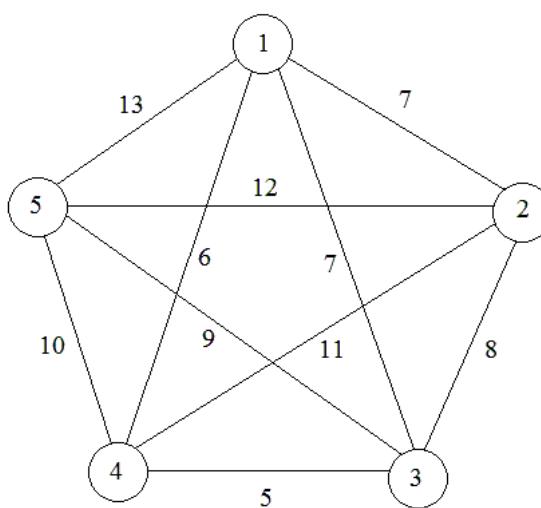
Se così non è, basterà riprendere uno dei sottoinsiemi con *lower bound* minore, e andare avanti nel modo descritto, fino a che tale condizione non sarà soddisfatta.

Si noti comunque che il costo della migliore soluzione finora individuata può essere usato come maggiorazione (*upper bound*) del costo ottimo: ciò consente subito di escludere da successive esplorazioni i sottoinsiemi con *lower bound* maggiore o uguale ad esso.

In genere, con la tecnica di *branch-and-bound* si riesce a realizzare un notevole risparmio sul numero di soluzioni esplorate; molto dipende dalla bontà dei criteri di *branching* e, soprattutto, di *bounding*, legati al particolare problema. Si tenga tuttavia presente che, come sempre in questa classe di problemi, non si riesce a migliorare l'ordine di grandezza dell'enumerazione completa nei casi peggiori.

Come può essere adattata quest'idea al TSP? Se consideriamo la matrice dei costi, anche nel caso asimmetrico, notiamo che a una soluzione concorrerà comunque uno e un solo costo per ogni riga e uno e un solo costo per ogni colonna (ricordiamoci di escludere sempre dai nostri discorsi la diagonale principale, e teniamo presente che detta condizione è necessaria ma non sufficiente a determinare un ciclo completo). Pertanto, se diminuiamo di una costante tutti i costi su una certa riga o su una certa colonna, diminuirà della stessa quantità il costo di ciascuna soluzione... ma l'ottima resterà ottima!

Per operare una riduzione dei costi su una riga (o colonna) scegliamo come costante il minore dei costi su quella riga (o colonna), sicché almeno uno scenderà a zero.



Prendiamo ad esempio la matrice dei costi di questo grafo, non orientato e completo, con cinque nodi:

$$\begin{array}{cccccc}
 - & 7 & 7 & 6 & 13 \\
 7 & - & 8 & 11 & 12 \\
 7 & 8 & - & 5 & 9 \\
 6 & 11 & 5 & - & 10 \\
 13 & 12 & 9 & 10 & -
 \end{array}$$

Possiamo ridurla al massimo togliendo 6 dalla prima riga, 7 dalla seconda, 5 dalla terza e dalla quarta, 9 dalla quinta, e poi togliendo ancora 1 dalla seconda colonna e 4 dalla quinta; otteniamo:

$$\begin{array}{ccccc}
 - & 0 & 1 & 0 & 3 \\
 0 & - & 1 & 4 & 1 \\
 2 & 2 & - & 0 & 0 \\
 1 & 5 & 0 & - & 1 \\
 4 & 2 & 0 & 1 & -
 \end{array}$$

dove abbiamo almeno uno 0 su ciascuna riga e su ciascuna colonna. La riduzione operata ammonta a 37, e poiché i costi rimasti in matrice sono ovviamente non negativi, il valore 37 sicuramente costituisce un *lower bound* per l'insieme iniziale, che comprende tutte le soluzioni ammissibili. In altre parole: 37 è il minimo costo che ci possiamo aspettare, qualora trovassimo un ciclo costituito da soli archi corrispondenti a 0 nella matrice ridotta, ma in realtà potrebbe andar peggio. (Qui si vede subito che non sarà possibile trovare un tale ciclo, poiché le ultime due righe hanno un solo zero nella stessa colonna.)

Per la fase di *branching*, prendiamo per l'appunto in considerazione gli 0 nella matrice ridotta: associamo ad ogni 0 la somma del minimo sulla sua riga col minimo sulla sua colonna, e poi consideriamo quello o uno di quelli ai quali è stato associato il valore più alto. Nel nostro esempio, numerando righe e colonne da 1, come i nodi del grafo, sono quelli di posto (1, 2) e (2, 1), ai quali è associato il valore  $h = 2$ .

Scegliamo il primo, e in base ad esso suddividiamo l'insieme delle soluzioni in due sottoinsiemi disgiunti. Perché abbiamo fatto così? Se un certo ciclo *non* contiene l'arco (1, 2), resta pur vero che il nodo 1 dev'essere raggiunto da qualche altro nodo, per cui al costo di tale ciclo concorrerà, nella migliore delle ipotesi, il minore della prima riga, escluso quello in (1, 2); analogamente, poiché dal nodo 2 si deve raggiungere un qualche altro nodo, al costo del ciclo concorrerà, nella migliore delle ipotesi, il minore della colonna 2, escluso quello in (1, 2). La somma di questi due costi rappresenta, in un certo senso, il “danno” subito da un ciclo a causa della mancanza dell'arco (1, 2). Siccome questo è uno degli archi la cui mancanza causa il massimo danno, si spera che una soluzione ottima si trovi nel sottoinsieme delle soluzioni che lo contengono.

Le soluzioni che *non* comprendono l'arco (1, 2) avranno quindi un *lower bound* di  $37 + h = 39$ . Quelle che invece comprendono l'arco da 1 a 2, certamente non comprendono quello da 2 a 1 (altrimenti vi sarebbe un sotto-ciclo), per cui possiamo depennare il costo in (2, 1), oltre a tutta la prima riga e a tutta la seconda colonna:

-	-	-	-	-
-	-	1	4	1
2	-	-	0	0
1	-	0	-	1
4	-	0	1	-

Su questa matrice possiamo operare due riduzioni di costo 1 (sulla seconda riga e sulla prima colonna) per un totale di 2; quindi anche al sottoinsieme delle soluzioni che comprendono l'arco da 1 a 2 può essere associato un *lower bound* dato da  $37 + 2 = 39$ .

Abbiamo completato la fase di *bounding*, e abbiamo due sottoinsiemi con stesso *lower bound*: ciò non deve stupirci, perché l'istanza del problema che stiamo risolvendo è simmetrica, dunque anche nell'altro sottoinsieme vi sarà una soluzione di costo minimo che comprenderà l'arco da 2 a 1.

Continuiamo, sviluppando il sottoinsieme delle soluzioni che contengono l'arco  $(1, 2)$ . Operate le riduzioni, otteniamo la matrice:

-	-	-	-	-
-	-	0	3	0
1	-	-	0	0
0	-	0	-	1
3	-	0	1	-

Ai tre 0 di posto  $(3, 4)$ ,  $(4, 1)$  e  $(5, 3)$  è associato il valore  $h = 1$  (agli altri il valore 0), sicché scegliamo ad esempio l'arco  $(3, 4)$  per il nuovo *branching*. Le soluzioni che non comprendono l'arco da 3 a 4 avranno un *lower bound* pari a  $39 + h = 40$ . Quelle che invece comprendono l'arco da 3 a 4, certamente non comprendono quello da 4 a 3, per cui possiamo depennare il costo in  $(4, 3)$ , oltre a tutta la terza riga e a tutta la quarta colonna:

-	-	-	-	-
-	-	0	-	0
-	-	-	-	-
0	-	-	-	1
3	-	0	-	-

Nessuna riduzione può essere fatta su questa matrice, per cui al sottoinsieme delle soluzioni che comprendono l'arco da 3 a 4 rimane associato il *lower bound* 39.

Quindi continuiamo sviluppando questo sottoinsieme. Lo 0 al quale è associata la somma più alta ( $h = 1 + 3 = 4$ ) corrisponde all'arco  $(4, 1)$ ; alle soluzioni che non lo comprendono è dunque assegnato un *lower bound* pari a  $39 + 4 = 43$ , mentre quelle che lo comprendono certamente non avranno nemmeno l'arco da 2 a 3: infatti contengono già gli archi  $(1, 2)$ ,  $(3, 4)$  e  $(4, 1)$ , per cui l'arco  $(2, 3)$  chiuderebbe un ciclo non hamiltoniano. Allora depenniamo il costo in  $(2, 3)$ , oltre a quelli che sono

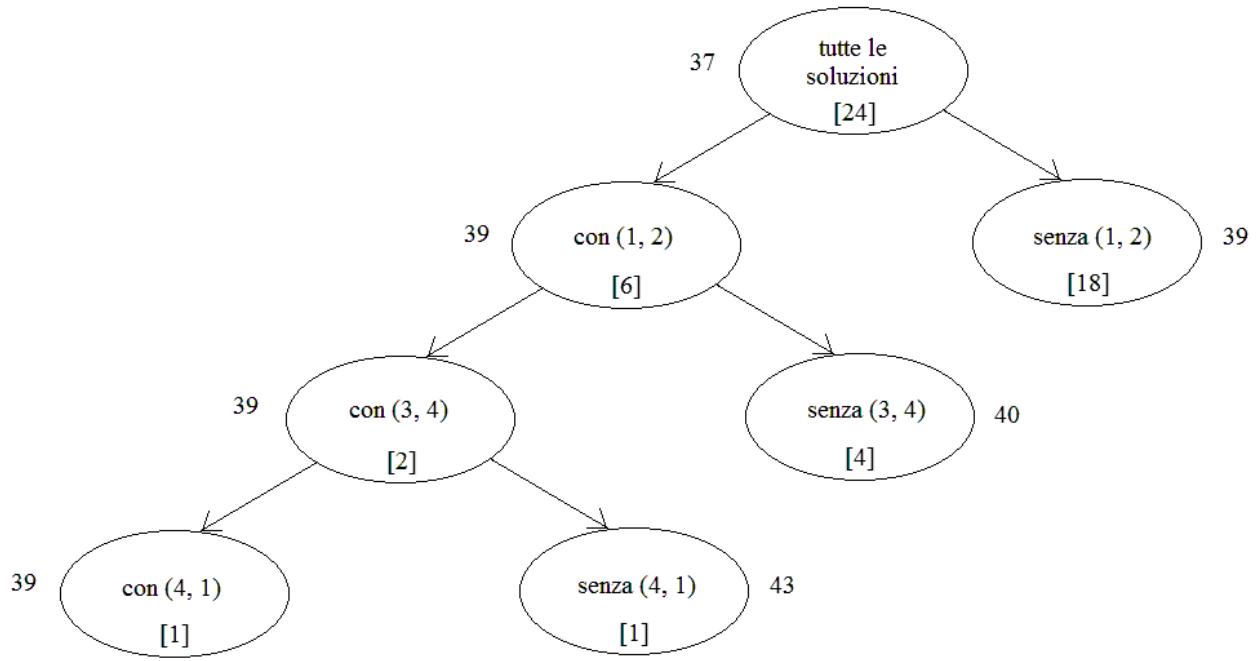
rimasti nella quarta riga e nella prima colonna. Siamo giunti a una matrice che, di fatto, può essere riguardata come  $2 \times 2$ :

$$\begin{array}{ccccc} - & - & - & - & - \\ - & - & - & - & 0 \\ - & - & - & - & - \\ - & - & - & - & - \\ - & - & 0 & - & - \end{array}$$

nella quale non è più possibile alcuna riduzione; sicché il sottoinsieme delle soluzioni che hanno l'arco  $(4, 1)$  non soltanto mantiene il *lower bound* 39, ma è costituito dalla sola soluzione che comprende anche gli archi  $(2, 5)$  e  $(5, 3)$ .

In effetti, questa soluzione costituita dai cinque archi  $(1, 2)$ ,  $(3, 4)$ ,  $(4, 1)$ ,  $(2, 5)$  e  $(5, 3)$ , che determina quindi il ciclo  $[1, 2, 5, 3, 4, 1]$ , ha costo 39, che è il minimo possibile. Di ciò possiamo essere sicuri, perché tutti i sottoinsiemi non esplorati hanno minorazioni maggiori o uguali a 39.

I risultati di questo esempio, breve e fortunato, sono riassunti nell'*albero* di figura.



Gli insiemi di soluzioni rappresentati dalle *foglie* costituiscono una *partizione* dell'insieme di tutte le soluzioni, rappresentato dalla *radice*, da cui il calcolo è iniziato: e ciò è vero ad ogni passo del processo. Siamo stati fortunati perché abbiamo percorso un *ramo* che ci ha portati a una foglia che rappresenta una soluzione *ottima*, poiché il suo costo è minore o uguale al *lower bound* di tutte le altre foglie.

Per inciso, il *lower bound* 43 compete alla soluzione  $[1, 2, 3, 4, 5, 1]$ , mentre 40 è il costo del ciclo  $[1, 2, 3, 5, 4, 1]$  e, infine, ha ovviamente costo minimo 39 anche il ciclo simmetrico  $[1, 4, 3, 5, 2, 1]$ , che in effetti non contiene l'arco  $(1, 2)$  ma l'arco  $(2, 1)$ .

In generale, qualora presenti un costo superiore al *lower bound* di qualche altra foglia, la soluzione trovata può servire per aggiornare l'*upper bound* (se questo è superiore); si deve però continuare il processo, sviluppando l'insieme rappresentato da una delle foglie con *lower bound* minore.

Con questo esempio banale – attraverso il quale non si arriva di certo a evincere come trattare le particolari casistiche, né come risolvere al meglio i dettagli implementativi – abbiamo voluto soltanto accennare a un’idea che ha avuto grande successo ed è stata nel corso degli anni perfezionata e combinata con altre tecniche: ad esempio col metodo dei *piani di taglio*, già impiegato da Dantzig nella programmazione lineare intera, dando luogo al cosiddetto *branch-and-cut*.

I “tagli” servono per escludere parti dello spazio di ricerca del problema rilassato, senza rischiare di eliminare soluzioni intere ammissibili: in generale, sono generati in ogni “nodo decisionale”, con la speranza di ottenere una soluzione intera o un *lower bound* più elevato, e se ciò non avviene si passa alla fase di *branching*.

Più recentemente, con l’ulteriore ausilio di buone euristiche e implementazioni assai sofisticate, il metodo del *branch-and-cut* è stato usato per risolvere esattamente istanze del TSP con un numero di nodi dell’ordine delle decine di migliaia, in netto contrasto coi tempi previsti dai casi peggiori (si vedano i resoconti degli ultimi successi all’indirizzo <http://www.math.uwaterloo.ca/tsp/>).

## Un algoritmo di programmazione dinamica.

Nel 1962, applicando il principio della *programmazione dinamica*, M. Held e R. M. Karp realizzarono l’algoritmo (esponenziale) esatto per il TSP (anche asimmetrico) che vanta tuttora il miglior tempo di esecuzione nel caso peggiore.

Vediamo come funziona. Fissiamo come al solito il nodo di partenza: sia 1. A partire da 1, dovremo visitare tutti gli altri nodi, termineremo quindi in un certo nodo  $k$ , e infine da quest’ultimo dovremo ritornare in 1. Se sapessimo quali sono i costi *minimi* per andare da 1 a ciascuno degli altri nodi  $k = 2, \dots, n$ , potremmo aggiungere ad ognuno il costo del rispettivo arco  $(k, 1)$  e sapere così, scegliendo il minore, qual è il costo minimo del ciclo completo. Allo stesso modo, per sapere qual è il costo minimo da 1 a  $k$ , bisognerà calcolare i costi minimi da 1 a qualsiasi altro nodo che non sia  $k$  (e neanche 1, ovviamente) e poi sommarvi rispettivamente i costi degli archi diretti dall’ultimo nodo raggiunto al nodo  $k$ . Così di seguito, arriveremo a dover conoscere i costi minimi da 1 a qualsiasi altro nodo, seguendo un cammino che *non ne comprenda altri*: ma allora questi saranno semplicemente i costi degli archi che collegano direttamente 1 a ciascuno degli altri nodi!

Indichiamo con  $opt(S, k)$ , con  $k$  appartenente a  $S$ , il costo minimo di un cammino da 1 a  $k$  che tocchi esclusivamente nodi nell’insieme  $S$  (escluso il nodo di partenza 1, che non appartiene a  $S$ ). L’idea è dunque quella di calcolare tutti questi costi, per ogni  $S$  contenuto o uguale all’insieme dei nodi da 2 a  $n$ , e per ogni  $k$  appartenente a  $S$ , cominciando da quelli già noti, e cioè quelli per cui  $S$  è costituito da un solo nodo. Applichiamola a un esempio semplicissimo, addirittura un grafo completo di soli

quattro nodi, ma per generalità con una matrice dei costi non simmetrica:

$$\begin{array}{cccc} - & 3 & 4 & 5 \\ 1 & - & 2 & 6 \\ 3 & 3 & - & 7 \\ 2 & 4 & 6 & - \end{array}$$

$$opt(\{2\}, 2) = C(1, 2) = 3$$

$$opt(\{3\}, 3) = C(1, 3) = 4$$

$$opt(\{4\}, 4) = C(1, 4) = 5$$

$$opt(\{2, 3\}, 2) = opt(\{3\}, 3) + C(3, 2) = 4 + 3 = 7$$

$$opt(\{2, 3\}, 3) = opt(\{2\}, 2) + C(2, 3) = 3 + 2 = 5$$

$$opt(\{2, 4\}, 2) = opt(\{4\}, 4) + C(4, 2) = 5 + 4 = 9$$

$$opt(\{2, 4\}, 4) = opt(\{2\}, 2) + C(2, 4) = 3 + 6 = 9$$

$$opt(\{3, 4\}, 3) = opt(\{4\}, 4) + C(4, 3) = 5 + 6 = 11$$

$$opt(\{3, 4\}, 4) = opt(\{3\}, 3) + C(3, 4) = 4 + 7 = 11$$

$$\begin{aligned} opt(\{2, 3, 4\}, 2) &= \min \{ opt(\{3, 4\}, 3) + C(3, 2), opt(\{3, 4\}, 4) + C(4, 2) \} \\ &= \min \{ 11 + 3, 11 + 4 \} = 14 \end{aligned}$$

$$\begin{aligned} opt(\{2, 3, 4\}, 3) &= \min \{ opt(\{2, 4\}, 2) + C(2, 3), opt(\{2, 4\}, 4) + C(4, 3) \} \\ &= \min \{ 9 + 2, 9 + 6 \} = 11 \end{aligned}$$

$$\begin{aligned} opt(\{2, 3, 4\}, 4) &= \min \{ opt(\{2, 3\}, 2) + C(2, 4), opt(\{2, 3\}, 3) + C(3, 4) \} \\ &= \min \{ 7 + 6, 5 + 7 \} = 12 \end{aligned}$$

e infine:

$$\begin{aligned} opt(\{1, 2, 3, 4\}, 1) &= \min \{ opt(\{2, 3, 4\}, 2) + C(2, 1), \\ &\quad opt(\{2, 3, 4\}, 3) + C(3, 1), \\ &\quad opt(\{2, 3, 4\}, 4) + C(4, 1) \\ &\quad \} \\ &= \min \{ 14 + 1, 11 + 3, 12 + 2 \} = 14 \end{aligned}$$

Il costo minimo di un ciclo completo è dunque 14. Procedendo a ritroso, si determinano gli archi che lo compongono. Qui, poiché al costo 14 si arriva sia da  $11 + 3$  sia da  $12 + 2$ , la soluzione non è unica. Infatti:

$$\begin{aligned} 11 + 3 &= opt(\{2, 3, 4\}, 3) + C(3, 1) \\ &= opt(\{2, 4\}, 2) + C(2, 3) + C(3, 1) \\ &= opt(\{4\}, 4) + C(4, 2) + C(2, 3) + C(3, 1) \\ &= C(1, 4) + C(4, 2) + C(2, 3) + C(3, 1) \end{aligned}$$

$$\begin{aligned} 12 + 2 &= opt(\{2, 3, 4\}, 4) + C(4, 1) \\ &= opt(\{2, 3\}, 3) + C(3, 4) + C(4, 1) \\ &= opt(\{2\}, 2) + C(2, 3) + C(3, 4) + C(4, 1) \\ &= C(1, 2) + C(2, 3) + C(3, 4) + C(4, 1) \end{aligned}$$

per cui si hanno i due cicli ottimi  $[1, 4, 2, 3, 1]$  e  $[1, 2, 3, 4, 1]$  (e ovviamente *non* i loro “rovesci”, perché qui il problema è asimmetrico).

Potete provare questo metodo sul precedente esempio di grafo con cinque nodi; se procederete a mano, vi accorgerete che la mole di calcoli da eseguire aumenterà sensibilmente. Tuttavia, su grafi completi di  $n$  nodi, l’algoritmo di Held e Karp ha una complessità temporale dell’ordine di  $n^2 2^n$ , che, già per  $n$  dell’ordine della decina, è decisamente migliore di  $n!$  (questo non è un punto esclamativo, ma denota il solito fattoriale di  $n$ ). Purtroppo, come non è difficile intuire, anche lo *spazio* usato da questo algoritmo per mantenere le tracce cresce esponenzialmente con  $n$ , laddove un algoritmo che esaminasse tutte le permutazioni userebbe uno spazio proporzionale a  $n!$ .

Nel 1970 Held e Karp proposero pure un metodo iterativo che, partendo dal problema rilassato, riesce a stimare un buon *lower bound* per il valore ottimo del TSP.

## Il TSP METRICO e gli alberi ricoprenti di costo minimo.

Tra quelli “difficili”, il TSP è forse il problema di ottimizzazione combinatoria che è stato meglio studiato, attraverso l’applicazione di varie tecniche alle quali è servito come banco di prova; ad alcune di queste abbiamo appena accennato nei paragrafi precedenti. Non soltanto migliaia di articoli scientifici sono stati dedicati a tutte le sue numerose varianti, ma pure interi libri, fino agli anni più recenti: e anche la complessità formale di questi testi, riservati agli specialisti, è cresciuta nel tempo!

Un caso particolare del problema è costituito dal cosiddetto TSP METRICO. Supponiamo cioè che il grafo, di  $n$  nodi, sia completo e che i costi, positivi e simmetrici, soddisfino la *disuguaglianza triangolare*: per ogni terna di nodi  $i, j, k$ ,  $C(i, j) \leq C(i, k) + C(k, j)$ .

In questa ipotesi il problema rimane ugualmente “difficile”, tuttavia si può avere una certa garanzia da alcuni algoritmi di approssimazione.

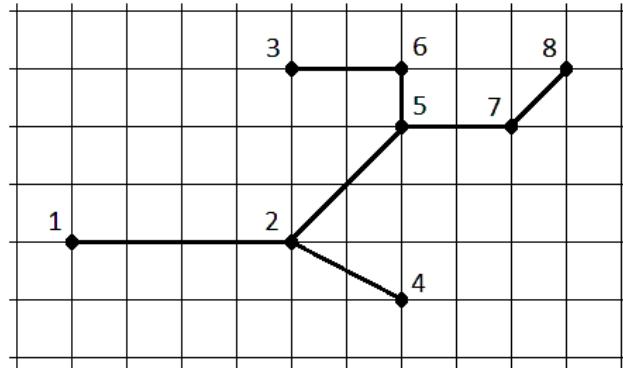
L’euristica *nearest neighbour*, che sceglie ad ogni passo il nodo più vicino tra quelli non ancora visitati, trova sempre un ciclo (poiché il grafo è completo) ma non si traduce in un algoritmo di approssimazione con fattore costante.

Apriamo una parentesi, per ricordare la nozione di *albero ricoprente* (o *di supporto*) *di costo minimo* (*minimum spanning tree*, MST), peraltro coinvolta nel quesito “Gli elettricisti” della gara finale *Kangourou dell’Informatica* di maggio 2009.

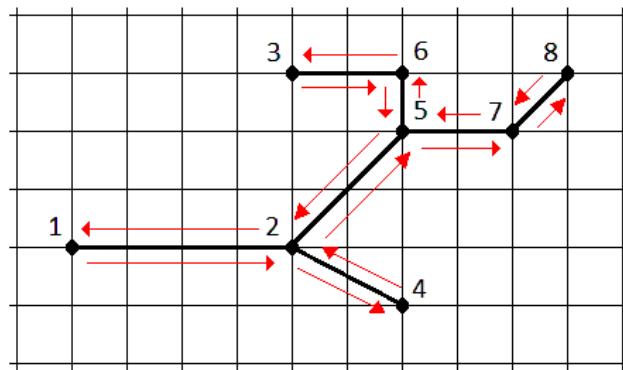
Un albero ricoprente è un sottografo che comprende tutti i nodi e ha la proprietà di essere un albero non orientato. Dire che è un albero non orientato equivale a dire che è un grafo *connesso minimale* (togliendo uno qualsiasi dei suoi archi, non è più vero che da ciascun nodo si può raggiungere ogni altro nodo) ed equivale anche a dire che è un grafo *aciclico massimale* (aggiungendovi un arco si crea inevitabilmente un ciclo). Ogni albero ricoprente ha  $n - 1$  archi (non orientati).

Supponiamo allora di avere un albero ricoprente di costo minimo, cioè tale che la somma dei costi degli archi che lo compongono sia la più piccola possibile; diremo dopo come calcolarlo.

Nella figura sottostante vediamo un esempio nel piano, con 8 nodi: assumendo che il costo di un arco sia la distanza euclidea tra i due nodi che unisce (condizione sufficiente affinché le disuguaglianze triangolari siano tutte soddisfatte), abbiamo tracciato soltanto i 7 archi che costituiscono l'MST (unico, in questo esempio).

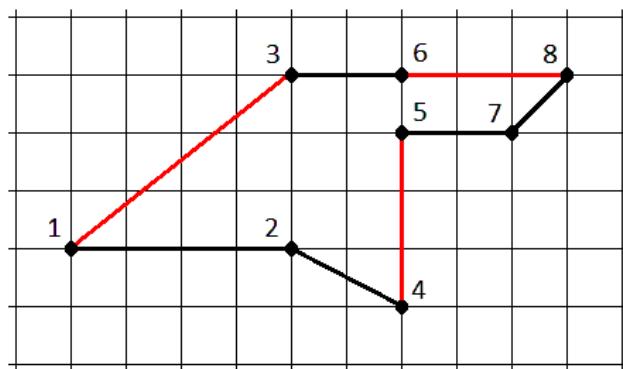


Partiamo da un nodo arbitrario, e giriamo attorno a questo albero, tenendo sempre la destra, come indicato nella successiva figura, fino a ritornare al nodo di partenza.



(In alternativa, avremmo potuto scegliere di tenere sempre la sinistra.) Se partiamo dal nodo 1, il nostro percorso sarà [1, 2, 4, 2, 5, 7, 8, 7, 5, 6, 3, 6, 5, 2, 1], chiaramente lungo il doppio dell'MST.

Da questo percorso possiamo ricavare un ciclo, certamente non più lungo in virtù delle disuguaglianze triangolari, semplicemente saltando nella lista i nodi già visitati e concludendo su 1: [1, 2, 4, 5, 7, 8, 6, 3, 1].



Sarà un ciclo ottimo? Provate a controllare! In generale, se lo fosse, si tratterebbe davvero di un caso *molto* fortunato! Tuttavia possiamo affermare che il suo costo non sarà di certo superiore a due volte il costo dell'MST; ma a sua volta il costo dell'MST è inferiore al costo del ciclo ottimo: infatti, se da un ciclo ottimo togliamo un arco otteniamo un albero ricoprente!

In conclusione: il ciclo che abbiamo ricavato ha un costo sicuramente minore del doppio di quello del ciclo ottimo. Questo procedimento, noto come “algoritmo a doppio albero”, costituisce dunque un algoritmo di approssimazione con fattore 2 per il TSP METRICO: alla peggio, si pagherà il 100% in più! La parte preponderante del tempo di esecuzione è occupata dal calcolo di un MST, compito risolvibile in modo efficiente.

In effetti, il problema di come trovare un MST, semplice ma di fondamentale importanza, ha una storia piuttosto lunga. Sono conosciuti alcuni algoritmi la cui esecuzione richiede un tempo di ordine variabile a seconda delle strutture di dati specifiche che vi si impiegano, ma comunque limitato da un polinomio:  $m \cdot n$ ,  $m \cdot \log_2 n$ ,  $n^2$ ,  $m + n \cdot \log_2 n$ , dove  $m$  è il numero degli archi, o addirittura soltanto  $n \cdot \log_2 n$ , per un insieme di  $n$  punti del piano come nell'esempio che abbiamo fatto.

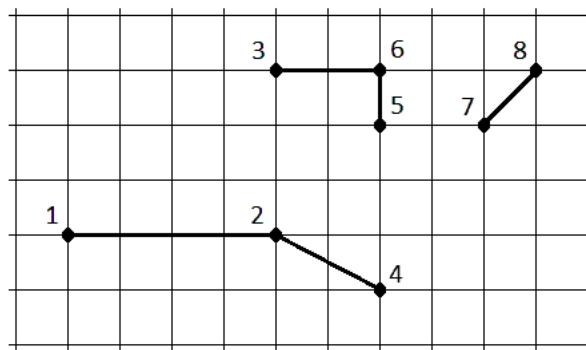
Di due di essi parlammo dando le soluzioni alla suddetta gara: l'*algoritmo di Kruskal* (Joseph B. Kruskal, 1956) e l'*algoritmo di Prim* (che ispirò Dijkstra; Robert C. Prim, 1957, ma già pubblicato da Vojtěch Jarník, in un articolo in lingua ceca, nel 1930). Il più antico, e forse anche il più facile da realizzare, è tuttavia l'*algoritmo di Borůvka*, progettato da Otakar Borůvka nel 1926 per la costruzione di una rete elettrica in Moravia. L'idea è la seguente: all'inizio si mettono nell'albero T soltanto gli  $n$  nodi del grafo (senza alcun arco), che costituiscono altrettante *componenti connesse*, cioè “parti separate” dell'albero; dopodiché:

**finché** T ha meno di  $n - 1$  archi:

**per ogni** componente连通的 K di T:

- (\*) sia  $(i, j)$  l'arco di minor costo, con  $i$  nodo in K e  $j$  nodo fuori da K;  
si aggiunge a T l'arco  $(i, j)$ , a meno che non vi sia già

Al termine, T sarà un MST. Si osservi che ad ogni iterazione del ciclo esterno possono essere aggiunti più archi. Ad esempio, considerando il grafo precedente, all'inizio in T avremo soltanto gli otto nodi; ma già dopo la prima iterazione gli archi saranno cinque:



E, dopo la seconda iterazione, queste tre componenti connesse risulteranno collegate dai restanti due archi (2, 5) e (5, 7).

Nel trattare questo esempio è andato tutto bene; però, per avere la garanzia che l'algoritmo funzioni sempre correttamente, sarebbe sufficiente non trovarsi mai davanti a due o più alternative al momento della scelta (\*); dovrebbe essere chiaro il motivo di tale requisito, non richiesto dall'algoritmo di Kruskal che pure costruisce l'albero procedendo, in generale, da componenti separate. Basta dunque un controllo preliminare sui costi degli archi, eventualmente variandone di pochissimo alcuni per evitare che due archi abbiano uguale costo, senza inficiare la validità del risultato finale.

Per concludere il nostro discorso, ricordiamo che per il TSP METRICO è stato trovato un algoritmo di approssimazione con fattore  $3/2$  (certo), la cui esecuzione richiede un tempo di ordine  $n^3$  (N. Christofides, 1976); non si sa se ve ne siano di migliori (non probabilistici). Nel caso più generale di costi non simmetrici, c'è un algoritmo di approssimazione con fattore (certo)  $2/3 \cdot \log_2 n$  (U. Feige e M. Singh, 2007).

Un caso speciale del TSP METRICO che è stato studiato a fondo è il TSP EUCLIDEO: i nodi sono punti del piano e i costi sono le lunghezze dei segmenti che li collegano, come nel particolare esempio che abbiamo fatto in questo paragrafo; in tal caso, il ciclo ottimo è sempre un poligono (in generale non convesso). Per il TSP EUCLIDEO è stato ideato un algoritmo esatto con tempo di esecuzione sub-esponenziale (R. Z. Hwang e altri, 1993).

Sul grafo con 5 nodi che ci è servito per illustrare il metodo *branch-and-bound*, il TSP è metrico, ma non euclideo: non si riescono infatti a disporre i nodi sul piano in modo da rispettare le distanze reciproche. Provate a verificare questa affermazione e ad applicare a tale grafo sia l'euristica *nearest neighbour*, a partire da diversi nodi, sia l'algoritmo a doppio albero, dopo aver trovato l'MST.

## Il problema dell'imballaggio.

Così come il problema di ottimizzazione del commesso viaggiatore, anche quello dell'imballaggio ha un dato di input in meno del corrispondente decisionale: il numero di contenitori (tutti di stessa capacità), che naturalmente deve essere *minimizzato*. Il “costo” di una soluzione è dato dal numero di contenitori usati.

Pertanto, un'istanza del problema decisionale è immediatamente riconducibile a un problema di ottimizzazione: se, risolvendo quest'ultimo, si ottiene un risultato minore o uguale al numero di contenitori disponibili, la risposta al problema decisionale sarà “sì”, altrimenti sarà “no”.

Si può dimostrare che pure la riduzione inversa è fattibile in tempo polinomiale: vediamo come, partendo dal fatto che la soluzione ottima certamente non prevede più di  $n$  contenitori, essendo  $n$  il numero di oggetti da imballare. (Il caso in cui almeno uno dei pesi superi la capacità di un contenitore può essere escluso in tempo lineare.) Si risolve il problema decisionale con  $k$  (= numero di contenitori disponibili) =  $n/2$ ; se la risposta è “sì”, si prova nuovamente con  $k = n/4$ , altrimenti si

prova con  $k = 3n/4$ . Si prosegue dimezzando ad ogni passo l'intervallo di incertezza sul numero dei contenitori necessari, proprio come nella ricerca binaria, fino a trovarlo. Questo procedimento richiede di eseguire l'algoritmo che risolve il problema decisionale un numero di volte  $m \leq \text{floor}(\log_2 n)$ . Il valore di  $n$  è parte dell'input (che, per il problema di ottimizzazione, deve specificare  $n + 1$  interi), e quindi  $m$  è proporzionale (al più) alla dimensione dell'input. Si conclude che il problema di ottimizzazione non è “più difficile” di quello decisionale: se l'algoritmo che risolve il problema decisionale avesse complessità polinomiale, allora la avrebbe anche il problema di ottimizzazione.

Il problema di ottimizzazione dell'imballaggio è di grande rilevanza pratica. Si presenta spesso, nella realtà, in una molteplicità di forme: tagliare una serie di tubi di varie lunghezze da un numero minimo di tubi di lunghezza standard; registrare una serie di brani musicali di diversa durata (o di *file* di diversa lunghezza) sul minimo numero di dischi (o di altri dispositivi di memoria) di stessa capacità... Un'istanza di questo tipo si trova nel libretto della gara *Kangourou dell'Informatica* di marzo 2009: nel quesito proposto, “Archivio multimediale”, si doveva appunto usare il minor numero di chiavette USB di una stessa capacità prefissata.

Più euristiche *greedy* sono di spontanea formulazione per problemi di questo genere, considerando inizialmente un solo contenitore vuoto:

- ◆ *next fit*: ciascun oggetto è inserito nell'ultimo contenitore considerato, purché la capacità di questo non venga superata, altrimenti è messo in un nuovo contenitore;
- ◆ *first fit*: ciascun oggetto è inserito nel primo contenitore con capacità residua sufficiente tra quelli già usati, ovvero se non ci sono contenitori adatti ne usa uno nuovo;
- ◆ *best fit*: è scelto il contenitore la cui capacità residua sia la minima sufficiente a collocare l'oggetto correntemente considerato (si cerca così di mantenere disponibili le capacità residue più grandi, sebbene si vada a formare nel contenitore scelto una capacità residua che potrebbe non essere più sfruttabile a causa della sua piccolezza);
- ◆ *next-fit decreasing*, *first-fit decreasing* e *best-fit decreasing*: come i precedenti, ma prendendo gli oggetti in ordine decrescente di peso.

Quale, tra queste, sarà la strategia più conveniente? Darà una qualche garanzia sulla bontà della soluzione?

Non è un evento raro che possano capitare delle anomalie, in apparente contraddizione col senso comune. Consideriamo ad esempio i seguenti 33 pesi, che complessivamente ammontano a 3668 unità di misura:

1 peso di 442	7 pesi di 252	5 pesi di 127	4 pesi di 106
1 peso di 85	1 peso di 84	1 peso di 46	2 pesi di 37
3 pesi di 12	6 pesi di 10	2 pesi di 9	

e contenitori di capacità 524 unità di misura (Ronald L. Graham, 1978).

È evidente che occorrono almeno  $3668 / 524 = 7$  contenitori (qui il quoziente è esatto, e dunque non v'è bisogno di approssimare per eccesso). Un algoritmo di esaurimento esatto dovrebbe enumerare un bel po' di partizioni del nostro insieme di oggetti...

L'algoritmo *first-fit decreasing* riempie esattamente 7 contenitori, e quindi dà in questo caso il risultato ottimo. Eliminando il peso 46, la stessa procedura impiega non 7 bensì 8 contenitori, all'ultimo dei quali è assegnato soltanto un peso 9 (e in ciascuno degli altri rimane una capacità inutilizzata pari a 7 o a 8 unità di misura)!

L'analisi dettagliata dei suddetti algoritmi ha portato ai seguenti risultati:

- ◆ *next fit*: richiede un tempo lineare; fornisce un risultato  $\leq 2m - 1$ , dove  $m$  è la soluzione ottima, cioè il minimo numero di contenitori della capacità assegnata che possono ospitare gli oggetti del problema;
- ◆ *first fit*: può essere realizzato in modo da richiedere un tempo dell'ordine di  $n \cdot \log_2 n$ ; è ovvio che non possa essere peggiore del precedente, e infatti fornisce un risultato  $\leq \text{ceiling}(17 \cdot m / 10)$  (D. S. Johnson e altri, 1974; M. R. Garey e altri, 1976), recentemente “ritoccato” in  $\leq \text{floor}(17 \cdot m / 10)$  (G. Dósa e J. Sgall, 2013; nel 2014 essi hanno provato lo stesso risultato per il *best fit*): in pratica, nei casi peggiori, si userà circa il 70% in più dei contenitori necessari;
- ◆ *first-fit decreasing*: anche questo può essere realizzato come il precedente (l'ordinamento efficiente richiede di nuovo un tempo dell'ordine di  $n \cdot \log_2 n$ ); è un algoritmo di approssimazione con fattore 3/2 (D. Simchi-Levi, 1994), fattore che corrisponde al 50% in più ed è il minore possibile a meno che P = NP, ma la garanzia di prestazione *asintotica* (vale a dire, nella pratica, quando l'ottimo  $m$  è in realtà sufficientemente grande) è ancora migliore, poiché in effetti l'algoritmo fornisce un risultato  $\leq 11 \cdot m / 9 + 2/3$  (G. Dósa, 2007; D. S. Johnson, nella sua tesi di dottorato al MIT, nel 1973, aveva provato il limite  $11 \cdot m / 9 + 4$ ).

Non ci sono sostanziali differenze tra *first-fit decreasing* e *best-fit decreasing* (il fattore di garanzia asintotica di 11/9 è stretto anche per quest'ultimo): il costo della soluzione trovata dal primo può essere inferiore a quello risultante dall'applicazione del secondo (ad esempio, si provino le diverse strategie sull'istanza con 7 oggetti, di pesi 20, 11, 11, 4, 3, 3, 2, e contenitori di capacità 27: qui è immediato capire quale sia la soluzione ottima); ci sono dei casi, però, dove soltanto *best-fit decreasing* trova una soluzione ottima.

Per il problema dell'imballaggio esistono molti altri algoritmi di approssimazione, anche con un fattore di garanzia asintotica inferiore a 11/9. Anzi, esistono algoritmi (tempo-polynomiali) con fattori arbitrariamente buoni (cioè “pochissimo” maggiori di 1). Noi ci fermiamo qui, invitando – come sempre – il lettore a fare una sua ricerca, anche sui modi di risolvere il problema esattamente, cioè all'ottimo, dei quali non abbiamo parlato: ad esempio, un'idea è quella di reimpiegare il metodo *branch-and-bound*. Da tener presente che, se si fissa (con una *costante*) il numero delle differenti dimensioni degli oggetti, allora il problema è risolvibile esattamente in tempo-polonomiale nel numero di oggetti (M. X. Goemans e T. Rothvoß, 2014).

Sono state studiate diverse generalizzazioni del problema dell’imballaggio, ad esempio in due dimensioni, dove è richiesto di sistemare un insieme di rettangoli con lati interi, senza ruotarli, all’interno di rettangoli più grandi, da usare in numero minimo.

Va ricordato infine che gli algoritmi di *fitting* (ossia di *adattamento*) che non prevedono l’ordinamento dei pesi trovano applicazione *on-line* in parecchie circostanze, quando si devono allocare gli oggetti nell’ordine in cui arrivano, *senza conoscere i successivi*, come capita ad esempio nella gestione della memoria dei sistemi di elaborazione e nello *scheduling* di processi su macchine che lavorano in parallelo. Anche in questo caso non mancano sorprese: ad esempio, si provi la strategia *first fit* coi pesi 11, 14, 11, 2, 9, 3, 6, 4 (in quest’ordine) e contenitori di capacità 20; poi si ripeta la prova senza il peso 2. È noto un algoritmo on-line con fattore di garanzia asintotica 1.58889 (Steven S. Seiden, 2002) e un altro con fattore di garanzia assoluta 5/3 (János Balogh e altri, tra cui G. Dósa e J. Sgall, 2015).

Per il problema dell’imballaggio, sarà sensata una strategia *worst fit*, che scelga il contenitore la cui capacità residua sia la *massima* sufficiente a collocare l’oggetto correntemente considerato? (Ovviamente, si sceglierà un nuovo contenitore soltanto quando non se ne potrà fare a meno.) L’intenzione è quella di ridurre la comparsa di *piccole* capacità residue…

## Il problema dello zaino.

Se quelli del commesso viaggiatore e dell’imballaggio sono tipici problemi rispettivamente di permutazione e di partizione, il problema dello zaino rientra tra quelli di sottoinsieme. Nell’usuale problema di ottimizzazione dello zaino deve essere *massimizzato* il valore complessivo da trasportare, che costituisce il “valore” di una soluzione. In questo problema, dunque, il dato di input mancante rispetto al corrispondente decisionale è  $V$ , il valore minimo che si desidera trasportare. Pertanto, si può così enunciare: dato un intero positivo  $P$ , che indica il peso massimo sopportato dallo zaino, e date due liste di  $n$  interi positivi ciascuna,  $(p_1, \dots, p_n)$  e  $(v_1, \dots, v_n)$ , che rappresentano rispettivamente i pesi e i valori di  $n$  oggetti, determinare un insieme di oggetti il cui peso complessivo sia, al più,  $P$  e il cui valore complessivo sia il massimo possibile.

Esistono degli algoritmi che sfruttano bene l’idea della programmazione dinamica per risolvere esattamente questo problema: se in qualche maniera abbiamo già parzialmente riempito lo zaino, il maggior profitto lo otterremo massimizzando comunque il valore degli oggetti che vi possono ancora trovar posto, da scegliere ovviamente tra quelli tuttora disponibili.

Ribaltando la prospettiva, possiamo ragionare in questo modo: supponiamo che la capacità del nostro zaino aumenti progressivamente da 1 a  $P$ , e ad ogni stadio chiediamoci quale sia il massimo valore raggiungibile disponendo soltanto del primo oggetto, o dei primi due, o dei primi tre, … o di tutti gli  $n$  oggetti. Dobbiamo dunque calcolare tutti i numeri  $M(c, k)$  per  $c = 1, \dots, P$  e per  $k = 1, \dots, n$ , dove  $M(c, k)$  è il

massimo valore complessivo ottenibile scegliendo dai primi  $k$  oggetti, avendo  $c$  come limite superiore al peso complessivo.

Quando dovremo decidere se scegliere o meno il  $k$ -esimo oggetto, ci chiederemo innanzi tutto se il suo peso supera  $c$ : se sì, ovviamente non lo potremo scegliere, indipendentemente dal suo valore. Altrimenti, lo sceglieremo soltanto nel caso in cui la sua presenza riesca a migliorare il profitto: dovremo quindi considerare il miglior valore (già calcolato) col peso limite  $c - p_k$  e coi primi  $k - 1$  oggetti, e aggiungervi il valore  $v_k$ , per poter prendere di conseguenza la giusta decisione.

L'algoritmo usa una matrice di interi  $M$ , di  $P + 1$  righe per  $n + 1$  colonne, dove la prima riga e la prima colonna (che supponiamo abbiano indice 0) sono inizializzate col valore 0: infatti, se lo zaino non può contenere nulla o non vi è alcun oggetto, allora il suo valore (ottimo) è 0.

**per**  $c = 1, \dots, P$ :

**per**  $k = 1, \dots, n$ :

**se**  $p_k > c$  **allora**

$M(c, k) \leftarrow M(c, k - 1)$

**altrimenti**

$M(c, k) \leftarrow \max \{ M(c - p_k, k - 1) + v_k, M(c, k - 1) \}$

Fatto questo, nell'elemento in ultima riga e ultima colonna, cioè  $M(P, n)$ , è contenuto il valore di una soluzione ottima.

Vediamo subito qual è la matrice costruita su un esempio. Siano  $P = 9$ ,  $n = 6$ , con pesi  $(1, 2, 2, 3, 3, 5)$  e valori  $(1, 1, 4, 2, 3, 5)$ ; qui gli oggetti sono ordinati secondo il peso, ma ciò non ha alcuna importanza. Al termine, la matrice  $M$  risulta:

0	0	0	0	0	0	0
0	1	1	1	1	1	1
0	1	1	4	4	4	4
0	1	2	5	5	5	5
0	1	2	5	5	5	5
0	1	2	6	6	7	7
0	1	2	6	7	8	8
0	1	2	6	7	8	9
0	1	2	6	8	9	10
0	1	2	6	8	10	10

da cui si deduce che il valore ottimo dello zaino è 10.

Ma come si può ora risalire agli oggetti che vi concorrono? Partiamo dall'elemento in basso a destra (che contiene il valore ottimo) e, restando sull'ultima riga, procediamo verso sinistra fino ad incontrare una variazione di valore. Ricordando che righe e colonne sono indicate da 0, riscontriamo la prima variazione tra la colonna 5 e la colonna 4: ciò significa che l'oggetto 5 concorre alla soluzione, che ora può essere ridotta alla soluzione del problema col peso limite  $9 - p_5 = 9 - 3 = 6$  e coi primi 4 oggetti.

Dobbiamo quindi risalire la colonna 4 fino alla riga 6, dove troviamo un 7 alla cui sinistra c'è un 6: subito una variazione. Allora anche l'oggetto 4 concorre alla soluzione, che adesso si riduce al problema col peso limite  $6 - p_4 = 6 - 3 = 3$  e coi primi 3 oggetti. Restiamo sulla colonna 3 e risaliamo alla riga 3, dove si trova un 5 alla cui sinistra c'è un 2: ancora subito una variazione. Scegliamo dunque l'oggetto 3, che ha peso 2; il nuovo peso limite da considerare è  $3 - 2 = 1$ , coi primi 2 oggetti soltanto. In effetti, salendo all'elemento di colonna 2 e riga 1, dobbiamo poi spostarci di due posti a sinistra per trovare una variazione, che denota l'inclusione dell'oggetto 1 nello zaino di valore ottimo. In conclusione, per ottenere lo zaino di valore 10, che è il massimo, prendiamo gli oggetti 1, 3, 4 e 5, per un peso complessivo di 9 unità: in questo caso la capacità dello zaino è pienamente sfruttata. (Notate che qui un algoritmo *greedy* sceglierrebbe dapprima gli oggetti 6 e 3, e poi o 1 o 2, ancora con valore 10 e peso o 8 o 9: quale soluzione è trovata dipende dall'ordinamento iniziale degli oggetti, quando più d'una è ottima. Trovate un'istanza la cui unica soluzione ottima non sia fornita quand'è scelto ad ogni passo l'oggetto di maggior valore possibile.) Se cambiassimo l'ultimo dei valori in input da 5 a 6, nella matrice finale vi sarebbe un mutamento soltanto nei tre elementi in fondo alla colonna più a destra, che risulterebbero incrementati di un'unità. Il valore ottimo dello zaino sarebbe 11, e si troverebbe subito una variazione che indicherebbe l'inclusione dell'oggetto 6: riducendo il problema al peso limite  $9 - p_6 = 9 - 5 = 4$  e ai primi 5 oggetti, si dovrebbe risalire la colonna 5 fino alla riga 4, dove occorrono tre spostamenti a sinistra per trovare la successiva variazione. In conclusione, sarebbero presi gli oggetti 1, 3 e 6, per un peso complessivo di 8 unità.

Vediamo come si può scrivere questa seconda parte dell'algoritmo:

$c \leftarrow P$ ;  $k \leftarrow n$ ;

**finché**  $c > 0$  e  $k > 0$ :

**se**  $M(c, k) \neq M(c, k - 1)$  **allora**

è scelto l'oggetto  $k$

$c \leftarrow c - p_k$

$k \leftarrow k - 1$

Se lo scopo è quello di riempire lo zaino il più possibile, pur di non superare la capacità  $P$ , basta far coincidere i valori degli oggetti con i rispettivi pesi: si pensi, ad esempio, a dei lingotti d'oro di diverse dimensioni. Si provi a risolvere questo problema, con uno zaino di capacità 150 (unità di peso) e otto lingotti di peso 16, 27, 37, 42, 52, 59, 65 e 95: quali dovranno essere scelti? (Qui si riesce a riempire lo zaino al massimo del peso sopportato; ma se la sua capacità scendesse a 149, allora la soluzione ottima lo riempirebbe *quasi* completamente, scegliendo lingotti tutti diversi dall'istanza precedente...)

Se abbiamo più oggetti uguali, di stesso peso e stesso valore, basta farli comparire altrettante volte nelle relative liste: l'algoritmo sopra delineato funziona comunque (si veda, ad esempio, "Ladro chi ruba e chi riempie lo zaino", finale del 2011).

Una variante significativa del problema sottintende l'illimitata disponibilità di esemplari di ciascuno degli oggetti descritti da una coppia di numeri peso-valore:

quindi qui sarebbe del tutto superflua una ripetizione di stesso peso con stesso valore nelle liste dei dati, o di stesso peso con valore più basso, poiché a parità di peso non sarebbero mai scelti esemplari di oggetti di valore inferiore. L'algoritmo che ora descriveremo funzionerebbe comunque.

Questo problema, noto come problema dello zaino *con ripetizioni*, per essere risolto richiede uno spazio di memoria inferiore, ma l'ordine di grandezza temporale non cambia. È sufficiente una sola colonna della matrice usata nel caso precedente, e quindi diciamo che adesso  $M$  è soltanto un *array* di  $P + 1$  interi, i cui elementi hanno indice da 0 a  $P$ .  $M(c)$  è il massimo valore raggiungibile con uno zaino di capacità  $c$ , e se abbiamo già calcolato gli ottimi per le capacità inferiori, allora per calcolarlo basterà chiedersi, per ogni  $k = 1, \dots, n$ , quale profitto si ottiene prendendo lo zaino ottimo di capacità  $c - p_k$  (per quei pesi  $p_k$  che non superino  $c$ ) e aggiungendovi un esemplare dell'oggetto  $k$ -esimo. Naturalmente si sceglierà il massimo.

Ci servirà però un altro array dello stesso formato, chiamiamolo  $S$ :  $S(c)$  ricorderà appunto qual è l'oggetto che è stato scelto per concorrere al valore massimo di uno zaino di capacità  $c$ .

L'algoritmo consisterà dunque dei seguenti passi:

```

 $M(0) \leftarrow 0$ 
per  $c = 1, \dots, P$ :
   $M(c) \leftarrow 0$ ;  $S(c) \leftarrow 0$ 
  per  $k = 1, \dots, n$ :
    se  $p_k \leq c$  allora
       $b \leftarrow M(c - p_k) + v_k$ 
      se  $b > M(c)$  allora
         $M(c) \leftarrow b$ ;  $S(c) \leftarrow k$ 

```

Al termine, in  $M(P)$  è contenuto il valore di una soluzione ottima.

Provandolo sull'istanza del problema con  $P = 9$ ,  $n = 4$ , pesi  $(1, 2, 3, 5)$  e valori  $(1, 3, 5, 9)$ , si ottiene:

$$\begin{array}{rcccccccccccc}
 M & = & 0 & 1 & 3 & 5 & 6 & 9 & 10 & 12 & 14 & 15 \\
 S & = & 0 & 1 & 2 & 3 & 1 & 4 & 1 & 2 & 3 & 1
 \end{array}$$

Il valore ottimo dello zaino è dunque 15.

Quali oggetti lo compongono? Partiamo dall'ultimo elemento di  $S$ , quello di indice  $P = 9$ : vi troviamo 1; ciò significa che è stato scelto un esemplare dell'oggetto 1, e togliendo da 15 il suo valore (che è 1) scendiamo a 14. In corrispondenza del valore 14, che troviamo in  $M(8)$ , abbiamo  $S(8) = 3$ : quindi prendiamo un esemplare dell'oggetto 3, che vale 5. Togliendo 5 da 14 scendiamo a 9, che è il valore di  $M(5)$ ;  $S(5)$  vale 4, e dunque dobbiamo scegliere un esemplare dell'oggetto 4, il cui valore è proprio il residuo 9.

In conclusione, nello zaino ci saranno tre oggetti diversi: 1, 3 e 4, e lo stesso risultato sarebbe stato fornito anche dall'algoritmo senza ripetizioni. Ma se proviamo con  $P =$

10, la scelta cade su due esemplari dell'oggetto 4, che riempiono lo zaino, per un valore complessivo (ottimo) pari a 18.

Scriviamo la parte finale dell'algoritmo:

```

 $b \leftarrow M(P)$ 
per  $c = P, \dots, 1$  (a scendere):
    se  $M(c) = b$  e  $S(c) > 0$  allora
         $k \leftarrow S(c)$ 
        è scelto l'oggetto  $k$ 
         $b \leftarrow b - v_k$ 

```

Entrambi gli algoritmi descritti richiedono un tempo di esecuzione dell'ordine di  $n \cdot P$  (determinato dai due cicli “per” annidati), che *non* è polinomiale nella dimensione dell'input: infatti l'input è costituito da circa  $(n + 1) \cdot \log_2 P + n \cdot \log_2 v_{\max}$  cifre binarie (a rigore, sarebbe polinomiale se i numeri fossero codificati in un sistema *unario*, non pesato, ma la complessità computazionale della procedura non muterebbe).

In tali casi, quando l'input è una sequenza di numeri naturali e il tempo di esecuzione è limitato da un polinomio nel maggiore di tali numeri e nella lunghezza dell'intero input, si parla di algoritmi *pseudo-polynomiali*. E ciò induce a classificare il problema dello zaino tra quelli sì “difficili” dal punto di vista computazionale (nella sua versione decisionale, come abbiamo detto, è NP-completo), ma in un senso un po’ meno “forte” di altri, quali ad esempio i problemi del commesso viaggiatore e dell'imballaggio (che, nella loro forma decisionale, sono *fortemente* NP-completi).

Usando sempre il paradigma della programmazione dinamica, si possono dare versioni che impiegano un tempo dell'ordine di  $n \cdot U$ , dove  $U$  è una maggiorazione del valore della soluzione ottima. (Come potrebbe essere calcolata una tale maggiorazione, affinché non risulti eccessiva?)

Questi algoritmi, basati su una semplice formula ricorsiva, sono dovuti sostanzialmente a Richard E. Bellman e a George B. Dantzig (anni 1956-57); diventano davvero onerosi per *grandi* numeri in input: in tali circostanze si può ricorrere a un'approssimazione, scalando in modo opportuno i pesi o i valori degli oggetti in gioco.

D'altro canto, per istanze con pesi e valori piccoli, si tenga presente che è stato trovato un algoritmo con tempo di ordine  $n \cdot v_{\max} \cdot p_{\max}$  (D. Pisinger, 1999).

Concludiamo dicendo che anche il problema di ottimizzazione dello zaino si mostra non “più difficile” del corrispondente decisionale (sul viceversa nessuno dovrebbe avere dei dubbi: per risolvere il problema decisionale, basterà infatti controllare se il valore ottimo dello zaino è maggiore o uguale al valore minimo fissato appunto per il problema decisionale).

## Problemi NP-ardui.

Abbiamo parlato di tre classici problemi di ottimizzazione – quelli del commesso viaggiatore, dell'imballaggio e dello zaino – e abbiamo detto che sono equivalenti ai

corrispondenti problemi decisionali, che sono NP-completi, e quindi sono pure equivalenti a tutti gli altri problemi NP-completi. (Ricordiamo che “equivalenti” è riferito all’impegno di calcolo, e sostanzialmente vuol dire che si passa dall’uno all’altro in tempo polinomiale.) Essi fanno parte di una classe di problemi che vengono appunto chiamati *NP-equivalenti*.

Ovviamente, la classe dei problemi NP-completi (tutti decisionali) è contenuta in quella degli NP-equivalenti, e tutti quanti, a loro volta, fanno parte di una classe più generale, quella dei cosiddetti problemi *NP-ardui* (o *NP-difficili*; *NP-hard*, secondo la terminologia anglosassone).

Un problema computazionale (decisionale o meno) è quindi detto NP-arduo se è “difficile” almeno quanto i più difficili problemi della classe NP (un po’ più precisamente, se tutti i problemi della classe NP si riducono ad esso in tempo polinomiale).

Alcuni problemi NP-ardui potrebbero essere davvero più difficili di qualsiasi problema in NP. O potrebbero esistere problemi decisionali NP-ardui che non appartengono a NP perché neanche un algoritmo non deterministico è in grado di risolverli in tempo polinomiale.

Citiamo due problemi decisionali, provati NP-ardui, la cui appartenenza a NP è in forte dubbio:

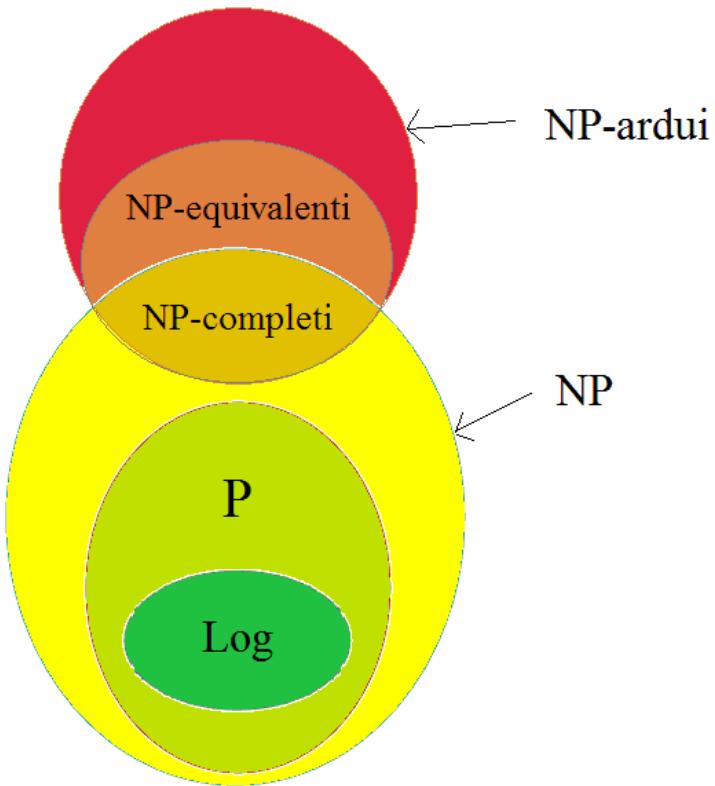
- ◆ data una proposizione come per SAT, stabilire se *la maggior parte* delle assegnazioni di valori di verità alle variabili rende vera la proposizione;
- ◆ dati due interi positivi,  $k$  e  $S$ , e data una lista di  $n$  interi positivi, stabilire se con gli  $n$  numeri in lista si possono formare  $k$  liste, che differiscano l’una dall’altra per almeno un numero, ciascuna delle quali abbia somma maggiore o uguale a  $S$ . (Si noti che certificare la risposta affermativa con  $k$  liste che soddisfino la condizione richiesta, come sembra inevitabile, non può andar bene: significa sì produrre un testimone, ma in generale troppo lungo, poiché  $k$  può essere esponenziale in  $n$ , ad esempio  $k = 2^{n-1}$ .)

Ci sono molte altre questioni aperte, sulle quali sorvoliamo.

Per riassumere, possiamo azzardare il diagramma in figura alla pagina successiva, che è soltanto una parte sommaria di un disegno ben più ampio e articolato.

I problemi più facili, dal punto di vista computazionale, sono quelli che richiedono un tempo logaritmico (l’inverso dell’esponenziale): ad esempio, stabilire se un dato numero è presente in una data sequenza *ordinata*, nel qual caso si può applicare un algoritmo di *ricerca binaria* (ammessa la possibilità di accesso diretto agli elementi della sequenza).

Tuttavia, sebbene pochi lo credano possibile, qualcuno un giorno potrebbe dimostrare che tutti i problemi in NP, e di conseguenza almeno anche quelli NP-equivalenti, si possono risolvere in modo efficiente: nell’eventualità, P e NP si fonderebbero in un’unica classe.



Ritorniamo un attimo sui problemi decisionali NP-completi che compaiono nel breve elenco da noi stilato in precedenza. Notiamo che per alcuni di essi si formulano in maniera abbastanza naturale le corrispondenti versioni “di ottimizzazione”; per altri si tratterà di esibire una soluzione, se c’è, anziché limitarsi a stabilirne l’esistenza o meno, ma spesso i due compiti sono ugualmente impegnativi. Aggiungiamo di seguito qualche notizia.

- ◆ Per i problemi di *soddisfattibilità*, si chiede di massimizzare il numero di clausole vere. Mentre 2SAT è in P, già MAX-2SAT è NP-arduo. Per MAX-3SAT e più in generale per MAX-SAT sono stati trovati algoritmi di approssimazione tempo-polynomiali, rispettivamente con fattore  $8/7$  (di meno non è possibile, ammesso che P sia diverso da NP) e con fattore  $4/3$  (o anche meno). Attenzione: in un problema di massimizzazione, si deve considerare il fattore “reciproco”!
- ◆ Anche ai problemi del *sottografo completo* e dell’*insieme indipendente* corrisponde una massimizzazione (del numero di nodi): per questi non v’è speranza di ottenere algoritmi di approssimazione tempo-polynomiali con fattore costante, a meno che  $P = NP$ .  
Si può dimostrare che ogni grafo (non orientato) con  $n$  nodi ha, al più,  $3^{n/3}$  insiemi indipendenti *massimali* (aggiungendovi un qualsiasi altro nodo non sono più indipendenti), ed esiste un algoritmo per generarli tutti quanti in un tempo di ordine  $n^2 3^{n/3}$ . Naturalmente, tra i massimali vi sono anche i massimi!
- ◆ Il problema della *colorabilità dei nodi* si traduce invece in una minimizzazione: si cerca infatti il cosiddetto *numero cromatico* del grafo dato (non orientato),

ossia il minimo numero di colori da assegnare ai suoi nodi, in modo tale che nessuna coppia di nodi adiacenti abbia lo stesso colore. Esso corrisponde al minimo numero di insiemi indipendenti in cui possono essere ripartiti i nodi del grafo. Dunque, il numero cromatico di un grafo completo con  $n$  nodi è  $n$ .

Per grafi planari, escluso il caso banale di grafo privo di archi, il numero cromatico può essere soltanto 2, 3 o 4. Se vi è almeno un ciclo formato da un numero dispari di archi (e ciò è controllabile in tempo lineare, come abbiamo già osservato) allora 2 è escluso, e le possibilità si riducono a 3 oppure 4. Esiste un algoritmo tempo-polonomiale per colorare qualsiasi grafo planare con quattro colori (N. Robertson e altri, 1996): applicandolo, useremo al massimo *un colore in più* del necessario. Tale procedimento costituisce dunque un algoritmo di approssimazione *assoluto* per la colorazione minima di un grafo planare che abbia almeno un ciclo di lunghezza dispari.

(Suggerisco, come utile esercizio, di provare a disegnare un grafo planare che abbia numero cromatico 4.)

Per grafi qualsiasi, si potrebbe calcolare il massimo numero  $k$  di archi incidenti su un nodo e poi procedere con una facile colorazione *greedy*: si colorano i nodi uno alla volta, assegnando al nodo corrente il colore *minimo* (i colori sono rappresentati da numeri) che non sia stato assegnato ad alcun nodo ad esso adiacente già colorato. Così facendo si useranno, al più,  $k + 1$  colori: ma questa è soltanto una mera maggiorazione del numero cromatico, e tale procedimento non costituisce un algoritmo di approssimazione!

Per questo problema non è noto alcun algoritmo che offra una garanzia di prestazione ragionevole; anzi, nuovamente, se si trovasse un algoritmo di approssimazione tempo-polonomiale con fattore costante, allora il problema stesso ammetterebbe un algoritmo esatto tempo-polonomiale, il che comporterebbe  $P = NP$ .

Come abbiamo visto per il commesso viaggiatore, anche qui può essere usata la tecnica della programmazione dinamica, basata sul fatto che, in una colorazione ottima, ad ogni colore corrisponde un insieme indipendente e ad almeno uno dei colori corrisponde un insieme indipendente massimale. Sfruttando il risultato ricordato sopra per gli insiemi indipendenti massimali, si ottiene infine un algoritmo che richiede un tempo di ordine  $n^2 c^n$  (dove  $c$  è una costante di poco inferiore a 5/2), ma anche uno spazio esponenziale.

- ♦ I problemi di ottimizzazione della *partizione* chiedono di minimizzare la differenza tra le due (o più) parti risultanti: questo è pure l'obiettivo dello *scheduling* (deterministico) di processi indipendenti su due (o più) processori paralleli, già incontrato nel primo capitolo.

Come abbiamo visto, il problema della partizione è un caso particolare di SUBSET-SUM, che a sua volta è un caso particolare di KNAPSACK (ma tutti, ricordiamolo, sono NP-completi); ne consegue che anche per i primi due si può dare un algoritmo pseudo-polonomiale come per il problema dello zaino.

- ♦ Il problema di *sequencing* ottimo altro non è che quello dello *scheduling* (deterministico) di processi indipendenti su un singolo processore, allo scopo di minimizzare la somma delle penalità relative ai lavori conclusi in ritardo. Anche questo problema è risolvibile con un algoritmo di programmazione dinamica in tempo pseudo-polynomiale.

Per concludere la nostra breve rassegna, citiamone qualche variante.

L'istante di tempo 0 coincide sempre con l'inizio del primo processo schedulato. Indichiamo con  $c_i$  l'istante in cui terminerà il processo  $i$ -esimo. Se vogliamo minimizzare la sommatoria dei  $c_i$ , eventualmente pesati coi coefficienti  $p_i$ , allora dobbiamo calcolare i rapporti  $t_i / p_i$  e poi eseguire i relativi processi seguendo l'ordine crescente di tali rapporti (W. E. Smith, 1956). Dunque il problema è chiaramente in P. Su due o più processori paralleli, questa sarebbe soltanto una buona euristica, non più un algoritmo esatto. Quando i  $p_i$  valgono tutti  $1/n$ , significa che l'obiettivo è la minimizzazione del *tempo medio di permanenza* di un processo nel sistema: tale obiettivo è raggiunto schedulando i processi in ordine crescente di tempo di elaborazione  $t_i$  (*shortest-processing-time first*).

Minimizzare invece la sommatoria dei  $(c_j - d_j)$ , ossia dei ritardi rispetto ai tempi desiderati, limitata a quei processi che terminano con un ritardo positivo, è di nuovo un problema NP-arduo, risolvibile ancora in tempo pseudo-polynomiale purché tali ritardi non siano pesati con dei coefficienti  $p_j$  arbitrari. Nel caso particolare in cui i tempi desiderati e gli eventuali pesi dei ritardi siano concordi con i tempi di elaborazione (cioè:  $t_i \leq t_j$  implica che  $d_i \leq d_j$  e  $p_i \geq p_j$ ), il problema è risolvibile in maniera esatta semplicemente schedulando i processi in ordine crescente di tempo di elaborazione, e quindi in tempo polinomiale.

Come dicevamo, le conoscenze raggiunte in proposito inducono a credere che non esistano algoritmi efficienti per nessuno dei problemi NP-ardui – molti pensano che, se ne esistessero, probabilmente qualcuno sarebbe stato trovato! D'altra parte, nessuno è riuscito nemmeno a provare che anche uno solo dei problemi NP-equivalenti sia intrinsecamente esponenziale, cioè che non esista alcun algoritmo efficiente per risolverlo. Evidentemente sfugge ancora all'intuizione umana la *causa essenziale* dell'intrattabilità di un problema... Oppure  $P \neq NP$  sarà indimostrabile nella teoria della complessità computazionale?

Ricordo uno studio, intorno all'inizio del secolo, condotto da un gruppo di ricercatori italiani e francesi, i quali usavano algoritmi ispirati da fenomeni fisici, per cercare di risolvere alcuni problemi NP-completi. Ad esempio, un modello di materiale anti-ferromagnetico servì loro per decidere se i nodi (pur dell'ordine dei milioni) di grafi generati casualmente fossero colorabili con tre colori. Tra i tanti risultati interessanti, essi rilevarono, al crescere del numero di nodi, una soglia piuttosto netta del rapporto tra i numeri di archi e di nodi (2.35) al di sotto della quale, di solito, la colorazione era fatta piuttosto rapidamente, al di sopra invece era presto stabilita l'impossibilità di compierla; lo sforzo computazionale maggiore si concentrava proprio nelle istanze in cui il rapporto tra archi e nodi cadeva in un intorno assai ristretto di detta soglia.

Talvolta, però, il loro procedimento – che, con non tanti nodi, rallentava quando il suddetto rapporto superava 2.2 – con troppi nodi non arrivava ad alcuna risposta... Sebbene i problemi NP-ardui siano presumibilmente intrattabili, in molti casi sono stati realizzati degli algoritmi che “funzionano bene” per la maggior parte delle applicazioni di routine e riescono pure a risolvere all’ottimo certe istanze di dimensione sorprendente: l’abbiamo notato a proposito del commesso viaggiatore. Forse i “casi peggiori” non sono poi così frequenti nella realtà! Tuttavia, l’eventualità che si presenti qualche caso particolarmente insidioso è sempre in agguato: come ribadiamo, un procedimento generale che funzioni in modo efficiente in *tutte* le circostanze è, molto probabilmente, impossibile da ottenersi.

Non possiamo tralasciare una questione importante, ma vi accenneremo soltanto: parlando di problemi decisionali, che cosa si può dire sui loro *complementi*? (La nozione non è nuova: ne abbiamo già discusso a proposito dei problemi indecidibili, nel capitolo precedente.)

Il complemento di un problema decisionale non è altro che il problema (pure decisionale) che ha risposta “sì” in tutti i casi in cui il primo ha risposta “no”, e viceversa.

Ad esempio, il complemento del problema del ciclo hamiltoniano si pone in questi termini: dato un grafo (non orientato), è vero che *non* ha alcun ciclo hamiltoniano? Ebbene, non si sa se questo problema sia in NP; probabilmente no, o almeno finora nessuno ha trovato modo di produrre un testimone, controllabile in tempo polinomiale, per certificare l’eventuale risposta affermativa alla generica istanza di tale problema (o, equivalentemente, l’eventuale risposta negativa alla generica istanza dell’originario problema del ciclo hamiltoniano).

In verità, il punto è che per nessun complemento di problema NP-completo è stato trovato un siffatto testimone. Detto diversamente: tutto fa pensare che i complementi dei problemi NP-completi stiano tutti al di fuori della classe NP, sebbene rientrino ovviamente in quella dei problemi NP-equivalenti (il passaggio da un problema decisionale al suo complemento comporta infatti la semplice inversione della risposta). La “asimmetria” risiede soltanto nel fatto che, per i problemi in NP, è richiesto un testimone per verificare efficientemente le sole risposte “sì”.

Dovrebbe essere altrettanto ovvio che il complemento di un problema in P è ancora in P.

Provate a riprendere uno qualsiasi dei nostri esempi di problemi NP-completi, e a pensare a quale aspetto potrebbe avere un testimone, verificabile in modo efficiente, per la risposta “no” alla generica istanza che non soddisfa la richiesta: non vincereste il milione di dollari messo in palio dal Clay Mathematics Institute (leggete qui: <http://www.claymath.org/millennium-problems/>), perché la vostra scoperta non implicherebbe che P = NP, tuttavia passereste alla storia per aver dimostrato che l’intera classe NP e quella dei suoi complementi in realtà coincidono!

Che dire dei (pochissimi) problemi noti in NP per i quali non si sa se siano NP-completi, né si sa se appartengano a P? Uno l’abbiamo menzionato: decidere se due

grafi arbitrari sono isomorfi; ma dove starà il suo complemento? Di un altro, importante, parleremo tra breve.

## Primalità.

La *teoria dei numeri* è quella branca della matematica che tratta delle proprietà dei numeri interi; è ricca pure di problemi straordinariamente semplici nella loro formulazione, ma notoriamente difficili da risolvere, se non impossibili. Per molti matematici, il suo studio ha – o ha avuto – il fascino di una forma di pura contemplazione, svincolata dal peso di possibili conseguenze pratiche; tuttavia, come si può ben intuire, notevolissime sono le sue applicazioni.

Già all'inizio del XIX secolo, Carl Friedrich Gauss (un nome che non ha bisogno di alcuna presentazione) scrisse, ad esempio, che il problema di distinguere i numeri primi da quelli composti, e di scomporre questi ultimi nei loro fattori primi, è da reputarsi uno dei più importanti e utili dell'aritmetica, e ogni strada deve essere esplorata, con ogni mezzo, per arrivare a un metodo risolutivo il più possibile efficiente.

Il teorema fondamentale dell'aritmetica ci dice che i numeri primi sono i “mattoni” moltiplicando i quali sono costruiti tutti gli altri numeri naturali (esclusi 0 e 1) in modo unico, a parte l'ordine dei fattori.

Partiamo dal *triangolo di Pascal* (o di Tartaglia, sebbene la sua origine, in Oriente, preceda ancora di almeno otto secoli il matematico bresciano Niccolò Fontana detto Tartaglia, dal quale in Italia prende nome), numerando righe e colonne da 0:

	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

eccetera. Il numero che si trova all'incrocio della riga  $n$  con la colonna  $k$  ( $0 \leq k \leq n$ ) ci dice quanti sono i sottoinsiemi distinti che possiamo formare scegliendo  $k$  elementi da un insieme che ne ha  $n$ . Si noti che ciascuna riga è palindroma. La somma dei numeri sulla riga  $n$  fornisce quindi il numero dei sottoinsiemi distinti di un insieme di  $n$  elementi (compresi i due casi limite costituiti dall'insieme vuoto e da tutto l'insieme considerato); tale numero è  $2^n$ .

E, giacché parliamo del triangolo di Tartaglia, descriviamo qual è il modo più conveniente per calcolarlo, riga dopo riga, fino alla riga  $N$ ; a questo scopo, ci serviamo di un array  $T$ , di  $N+1$  elementi interi, con indice da 0 a  $N$ , inizializzati col valore 0:

```

T(0) ← 1
stampa 1 e va a capo
per  $i = 1, \dots, N$ :
    per  $k = i, \dots, 1$  (a scendere):
        T( $k$ ) ← T( $k$ ) + T( $k - 1$ )
        stampa T( $k$ )
    stampa 1 e va a capo

```

Tra le tante meravigliose proprietà di questa struttura vi è quella per cui i numeri che compaiono in una riga  $p$ , con  $p$  numero primo, sono tutti multipli di  $p$ , se si escludono gli 1 iniziale e finale. E questa proprietà vale soltanto per le righe corrispondenti a numeri primi. Teniamola presente.

Parlando di numeri naturali o interi, bisogna anzitutto riconoscere quando un numero (magari calcolato come differenza tra due altri numeri) è multiplo di un certo numero. Dire che  $m$  è multiplo di  $n$  equivale a dire che  $m$  è divisibile esattamente (cioè con resto nullo) per  $n$ , ossia che  $n$  divide (esattamente)  $m$ . Per indicare che la differenza tra  $a$  e  $b$  è multiplo di  $n$  si usa scrivere  $a \equiv b \pmod{n}$ , che si legge “ $a$  è congruo a  $b$ , modulo  $n$ ”: dividendo sia  $a$  sia  $b$  per  $n$  si ottiene lo stesso resto.

Un risultato fondamentale per il nostro discorso fu enunciato dal celeberrimo Pierre de Fermat nel 1640, ed è noto come “*piccolo teorema di Fermat*”: se  $p$  è un numero primo, allora, per ogni naturale  $b$  non multiplo di  $p$ , il numero  $b^{p-1} - 1$  è multiplo di  $p$ , ossia  $b^{p-1} \equiv 1 \pmod{p}$ .

Come corollario,  $b^m \equiv b^n \pmod{p}$  se  $m \equiv n \pmod{p-1}$ , con  $m$  e  $n$  positivi; un caso particolare è  $m = p$  e  $n = 1$ . Ciò è servito come punto d'avvio per l'ideazione del sistema di crittografia RSA, a cui accenneremo nell'ultimo paragrafo.

In forma equivalente, il piccolo teorema di Fermat può essere così enunciato: se  $p$  è primo, allora  $p$  divide  $X^p - X$  per ogni  $X$  (ci riferiamo ai polinomi sugli interi): vale infatti l'uguaglianza  $X^p - X = X(X^{p-1} - 1)$ , e se  $X$  non è multiplo di  $p$ , allora  $p$  deve dividere  $X^{p-1} - 1$ .

Assumendo vera la proprietà del triangolo di Tartaglia notata poc'anzi – di cui tralasciamo la dimostrazione – possiamo facilmente provare il piccolo teorema di Fermat per induzione su  $X$ .

Se  $X = 0$  (caso base), allora ciò che si vuol provare è chiaramente vero, poiché  $p$  divide 0.

Resta da mostrare che, se ciò che si vuol provare vale per  $X$  (ipotesi induttiva), allora vale anche per  $X + 1$ . Supponendo che valga per  $X$ , si ha che se  $p$  è primo, allora  $X^p \equiv X \pmod{p}$  per ogni  $X$ ; d'altra parte, se sviluppiamo il binomio  $(X + 1)^p$ , tutti i coefficienti tranne gli 1 iniziale e finale sono multipli di  $p$ , per cui rimangono soltanto il primo e l'ultimo termine dello sviluppo:  $(X + 1)^p \equiv (X^p + 1) \pmod{p}$ , e quindi  $(X + 1)^p \equiv (X + 1) \pmod{p}$  per l'ipotesi induttiva.

E questo completa la dimostrazione per induzione su  $X$ .

(Un altro modo ancora di dire la stessa cosa è che se  $n$  non divide  $X^n - X$  per qualche  $X$ , allora  $n$  non è primo.)

Purtroppo, *non vale il viceversa* di questo comunque utilissimo teorema: la condizione “ $p$  è un numero primo” è sufficiente, ma non necessaria. Il più piccolo *pseudoprimo assoluto* (capace di inficiare il “test di Fermat” per ogni “base”  $b$ ) è 561 (R. D. Carmichael, 1910):  $561 = 3 \times 11 \times 17$  ma, per ogni  $b$  non multiplo di 561,  $b^{560} - 1$  è divisibile per 561. Per inciso, basta considerare per  $b$  tutti i valori compresi tra 2 e 560; inoltre, non occorre calcolare esplicitamente un numero grande quanto  $b^{560} - 1$  (per fortuna, altrimenti non ci sarebbe alcuna speranza di riuscita!) dato che lo scopo è soltanto quello di verificarne la divisibilità per 561. Si possono infatti applicare le regole dell’*aritmetica modulare* sviluppata da Gauss, ricordando sostanzialmente che, rispetto a un qualsiasi divisore positivo, il resto del prodotto di due numeri è congruo al prodotto dei due resti, e dunque – in particolare – il quadrato di un numero è congruo al quadrato del resto del numero stesso (come abbiamo detto, essere *congruo* significa avere lo stesso resto rispetto al divisore considerato):  $a^2 \equiv (a \text{ mod } n)^2 \pmod{n}$ .

Esistono infiniti pseudoprimi assoluti (W. R. Alford, A. Granville e C. Pomerance, 1994), sebbene siano di gran lunga meno frequenti dei primi autentici: ce ne sono meno di 250'000 al di sotto di  $10^{16}$ .

A proposito: quanti sono i numeri primi? Circa 23 secoli fa, Euclide diede una mirabile dimostrazione della loro infinità. Per assurdo, supponiamo che  $p$  sia il più grande numero primo, e poniamo  $n$  uguale a 1 più il prodotto di tutti i numeri primi da 2 a  $p$ , presi ciascuno una volta; i casi sono due: o  $n$  è primo, ma questo contraddice l’assunto siccome  $n$  è chiaramente più grande di  $p$ , o  $n$  non è primo, ma allora deve avere necessariamente un fattore primo più grande di  $p$  poiché (per come  $n$  è stato costruito) dividendo  $n$  per qualsiasi numero primo da 2 a  $p$  si ottiene come resto 1.

Tuttavia, è pur vero che i numeri primi diventano sempre più rari man mano che si procede nella successione dei naturali. Verso la fine del ’700, il giovane Gauss stimò la quantità di numeri primi minori di  $n$  in circa  $n / \ln(n)$ , dove  $\ln$  è la funzione logaritmo naturale, risultato che poi migliorò... In effetti, già questa stima diviene più accurata al crescere di  $n$ , ma ancor oggi non è nota una “legge” precisa a tale scopo, e tanti sono i problemi riguardanti i numeri primi rimasti insoluti nel corso dei secoli: un esempio famoso è la congettura di Goldbach, di cui abbiamo già parlato.

Quando si tratta di indagare sulla primalità o meno di un qualsiasi numero “grande”, diciamo di centinaia di cifre decimali, non si può certo procedere per tentativi: eseguendo dieci miliardi di divisioni ogni secondo, si impiegherebbe qualcosa come  $10^{130}$  anni – l’età dell’universo è dell’ordine di  $10^{10}$  anni! – per verificare che un dato numero di 300 cifre decimali è primo, considerando come potenziali divisorì tutti i numeri primi fino alla sua radice quadrata. La ragione che ci autorizza a fermarci quando il divisore controllato eccede la radice quadrata del numero dato è semplice: se il numero è composto e ha un fattore primo maggiore della propria radice quadrata, allora deve averne anche uno minore.

Dopo Fermat, i matematici cercarono delle condizioni almeno sufficienti a determinare la primalità. Il *teorema di Wilson* (1770) afferma che  $p$  è primo se e soltanto se  $(p - 1)! + 1$  è multiplo di  $p$ : elegante, ma non porta ad alcun metodo praticabile. Una forma un po' più forte del “viceversa” del teorema di Fermat fu dimostrata da Lucas e poi migliorata da Derrick H. Lehmer nel 1927: se esiste  $a$  ( $1 < a < p$ ) tale che  $a^{p-1} \equiv 1 \pmod{p}$  e  $a^k \not\equiv 1 \pmod{p}$  per ogni esponente  $k$  ottenibile dividendo  $p - 1$  per un suo fattore primo, allora  $p$  è primo.

Nel 1876 Lucas l’aveva provato per ogni  $k < p - 1$ , riuscendo nell’ultimo anno della sua vita (1891) a indebolire la condizione: per ogni  $k$  divisore proprio di  $p - 1$ . (Ad esempio, per  $p = 13$  gli esponenti  $k$  considerati da Lucas sono 2, 3, 4 e 6, mentre Lehmer riesce a escludere 2 e 3.)

Nel 1974 il teorema di Lehmer ispirò un’idea feconda a Vaughan R. Pratt per attribuire un “certificato di primalità” ad ogni numero primo (fino ad allora non era noto se vi fosse un testimone e, nell’eventualità, quale forma potesse avere), controllabile in tempo polinomiale nella lunghezza in bit del numero stesso: sicché egli riuscì a dimostrare, in una pubblicazione dell’anno successivo, che il problema della primalità appartiene alla classe NP. Come abbiamo già osservato, il suo complemento era notoriamente in NP: si trattava allora di uno di quei famosi problemi che stanno in NP col loro complemento!

Il certificato di Pratt consiste in realtà di un insieme di certificati: uno per  $p$  e gli altri per i fattori primi di  $p - 1$ , e così via, ricorsivamente. Per induzione, si può provare che ogni numero primo ammette un certificato siffatto: così Pratt riuscì a invertire il teorema di Lehmer, per i numeri primi maggiori di 2.

Qui ci limitiamo a dare un esempio. Il certificato di primalità del numero 9439 potrebbe essere costituito anzitutto dalla lista [9439, 22, 2, 3, 11, 11, 13]: il primo è il numero  $p$ , il secondo gioca il ruolo di  $a$  nell’enunciato del teorema di Lehmer, i successivi sono i fattori primi di  $p - 1$ .

Per controllarlo, bisogna per prima cosa appurare che  $9439 - 1 = 2 \times 3 \times 11 \times 11 \times 13$ , e poi (usando le operazioni nell’aritmetica modulo 9439) calcolare:

$$22^{9438} \equiv 1 \quad 22^{9438/2} = 22^{4719} \equiv 9438 \quad 22^{9438/3} = 22^{3146} \equiv 733 \\ 22^{9438/11} = 22^{858} \equiv 6468 \quad 22^{9438/13} = 22^{726} \equiv 4139$$

(il primo è 1, gli altri quattro no, come deve essere). Per inciso, qui 22 è il numero *minimo* (anche se non è richiesto) che soddisfa le condizioni del teorema di Lehmer; poteva andar bene anche 23, per esempio. Si rammenti che qui non ci si deve preoccupare di *come generare* un certificato per un certo numero primo: l’importante è che sia *verificabile* in tempo polinomiale!

Ci dovranno essere poi le liste che attestano la primalità di 13, 11 e 3, mentre 2 (il primo numero primo, e unico pari) si assume primo senza bisogno di certificato: [13, 2, 2, 2, 3], [11, 2, 2, 5], [3, 2, 2]; si è aggiunto il 5, e quindi ci vuole un certificato anche per il 5: [5, 2, 2, 2]. Per ognuna di queste liste devono essere svolti i controlli di cui sopra. Sebbene non sia immediato constatarlo, la mole di calcoli da eseguire cresce circa proporzionalmente alla potenza 4 del numero di cifre di  $p$ .

Dalla metà degli anni '70 furono sviluppati diversi algoritmi *probabilistici* per azzardare una risposta circa la primalità di un dato  $n$ . Essi usano anche un altro numero, preso "a caso": un po' come se scegliessimo un certo  $b$  nel "test di Fermat", sperando di non aver a che fare con uno pseudoprimo rispetto a  $b$  (in tal caso, ritenendo  $n$  primo, che in realtà non è, avremmo un *falso positivo*). L'algoritmo di Solovay e Strassen (1977) e quello di Miller e Rabin (1980) sono più raffinati, ma possono dare sempre qualche *falso positivo*; se dicono che  $n$  è composto, lo è senz'altro. Invece, l'algoritmo di Adleman e Huang (1992) può dare soltanto qualche *falso negativo*; se dice che  $n$  è primo, lo è senz'altro (e lo certifica). Tutti sono polinomiali (circa di terzo grado) e possono trarre beneficio da metodi efficienti di moltiplicazione; tuttavia, all'atto pratico, l'ultimo è lento. Nelle applicazioni concrete, ad esempio in crittografia, gli esperimenti possono essere ripetuti in parallelo, su entrambi i tipi di algoritmi, con la speranza di ottenere presto una risposta certa in uno dei due sensi...

Fino al 2002, il miglior metodo *deterministico* disponibile era quello elaborato nella prima metà degli anni '80 da L. M. Adleman, C. Pomerance e R. S. Rumely, e successivamente migliorato da H. Cohen e H. W. Lenstra, spesso chiamato "test APRCL" dalle iniziali dei loro cognomi.

Anche il test APRCL è basato sul piccolo teorema di Fermat, ma è così sofisticato da non poter essere "ingannato" da alcuno pseudoprimo; l'algoritmo richiede un tempo dell'ordine del numero di cifre elevato a un esponente proporzionale al logaritmo del logaritmo del numero di cifre stesso, e quindi non è polinomiale poiché la dimensione dell'input compare per l'appunto all'esponente.

Nell'agosto del 2002 suscitò ammirazione – più che stupore – la notizia che tre matematici indiani, M. Agrawal, N. Kayal e N. Saxena, avevano scoperto un algoritmo tempo-polinomiale – per giunta elegante e relativamente semplice – per decidere la primalità. Sicché, come molti nella comunità scientifica avevano già ritenuto plausibile, il problema della primalità veniva collocato precisamente nella classe P, col suo complemento. Il frutto del lavoro dei tre fu da subito disponibile in rete, per i ricercatori di tutto il mondo; alcuni trovarono qualche miglioramento, compresi i tre ideatori.

Al momento, la complessità è espressa da un polinomio all'incirca di sesto grado nella lunghezza del numero sottoposto al test (H. W. Lenstra e C. Pomerance, 2011): parecchio più lento, dunque, rispetto ai test probabilistici, ma sicuro.

La giustificazione teorica del test AKS parte da una generalizzazione del piccolo teorema di Fermat: dato  $p > 1$  e preso un qualsiasi  $a$  tale che  $a$  e  $p$  siano primi tra loro, si ha che

$$(X + a)^p \equiv (X^p + a) \pmod{p} \text{ se e soltanto se } p \text{ è primo.}$$

Ricordiamo la proprietà del triangolo di Tartaglia; troppi sarebbero tuttavia i monomi da controllare nell'espansione binomiale: si pensi a un  $p$  di sole 50 cifre! Per rendere praticabile questa via, i tre indiani hanno pensato ai resti di un'ulteriore divisione, questa volta per un binomio della forma  $X^r - 1$ , con  $r$  primo e

ragionevolmente piccolo rispetto a  $p$ . Ma adesso, sebbene non sia ovvio, c'è qualche  $a$  che soddisfa l'equazione di cui sopra anche nel caso in cui  $p$  sia composto: in altre parole, si perde la sufficienza della condizione di primalità. Ebbene, i tre ricercatori hanno trovato un modo per scegliere oculatamente  $r$ , sì da controllare pochi  $a$  calcolati *ad hoc*.

Il loro articolo, pubblicato nel 2004, si trova in rete al seguente indirizzo:

[http://www.cse.iitk.ac.in/users/manindra/algebra/primality\\_v6.pdf](http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf).

## Fattorizzazione.

Allo stesso stadio di conoscenza in cui si trovava tra il 1975 e il 2002 il problema (decisionale) della primalità, ve n'è tuttora un altro: la versione decisionale del problema della *fattorizzazione*. Dati due numeri,  $n$  e  $k$ , si deve stabilire se  $n$  ha un fattore proprio (cioè maggiore di 1 e minore di  $n$  stesso) che sia minore o uguale a  $k$ . Si noti che  $k$  può essere un numero qualsiasi: è evidente che quando  $k$  è maggiore o uguale a  $n - 1$ , oppure precisamente uguale alla radice quadrata (troncata) di  $n$ , allora le istanze del problema sono equivalenti a quelle della non primalità. Ma come la mettiamo col testimone?

Questa volta è facile verificare che sia il problema della fattorizzazione sia il suo complemento appartengono a NP. Consideriamo il problema della fattorizzazione: un testimone della risposta “sì” è semplicemente un fattore proprio di  $n$  che sia minore o uguale a  $k$ , mentre un testimone della risposta “no” è la scomposizione di  $n$  in fattori primi, in cui ciascun fattore primo risulta maggiore di  $k$  ed è accompagnato dal proprio certificato di primalità.

Per dirla con Edmonds, il problema della fattorizzazione (che si dimostra non più facile della scomposizione in fattori primi) ha quindi una “buona caratterizzazione”. Tuttavia, la maggior parte degli specialisti non ritiene che prima o poi ne sarà dimostrata l'appartenenza alla classe P, come successe invece per il problema della primalità. Nel prosieguo cercheremo di intuirne la ragione.

Per numeri di forma particolare sono stati studiati test di primalità specifici, più veloci di quelli generali. Una classe famosa è costituita dai *numeri di Mersenne*:  $M_n = 2^n - 1$ . Nel 1644, il monaco francese Marin Mersenne affermò che  $M_n$  è primo per  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127$  e  $257$ , ed è composto per ogni altro  $n < 257$ . Su quale base azzardò quest'affermazione non si sa con certezza; comunque andò vicino alla verità, poiché commise cinque errori:  $M_{67}$  e  $M_{257}$  non sono primi, mentre  $M_{61}, M_{89}$  e  $M_{107}$  lo sono – e ci vollero tre secoli per arrivare a completare la verifica, nel 1947! Per inciso, il tredicesimo primo di Mersenne,  $M_{521}$ , un numero di 157 cifre decimali, è stato scoperto soltanto nel 1952, a Los Angeles, grazie a uno dei primi “potenti” calcolatori, chiamato SWAC; fino a quell'anno, si doveva a Lucas la prova di primalità del più grande numero primo conosciuto:  $M_{127}$ , che ha 39 cifre decimali. Ancor oggi, i più grandi numeri primi conosciuti sono numeri di Mersenne.

Dal 1996 è attivo un progetto di ricerca basata sul calcolo distribuito tra tanti computer sparsi in tutto il mondo (GIMPS, *Great Internet Mersenne Prime Search*,

<http://www.mersenne.org/>), grazie al quale sono stati trovati gli ultimi 17 dei 51 numeri primi di Mersenne ad oggi conosciuti ed è stato verificato che, al di sotto del 47° noto, non ve ne sono altri da scoprire. Il più grande ha esponente 82·589·933, quindi ha quasi 25 milioni di cifre decimali – ma è facile scriverlo in binario! È stato scoperto il 7 dicembre 2018, e confermato dopo ben due settimane di “test veloce”. Attualmente, GIMPS può contare sulla disponibilità di quasi due milioni di unità di elaborazione. (Si visiti anche il sito <http://primes.utm.edu/>.)

Se  $n$  non è primo, allora certamente  $M_n$  non è primo; ma anche quando  $n$  è primo,  $M_n$  è quasi sempre composto. Non si sa se i primi di Mersenne siano infiniti. Come si è intuito, i calcoli necessari per cercarli sono lunghissimi e possono costituire un’ottima verifica dell’efficienza e della precisione di un nuovo processore...

Un test di primalità per i numeri di Mersenne fu ideato e messo a punto da Lucas (1876) e poi condensato da Lehmer (1935) in un algoritmo assai semplice (benché non lo sia la matematica che ne costituisce la base): vediamolo.

Sia  $n$  primo, maggiore di 2; calcoliamo la sequenza:

- $U_0 = 4$
- $U_{k+1} = (U_k^2 - 2) \bmod M_n \quad \text{per } k = 0, \dots, n-3$

(qui con  $x \bmod y$  s’intende il resto della divisione di  $x$  per  $y$ ; l’operatore  $\bmod$  si legge “modulo”). Se l’ultimo numero calcolato,  $U_{n-2}$ , risulta uguale a 0 allora  $M_n$  è primo, altrimenti non lo è. La complessità è polinomiale di grado 3, migliorabile sfruttando metodi di moltiplicazione veloce.

Grazie a questo test, Lucas scoprì che  $M_{67}$  è composto; ma il test non dà alcuna informazione sui suoi fattori (lo stesso vale per gli altri test di primalità generali attualmente disponibili). Per sapere quali sono i fattori di  $M_{67}$  si dovette attendere il 1903, quando all’annuale convegno della American Mathematical Society, Frank N. Cole scrisse alla lavagna:

$$2^{67} - 1 = 147 \cdot 573 \cdot 952 \cdot 589 \cdot 676 \cdot 412 \cdot 927 = 193 \cdot 707 \cdot 721 \times 761 \cdot 838 \cdot 257 \cdot 287$$

e lo verificò, passo dopo passo, nel silenzio dell’aula. Alla ricerca di questi due fattori primi (non di Mersenne) di  $M_{67}$  pare che Cole avesse dedicato le domeniche di alcuni anni...

Ma come si possono trovare i fattori di un numero che sappiamo essere composto? La domanda non è affatto oziosa; ad esempio, se si riuscisse a scoprire un metodo abbastanza efficiente, andrebbero in crisi diversi sistemi di crittografia oggi ritenuti sufficientemente sicuri per la trasmissione di informazioni riservate e per la protezione di reti internazionali di dati: uno di essi sarà illustrato nel prossimo paragrafo.

Una volta appurato, per tentativi progressivi, che un “grande” numero  $n$  non ha fattori primi “piccoli”, si potrebbe procedere con un semplice metodo dovuto a Fermat, che inizia l’esplorazione partendo dalla radice quadrata del numero  $n$  in questione: accenniamolo.

Supponiamo  $n = u \cdot v$ , con  $u, v$  dispari “grandi” e  $u \geq v$ .

Siano  $x = (u + v)/2$ ,  $y = (u - v)/2$ . Allora si ha  $0 \leq y < x \leq n$ ,  $u = x + y$ ,  $v = x - y$ ; per cui:  $n = u \cdot v = (x + y) \cdot (x - y) = x^2 - y^2$ , ossia:  $y^2 = x^2 - n$ .

Se si trovano  $x$  e  $y$  che soddisfano quest’ultima equazione, allora la scomposizione di  $n$  è data da  $(x + y) \cdot (x - y)$ . Si noti che non è detto che questi due fattori,  $(x + y)$  e  $(x - y)$ , siano primi.

Sia  $k$  il più piccolo numero naturale maggiore o uguale alla radice quadrata di  $n$ ; proviamo successivamente i valori  $x = k$ ,  $x = k + 1$ ,  $x = k + 2$ , ..., controllando ogni volta se  $x^2 - n$  è un quadrato perfetto (il quadrato di  $y$ ).

Se  $n$  ha due fattori all’incirca della stessa grandezza (quindi vicini alla radice quadrata di  $n$ ) allora la soluzione dovrebbe essere trovata abbastanza rapidamente (altrimenti rimane il grosso problema di come trovare  $x$ ).

Ci sono vari modi per sveltire ulteriormente questo procedimento: ad esempio, tenendo presente che un quadrato perfetto non termina mai con 2, 3, 7 o 8.

Maurice Kraïtchik era un matematico belga di origine russa, che negli anni ’20 del Novecento scrisse alcuni trattati di teoria dei numeri e nel decennio successivo curò la pubblicazione di un periodico, *Sphinx*, dedicato alla matematica ricreativa. Egli stesso compilò un paio di volumi di giochi matematici, ristampati anche in anni recenti dalla casa editrice Dover di New York. In uno dei suoi libri pubblicato nel 1926, forse riprendendo un’idea di Gauss, compì il più grande passo avanti nell’ambito del problema che stiamo considerando: generalizzò il metodo di Fermat della “differenza di quadrati”, gettando le basi per molti dei moderni algoritmi di fattorizzazione.

Kraïtchik propose di cercare coppie  $(x, y)$  tali che  $x^2 - y^2$  (ma non  $x - y$ ) sia un multiplo di  $n$ , anziché soltanto  $n$  stesso, ossia tali che

$$x^2 \equiv y^2 \pmod{n} \quad \text{e} \quad x \not\equiv y \pmod{n}$$

Una qualsiasi coppia  $(x, y)$  che soddisfi queste condizioni fornisce come fattori propri di  $n$  il massimo comun divisore di  $(x + y, n)$  e di  $(x - y, n)$ . E il massimo comun divisore può essere calcolato assai efficientemente con l’algoritmo euclideo.

Si noti che ogni coppia  $(x, y)$  tale che  $x \equiv y \pmod{n}$  soddisfarebbe la condizione  $x^2 \equiv y^2 \pmod{n}$ , ma porterebbe alla banale scomposizione di  $n$  nei fattori 1 e  $n$ .

Nel 1984 Carl Pomerance ha dimostrato che se  $n$  ha almeno due diversi fattori primi dispari (come è nei casi che ci interessano), allora almeno la metà delle coppie  $(x, y)$  tali che  $x^2 \equiv y^2 \pmod{n}$  soddisfano anche la condizione  $x \not\equiv y \pmod{n}$ , purché sia  $x$  sia  $y$  non abbiano fattori primi in comune con  $n$ . Inoltre, quando  $n$  è il prodotto di due diversi numeri primi  $p$  e  $q$  (come nel sistema di crittografia RSA), se una coppia  $(x, y)$  soddisfa la prima condizione allora in ben due casi su tre soddisfa pure la seconda.

L’attenzione si è quindi spostata su come produrre una coppia di interi con le caratteristiche desiderate. In effetti, i raffinati procedimenti di fattorizzazione messi a punto dai primi anni ’80 (J. D. Dixon, C. Pomerance) ad oggi (l’algoritmo GNFS, *General Number Field Sieve*) differiscono nella maniera in cui è condotta tale

ricerca. Inoltre, questi algoritmi hanno l'interessante peculiarità di poter essere utilizzati in modo tale da ripartire il calcolo su diversi elaboratori indipendenti (così come nel progetto per la ricerca dei numeri primi di Mersenne), per poi ricombinare i risultati parziali in una fattorizzazione complessiva. Sicché, nel dicembre del 2009, è stato finalmente scomposto nei suoi due fattori primi un modulo  $N$  di un sistema RSA di 768 bit (232 cifre decimali); l'elaborazione ha richiesto, allora, l'equivalente di duemila anni di lavoro di un comune personal computer.

Nonostante certi ragguardevoli successi, rimane il fatto che, in generale, questi procedimenti presentino una complessità computazionale incomparabile con quella dei test di primalità: allo stato attuale, come abbiamo detto, il problema della fattorizzazione sembra essere intrinsecamente più difficile di quello della primalità. Si consideri, ad esempio, che soltanto nel 2004 fu calcolato un fattore (di 53 cifre decimali) di  $M_{971}$  (di 293 cifre decimali), che in quel momento era il più piccolo numero di Mersenne del quale non si conoscessero fattori, pur essendo nota da oltre mezzo secolo la sua non-primalità.

## L'algoritmo di crittografia RSA.

Notevoli applicazioni pratiche di alcuni importanti risultati della teoria dei numeri si hanno nella *crittologia* (dal greco antico κρυπτός, “nascosto”), lo studio sia delle condizioni di riservatezza nelle comunicazioni, sia delle tecniche di protezione dei dati sensibili o privati. Ciò che si vuol “nascondere” è il significato di un messaggio, e quindi – al fine di renderlo incomprensibile – lo si altera per mezzo di un procedimento concordato a suo tempo tra il mittente e il destinatario; quest’ultimo può ricavare il testo originale, rovesciando in qualche modo tale procedimento.

Il primo, arcinoto, caso documentato d’impiego militare di un’ingenua cifratura per sostituzione letterale si deve a Giulio Cesare, impegnato nella guerra gallica: ogni lettera era sostituita, ad esempio, con la terza successiva nell’alfabeto, a rotazione. Un significativo passo avanti fu fatto, molto tempo dopo, coi *cifrari a sostituzione polialfabetica*: una stessa lettera può essere sostituita da lettere diverse e, viceversa, a una stessa lettera del messaggio cifrato (o crittogramma) possono corrispondere lettere differenti nel testo in chiaro. Il primo cifrario di questo tipo, il cosiddetto “disco cfrante”, risale alla seconda metà del Quattrocento e il suo ideatore fu l’architetto e umanista Leon Battista Alberti, che tuttavia non si preoccupò della diffusione e delle possibili applicazioni di questa sua importante scoperta.

Trascorso circa un secolo, ebbe maggior successo il più semplice sistema di cifratura che prese nome dal diplomatico francese Blaise de Vigenère (il quale lo pubblicò in un suo trattato del 1586) ma già descritto dal bresciano Giovan Battista Bellaso in un opuscolo stampato a Venezia nel 1553.

Usiamo una notazione più moderna e compatta per illustrarlo brevemente e proporre un quesito al lettore. La “chiave”  $K$  era una sequenza di lettere abbastanza lunga (diciamo costituita da  $k$  lettere, indicate da 0 a  $k-1$ ), ma di solito non raggiungeva la lunghezza del messaggio da cifrare  $M$  (diciamo di  $m$  lettere, indicate da 0 a  $m-1$ ).

La chiave era quindi scritta ripetutamente sotto la sequenza M, dopodiché tutte le coppie di lettere (sopra quella del messaggio in chiaro, sotto quella della chiave) erano sommate modulo 26 (considerando l’alfabeto inglese e facendo corrispondere, come d’uso, la lettera A a 0 e la lettera Z a 25). Si otteneva così il crittogramma C:

$$C(i) = (M(i) + K(i \bmod k)) \bmod 26, \text{ per } i = 0, \dots, m - 1.$$

Si noti che se la chiave è costituita da una sola lettera – che sostituisce  $K(i \bmod k)$  nella precedente relazione – il metodo si riduce a uno degli antichi cifrari (a sostituzione monoalfabetica) già impiegati ai tempi di Giulio Cesare, facilmente decifrabili anche senza l’ausilio del computer: soltanto 25 sono infatti le possibilità. Chi riceveva il crittogramma – oltre che sapere quale metodo era stato impiegato – doveva conoscere la chiave e usare il procedimento inverso per decifrarlo.

Trascorsi altri tre secoli, sia Babbage (che già conosciamo) sia l’ufficiale prussiano Friedrich W. Kasiski riuscirono, indipendentemente l’uno dall’altro, a individuare il punto debole di tale sistema ritenuto “indecifrabile”, sfruttando tre cose: le sequenze ripetute nel messaggio cifrato, le relative distanze e la frequenza di ciascuna lettera nella lingua presumibilmente usata. In effetti, l’unica garanzia affinché il metodo di Vigenère possa ritenersi sicuro è che la chiave sia lunga (almeno) quanto il testo (cioè  $k \geq m$ ), sia generata in maniera del tutto casuale, e sia usata una sola volta (Claude E. Shannon, 1949): assai scomodo, dato che le chiavi devono essere in possesso sia del mittente sia del destinatario!

Propongo una sfida, con un suggerimento: riuscire a decifrare il crittogramma che segue, sapendo che il messaggio in chiaro è in italiano e la chiave usata è una parola che ricorre più volte in questo libro!

CPRXCWYSFKLRZHOIOXSQHKGNFZCLRBYWUGSLFOVVWUTSPYLRKJCRI  
AERBJWLZIMMIEKOLVOFSZMGFYKINDIVVFIXFUWMVIVYUSMSDRXONV

Fino alla metà degli anni ’70 del Novecento, tutte le tecniche di scrittura segreta erano *simmetriche*. Ciò significa che il procedimento di decifrazione è semplicemente il procedimento di cifratura eseguito al contrario: in sostanza, mittente e destinatario dispongono di informazioni equivalenti, e usano una stessa chiave per cifrare e decifrare. Questo implica che ciascun mittente e ciascun destinatario debbano accordarsi non soltanto sulla procedura, ma anche sulla chiave da usare (infatti, la procedura cifrerà il messaggio in modo dipendente dalla chiave scelta, cosicché solo conoscendo quest’ultima sarà possibile decifrare con facilità il crittogramma), e l’accordo deve avvenire – ovviamente – in condizioni di massima sicurezza: un po’ problematico, quando i potenziali mittenti (e magari i destinatari) sono tanti e nemmeno si conoscono di persona, come accade, ad esempio, nelle transazioni commerciali via internet.

Quest’ostacolo è noto come *problema di distribuzione (della chiave)*: ciascun destinatario si deve preoccupare di prendere le necessarie precauzioni per poter trasmettere la chiave per cifrare, in condizioni di sicurezza, a tutti i potenziali mittenti.

Negli ambienti universitari, tra il 1975 e il 1977 furono ideati i sistemi a chiave *asimmetrica*: la chiave usata per cifrare e quella usata per decifrare non coincidono; in particolare, ciò significa che, se io ho cifrato un messaggio, lo potrò decifrare facilmente soltanto se conosco anche la seconda chiave. Allora il destinatario può rendere *pubblica* la prima chiave (quella per cifrare) in modo che chiunque possa adoperarla, tenere per sé la seconda (detta chiave *privata*) e con questa decifrare tutti i messaggi (cifrati con la chiave pubblica) che gli arrivano dai diversi mittenti. Le chiavi pubbliche dei diversi destinatari potrebbero essere inserite in una sorta di elenco telefonico, certificato e magari diffuso in rete...

Questa idea, di per sé rivoluzionaria, già concepita da B. W. Diffie e M. E. Hellman nel 1975, fu realizzata un paio d'anni dopo, quando R. L. Rivest, A. Shamir e L. M. Adleman riuscirono a escogitare un algoritmo di cifratura (chiamato RSA, dalle iniziali dei loro cognomi) che può essere agevolmente capovolto *soltanto se* si dispone di un'ulteriore informazione (la chiave privata, appunto), oltre a quella servita per la cifratura. Il loro articolo, pubblicato nel 1978, si trova in rete al seguente indirizzo:

<http://people.csail.mit.edu/rivest/Rsapaper.pdf>.

Il cuore del sistema di crittografia asimmetrica RSA è infatti una funzione facilmente calcolabile (con l'aritmetica modulare) ma difficilmente invertibile. Per sommi capi, il procedimento è di seguito illustrato ed esemplificato, a beneficio di chi voglia saperne di più. Al nostro scopo, è sufficiente dire che ciascun destinatario dovrà scegliere una (sua esclusiva) coppia di numeri primi abbastanza grandi e divulgare il loro prodotto; la segretezza dei messaggi a lui inviati dipenderà dall'eventualità che nessuno riesca a scomporre il numero pubblicato nei suoi due fattori primi.

Il sistema RSA sfrutta conoscenze matematiche tutte peraltro già note al tempo di Fermat e fonda la propria sicurezza sulla presunta intrattabilità del problema della fattorizzazione. D'altra parte, per usarlo, occorre generare in modo efficiente numeri primi sufficientemente grandi: persino di parecchie migliaia di bit, volendo ottenere una buona affidabilità!

Vediamolo passo per passo, con l'aiuto di un semplicissimo esempio (didattico).

- Il destinatario sceglie due numeri primi  $p$  e  $q$ , da tenere segreti, molto grandi (ma non troppo vicini, altrimenti c'è il rischio che siano individuati da un buon algoritmo di fattorizzazione come GNFS) e un numero  $k$  che non abbia fattori primi in comune col prodotto  $(p - 1) \cdot (q - 1)$ ; calcola  $N = p \cdot q$  e rende pubblica la chiave costituita dalla coppia  $(N, k)$  in un elenco accessibile a chiunque.  $N$  deve essere esclusivo di quel destinatario, mentre diversi destinatari possono avere lo stesso  $k$ .

Inoltre, ricava la propria chiave privata  $d$  dalla formula

$$k \cdot d \equiv 1 \pmod{(p - 1) \cdot (q - 1)}$$

Da questa relazione, il valore di  $d$  (qui unico) può essere ricavato mediante un veloce procedimento, noto come *inversione modulare*, basato su un'estensione dell'algoritmo di Euclide per il calcolo del massimo comun divisore.

- *Esempio:* il destinatario sceglie  $p = 167$ ,  $q = 197$ ,  $k = 199$ , e quindi pubblica la chiave ( $N = 32899$ ,  $k = 199$ ). Inoltre deve essere  $199 \cdot d \equiv 1 \pmod{32536}$ , da cui risulta la chiave privata  $d = 327$ .
- Un mittente, che vuole inviare un messaggio al destinatario, trasforma il messaggio in un numero  $M$  (ad esempio, la sequenza dei codici ASCII dei caratteri che lo compongono, interpretata come numero binario; qualora risulti  $M \geq N$ , si dovrà spezzare il messaggio originario in più parti) e quindi calcola il crittogramma da inviare,  $C$ , secondo la formula  $C = M^k \pmod{N}$ . Ricordiamo che, sfruttando l'aritmetica modulare, non occorre calcolare esplicitamente  $M^k$ .

La funzione che a  $M$  (con  $0 \leq M \leq N-1$ ) associa  $C$  (con  $0 \leq C \leq N-1$ ) secondo questa formula è *biunivoca* (e quindi stabilisce una permutazione dei numeri da 0 a  $N-1$ ), ma ha un andamento assai “disordinato” per una metà del suo grafico (l'altra metà può essere ottenuta per simmetria). Dunque, neanche conoscendo un sottoinsieme preponderante (ma sparso) del suo enorme insieme di valori, si potrebbe tentare di invertirla (per estrapolazione).

A puro titolo d'esempio, nella pagina seguente è mostrato il grafico di  $C = M^7 \pmod{187}$  (in ordinata), per  $M = 0, 1, \dots, 186$  (in ascissa); si noti che a 0 è comunque associato 0 (e così a 1 e al massimo, 186, sono associati i valori stessi), mentre i valori in corrispondenza di  $M = 94, \dots, 186$  si possono ottenere per simmetria dai valori corrispondenti a  $M = 1, \dots, 93$ . In altre parole, se prendiamo il grafico da 1 a 186, lo tagliamo a metà in verticale e ruotiamo una delle due metà di 180 gradi, allora le due parti sono perfettamente uguali; lo stesso vale se lo tagliamo in orizzontale.

Se  $N$  avesse anche soltanto una dozzina di cifre decimali, per disegnare l'analogico grafico a puntini che qui potete vedere riprodotto, non basterebbe un foglio che ricoprisse l'intera superficie terrestre!

- *Esempio (continua):* un mittente vuole inviare al destinatario il semplice messaggio costituito dalla parola OK. In codice ASCII, la lettera O maiuscola è rappresentata dal byte 01001111, la K maiuscola dal byte 01001011, quindi OK = 010011101001011 che, letto in base 2, corrisponde al numero  $20 \cdot 299$  in notazione decimale; dunque  $M = 20299$  e  $C = 20299^{199} \pmod{32899}$ .

Ormai dovremmo sapere come calcolare rapidamente  $C$ . Poiché  $k = 199$  è 11000111 in base 2, calcoliamo in successione i resti ( $\pmod{32899}$ ) di  $M$  elevato a  $2^0, 2^1, \dots, 2^7$ , ricordando le proprietà dell'aritmetica modulare (esistono anche metodi più veloci che usano  $\pmod{p}$  e  $\pmod{q}$ ):

$$\begin{array}{ll} 20299^1 \equiv \mathbf{20299}; & 20299^2 \equiv \mathbf{22325}; \\ 20299^4 \equiv 22325^2 \equiv \mathbf{18674}; & 20299^8 \equiv 18674^2 \equiv 21775; \\ 20299^{16} \equiv 21775^2 \equiv 10237; & 20299^{32} \equiv 10237^2 \equiv 12854; \\ 20299^{64} \equiv 12854^2 \equiv \mathbf{6538}; & 20299^{128} \equiv 6538^2 \equiv \mathbf{9643}. \end{array}$$

Ci interessano i primi tre resti e gli ultimi due (qui sopra evidenziati in neretto), corrispondenti ai bit “1” di  $k$ ; infatti, per le proprietà delle potenze, risulta:

$$\begin{aligned}
 C \equiv 20299^{199} &= 20299^1 \cdot 20299^2 \cdot 20299^4 \cdot 20299^{64} \cdot 20299^{128} \equiv \\
 &20299 \cdot 22325 \cdot 18674 \cdot 6538 \cdot 9643 \equiv 24349 \cdot 18674 \cdot 6538 \cdot 9643 \equiv \\
 &29046 \cdot 6538 \cdot 9643 \equiv 9720 \cdot 9643 \equiv 709.
 \end{aligned}$$

Quindi il mittente invia il messaggio cifrato  $C = 709$ .

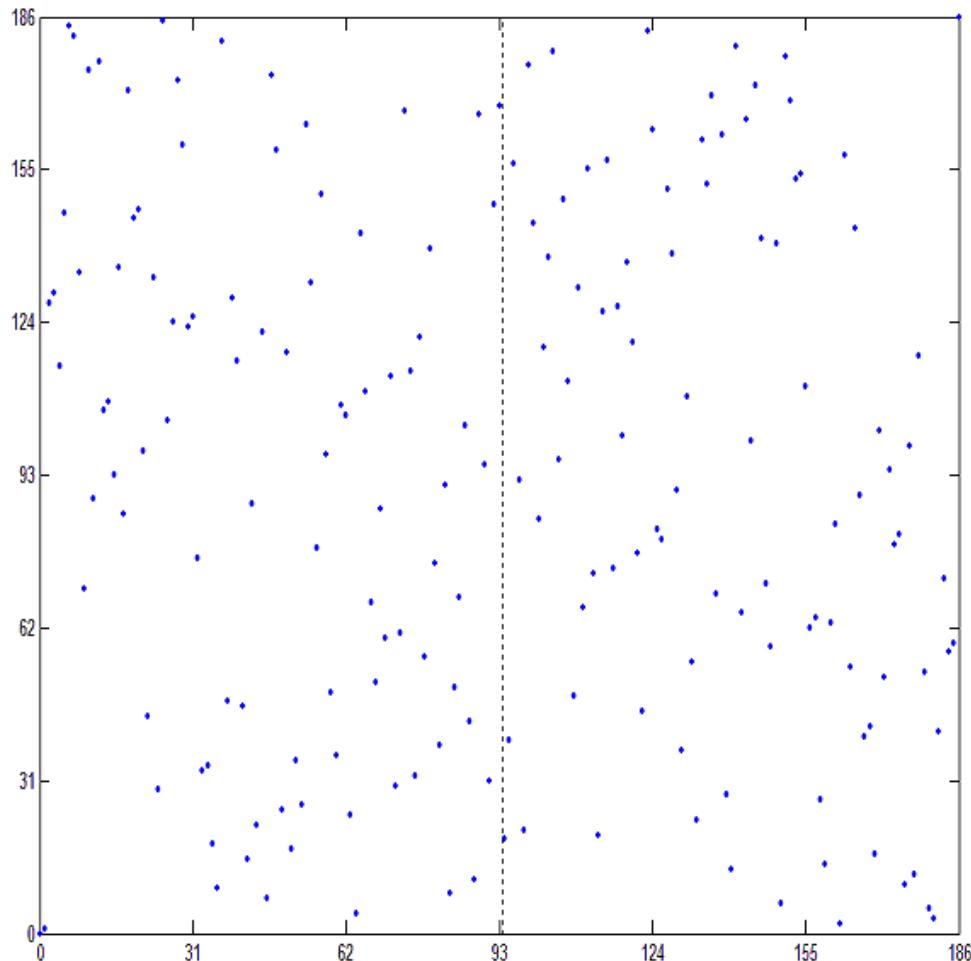
- Per decifrare il messaggio ricevuto  $C$ , il destinatario applica la formula  $M = C^d \bmod N$ .
- *Esempio* (continua):  $M = 709^{327} \bmod 32899$ . Procedendo in maniera analoga, poiché  $d$  è 101000111 in base 2, dobbiamo calcolare i resti di  $C$  elevato a  $2^0, 2^1, \dots, 2^8$  (qui c'è un bit in più):

$$\begin{array}{ll}
 709^1 \equiv \mathbf{709}; & 709^2 \equiv \mathbf{9196}; \\
 709^4 \equiv 9196^2 \equiv \mathbf{15986}; & 709^8 \equiv 15986^2 \equiv 25663; \\
 709^{16} \equiv 25663^2 \equiv 17387; & 709^{32} \equiv 17387^2 \equiv 31757; \\
 709^{64} \equiv 31757^2 \equiv \mathbf{21103}; & 709^{128} \equiv 21103^2 \equiv 15745; \\
 709^{256} \equiv 15745^2 \equiv \mathbf{11060}. &
 \end{array}$$

Ci interessano i resti evidenziati in neretto, corrispondenti ai bit "1" di  $d$ ; e possiamo calcolare:

$$\begin{aligned}
 M \equiv 709 \cdot 9196 \cdot 15986 \cdot 21103 \cdot 11060 &\equiv 5962 \cdot 15986 \cdot 21103 \cdot 11060 \equiv \\
 &129 \cdot 21103 \cdot 11060 \equiv 24569 \cdot 11060 \equiv 20299,
 \end{aligned}$$

che è proprio 0100111101001011, se lo esprimiamo in binario con due byte.



Rivest, Shamir e Adleman hanno individuato una funzione sì invertibile ma, all'atto pratico, soltanto da chi sia in possesso di informazioni privilegiate: i valori di  $p$  e  $q$ , e quindi la chiave  $d$  per decifrare. Tali informazioni sono note soltanto al destinatario e a chi riesce a fattorizzare  $N$ , magari servendosi di qualche procedimento efficiente finora sconosciuto...

Le procedure di cifratura e decifrazione, che in realtà coinvolgono messaggi piuttosto lunghi e manipolano numeri rappresentati da diverse migliaia di bit (a seconda del livello di sicurezza che si vuole raggiungere), devono essere svolte ovviamente da un computer, mediante apposito software, se non anche hardware dedicato.

Non soltanto RSA, ma pure altri algoritmi crittografici si basano sull'impossibilità di fattorizzare grandi numeri (o risolvere diversi problemi numerici, assai complicati) in tempi ragionevoli. Tutti sono usati nei sistemi di autenticazione, nelle cifrature di firme digitali, nei protocolli per comunicazioni e transazioni commerciali.<sup>18</sup> Data la relativa facilità con cui si possono produrre grandi numeri primi (sebbene più grandi siano i numeri usati, più costoso diventi il funzionamento dell'intero meccanismo), il solo punto debole di sistemi come RSA è che in futuro potrebbe essere disponibile un metodo efficiente di fattorizzazione (o forse è già stato scoperto e tenuto segretissimo?!), che li renderebbe non più utilizzabili. Altrimenti, pur di impiegare numeri sufficientemente grandi, dovrebbero rimanere ancora affidabili nel prossimo futuro, nonostante la continua crescita delle prestazioni dei computer. Ci riferiamo ai computer tradizionali: un'ulteriore seria minaccia è costituita dall'eventuale realizzazione di un *computer quantistico* con un numero abbastanza grande di *qubit* (dove un *qubit* ha quattro possibili stati, memorizzabili e manipolabili simultaneamente). Infatti, già nel 1994, Peter Shor, docente di matematica al MIT, scoprì un algoritmo di fattorizzazione tempo-polinomiale che può essere eseguito da un computer quantistico. (Tuttavia, non vi sono ancora indizi circa la possibilità di risolvere efficientemente con un computer quantistico un qualsiasi problema NP-arduo.)

Le più recenti ricerche riguardano la “crittografia omomorfica”, volta a garantire la sicurezza dei dati nella cosiddetta “nuvola informatica” (*cloud computing*) anche durante la loro elaborazione, permettendo l'esecuzione di (determinate) operazioni su di essi *senza decifrarli*; per la piena realizzazione di questo obiettivo sono allo studio anche nuovi processori dedicati.

Un altro tema interessante sono i sistemi di crittografia in cui la chiave privata è in qualche modo ripartita tra più soggetti, e soltanto l'azione combinata di almeno una certa parte di essi, mediante opportuni protocolli, permette cifrature e decifrazioni.

Tra i tanti quesiti sulla crittografia proposti nelle gare *Kangourou dell'Informatica*, ricordiamo “Messaggio cifrato” e “Crittografia”, pubblicati nel libretto del 2015.

**Parole chiave:** complessità esponenziale e pseudo-polinomiale, problema decisionale e certificazione della risposta, problemi NP-completi e NP-equivalenti, ottimizzazione combinatoria, tecniche di *branch-and-bound* e di programmazione dinamica, algoritmi di approssimazione, grafi e alberi ricoprenti, test di primalità, numeri di Mersenne, ricerca dei fattori primi, sistemi di crittografia a chiave asimmetrica.

---

<sup>18</sup> Così come diversi altri crittosistemi realizzati dopo RSA: la sicurezza di alcuni di essi è affidata a problemi matematici che, neppure disponendo di computer quantistici, si saprebbe come risolvere efficientemente. Certuni sono a chiave pubblica, ma in grado di resistere agli attacchi dei computer quantistici, e utilizzabili su computer tradizionali, cioè deterministic (*post-quantum cryptography*). La cosiddetta *crittografia quantistica*, studiata dagli anni '80 e sperimentata dal 2004, permetterebbe invece, quantomeno, lo scambio sicuro delle chiavi, poiché ne rivelerebbe l'eventuale intercettazione da parte di terzi.

## 5. Tappezzerie, decorazioni e altre cose ancora

Questo capitolo è dedicato ad alcuni fra i tanti istruttivi “divertimenti grafici” che si possono realizzare con il computer. Gli aspetti matematici e algoritmici delle questioni che tratteremo sono piuttosto semplici da comprendere, e pertanto crediamo che il lettore riuscirà senza eccessive difficoltà a scrivere i relativi programmi, in uno dei linguaggi a lui noti, e a provare in prima persona il piacere estetico procurato dalla creazione di immagini simili a quelle costruite dai programmi qui presentati: noi abbiamo provato a scriverli, e una scelta delle immagini ottenute è riportata nelle tavole a colori fuori testo, alla fine del volume.

### Tappezzeria per la mente.

Nel numero di settembre 1986 della rivista *Scientific American* – pubblicata in Italia come *Le Scienze* – Alexander K. Dewdney, il quale da oltre due anni vi curava la rubrica *Computer Recreations*, scrisse un interessante articolo, apparso in italiano due mesi più tardi col titolo “Tappezzeria per la mente: immagini al calcolatore quasi, ma non del tutto, ripetitive”.

La prima delle idee ivi proposte si rifà alle curve di livello disegnate sulle carte geografiche usate, ad esempio, dagli alpinisti. Una funzione di due variabili sufficientemente regolare è rappresentata da una superficie nello spazio tridimensionale, proprio come il suolo di un territorio montano, o meglio collinare. Se proiettiamo i punti di questa superficie sul piano orizzontale, colorandoli in funzione della loro quota, otteniamo un’immagine bidimensionale a colori.

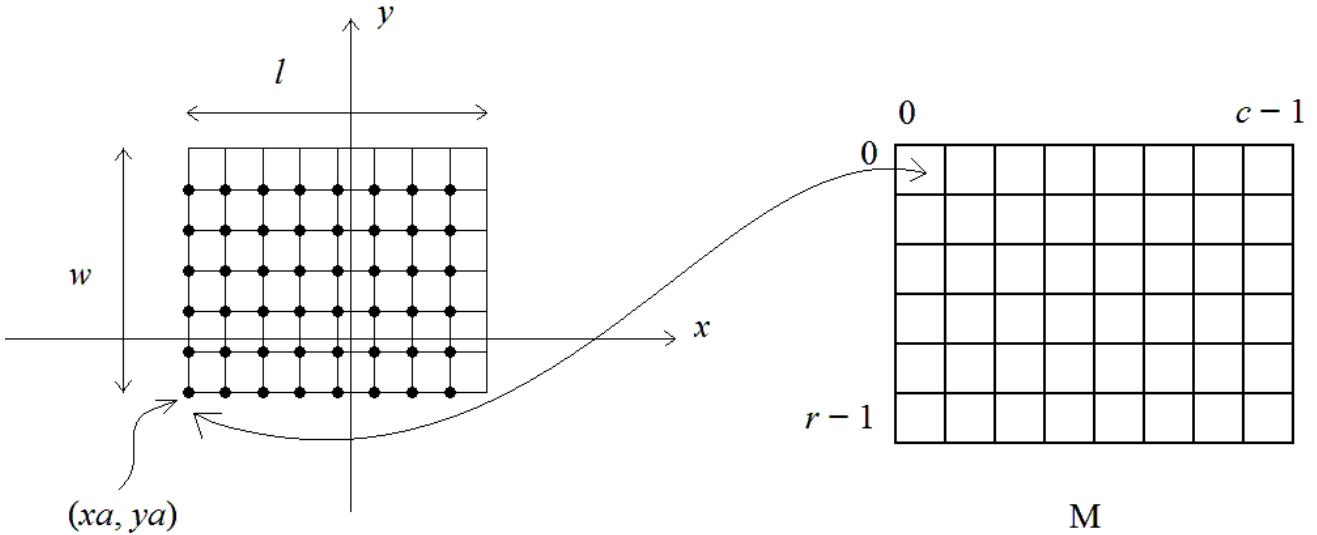
Com’è noto, un’immagine digitale è formata da una matrice di *pixel* (da *picture element*), cioè quadratini di colore. Supponiamo di operare su una matrice  $M$  di  $r$  righe per  $c$  colonne; formati *standard* sono  $480 \times 640$  e  $768 \times 1024$ , rettangoli con altezza pari ai  $3/4$  della base. Di solito un pixel è rappresentato da una tripla di *byte* – uno per il rosso, uno per il verde e uno per il blu – sicché per ognuno dei tre colori di base possono essere usati  $2^8 = 256$  livelli; ne consegue che i colori disponibili sono

$$2^8 \cdot 2^8 \cdot 2^8 = 2^{24} = 2^4 \cdot 2^{20} = 16 \text{ mebi}$$

( $1 \text{ mebi} = 2^{20}$  è un po’ più grande di  $1 \text{ mega} = 10^6$ ; tuttavia, c’è chi dice ancora “mega” intendendo in realtà “mebi”). E se un pixel è costituito da 3 byte, le matrici dei suddetti formati occupano 900 kibibyte e 2.25 mebibyte, rispettivamente.

In generale, dunque, dovremo costruire una griglia rettangolare di  $r \times c$  punti, equispaziati in verticale ed equispaziati in orizzontale, e “posarla” in una regione significativa del piano ( $x, y$ ); in corrispondenza di tali punti calcoleremo poi la funzione  $F$  di due variabili che abbiamo scelto, memorizzando questi campioni – tradotti in “colore” – nella matrice  $M$ .

Il procedimento è illustrato nella figura che segue.



I punti della griglia hanno coordinate cartesiane

$$\begin{aligned}x_j &= xa + l \cdot j / c & j &= 0, \dots, c - 1 \\y_i &= ya + w \cdot i / r & i &= 0, \dots, r - 1\end{aligned}$$

e, per avere una scala monometrica,  $w$  deve essere i  $3/4$  di  $l$ , se la matrice  $M$  è di uno dei suddetti formati.

Preparata una “tavolozza” di  $n$  colori, si calcolano i valori

$$F_{ji} = F(x_j, y_i)$$

e si suddivide il *range* in cui essi cadono in  $n$  intervalli, ciascuno dei quali è messo in corrispondenza con un colore; quindi a  $M(i, j)$ , ossia all’elemento di riga  $i$  e colonna  $j$  della matrice di pixel, si assegna il colore corrispondente all’intervallo in cui cade  $F_{ji}$ .

Come più semplice alternativa – che è quella adottata nell’articolo, al fine di ottenere immagini “quasi ripetitive” – per associare uno dei colori a  $F_{ji}$ , lo si approssima o tronca all’intero e si calcola il resto della sua divisione per  $n$ , ottenendo così un intero compreso tra 0 e  $n - 1$ . In tal modo, però, tutti i punti la cui quota differisca per un multiplo di  $n$  sono associati allo stesso colore.

Alla fine, per avere l’asse  $y$  orientato dal basso verso l’alto sullo schermo del computer, bisognerà comunque visualizzare la matrice  $M$  ribaltandola rispetto all’asse orizzontale.

In breve, nella regione del piano cartesiano considerata, la quota della superficie  $F$  è tradotta – in un modo o nell’altro – in colore. Più che la scelta dei colori, i parametri importanti di questo processo sono proprio quelli che determinano tale regione rettangolare, e cioè le coordinate del vertice in basso a sinistra  $(xa, ya)$ , la lunghezza  $l$  e la larghezza  $w$ , che ovviamente dovranno essere fissati tenendo conto della particolare funzione  $F$  da campionare.

Come esempio, nell’articolo di Dewdney è considerata la funzione:

$$F(x, y) = x^2 + y^2$$

proposta da John E. Connett dell'Università del Minnesota. Noi ne abbiamo provate tante altre, alcune delle quali sono di seguito elencate; si tratta di composizioni o combinazioni di funzioni, nella maggior parte dei casi continue e piuttosto “lisce” (*smooth*) almeno in certe regioni:

$$\begin{aligned} &x \cdot (4y - x) \\ &3 \cdot \sin(x^2 + y^2) \\ &\sin(x) \cdot \sin(y) - \tan(y \cdot x) \\ &\sin(x + y) + \cos(x + y) + \cos(y - x) - \sin(y) \\ &\sin(y - x) + \cos(x + y) + \cos(x) - \sin(y) \\ &\sin(x) - \cos(y) - \tan(x + y) + \sin(y) \cdot \tan(y - x) \\ &\sqrt{|y - x|} + \sin(y - x) - 2 \cdot \ln|x + y| \\ &\sqrt{|y \cdot x|} - \ln|y/x| + \sin(y - x) \cdot \ln|x + y| \\ &\sqrt{|y \cdot x|} - \ln|y/x| + \cos(y - x) - \sin(x + y) \\ &\sqrt{|y \cdot x|} - \ln|y/x| + \cos(y - x) - \ln|x + y| \end{aligned}$$

Lasciamo a voi il compito di cercare le porzioni di piano più interessanti per tali funzioni o altre ancora: diverse composizioni di funzioni logaritmiche, esponenziali, trigonometriche eccetera, a vostra scelta (un consiglio: mantenendo  $r$  e  $c$  fissati, provate con  $l$  e  $w$  sia piuttosto “piccoli” sia più “grandi”). Siamo così certi di lasciarvi pure il piacere di scoprire nuove immagini, che potrete confrontare con quelle riportate nella prima sezione delle tavole a colori, alla fine del volume, alcune delle quali sono state prodotte coi programmi realizzati dai miei allievi.

Curiosamente, proprio il mese successivo all'uscita dell'articolo di Dewdney, Brian Hayes – che aveva curato la rubrica su *Le Scienze* prima di lui – pubblicò un approfondimento sullo stesso tema: *On the bathtub algorithm for dot-matrix holograms* (Computer Language, Vol. 3, No. 10, ottobre 1986, pp. 21-32), disponibile all'indirizzo <http://bit-player.org/bph-publications/CompLang-1986-10-Hayes-holograms.pdf>.

Un altro più recente contributo che merita di essere letto e apprezzato, anche per le sorprendenti immagini che contiene, è stato scritto da Craig S. Kaplan della Università di Waterloo, Canada: *Aliasing Artifacts and Accidental Algorithmic Art*. Pubblicato negli atti della conferenza *2005 Renaissance Banff: Mathematical Connections in Art, Music and Science*, pp. 349-356, è pur'esso reperibile in rete: [http://www.cgl.uwaterloo.ca/~csk/papers/kaplan\\_bridges2005a.pdf](http://www.cgl.uwaterloo.ca/~csk/papers/kaplan_bridges2005a.pdf).

In quest'ultimo articolo si fa altresì riferimento ai “diagrammi di Voronoi”.

Nella sua versione più semplice, un diagramma di Voronoi non è altro che una partizione del piano, ottenuta a partire da un insieme finito di punti, detti “generatori”, a ciascuno dei quali è associato un colore; ciascun punto del piano è quindi colorato con il colore del generatore ad esso più vicino, secondo l'usuale distanza euclidea.

Si costruiscono così immagini spettacolari, talvolta simili a quelle prodotte da un caleidoscopio. Kaplan ne aveva già parlato in un suo precedente lavoro: *Voronoi Diagrams and Ornamental Design*, che si può trovare al seguente indirizzo:  
[http://www.cgl.uwaterloo.ca/~csk/papers/kaplan\\_isama1999.pdf](http://www.cgl.uwaterloo.ca/~csk/papers/kaplan_isama1999.pdf).

## Decorazioni per uova Fabergé.

Trascorso un anno dalle pubblicazioni di Dewdney e Hayes, Clifford A. Pickover ebbe un’idea leggermente diversa, che gli permise di creare dei motivi simili a quelli – assai elaborati e in qualche modo simmetrici – che ornano le famose “uova Fabergé”.

La tradizione di regalare uova pasquali decorate fu inaugurata in Russia nel 1885 dallo zar Alessandro III, che da quell’anno commissionò un uovo al suo gioielliere di fiducia, Peter Carl Fabergé, per donarlo alla moglie in occasione della Pasqua. Questa usanza fu continuata dal figlio, Nicola II, che gli succedette nel 1894; anzi, da allora il dono fu raddoppiato: un uovo alla moglie e uno alla zarina madre. Fino alla caduta dell’impero russo, nel 1917, furono prodotti una cinquantina di questi veri e propri capolavori dell’arte orafa: ogni uovo, fatto d’oro, pietre preziose e altri materiali pregiati, era decorato con miniature e smalti che riproducevano splendidi motivi ornamentali, e richiedeva circa un anno di lavoro a un abile maestro orafo.

Pickover usò come funzione la somma di due sinusoidi, una dipendente da  $x$  e l’altra da  $y$ , con stessa frequenza ma diverse fasi; tenne però fissa la griglia di punti sul piano cartesiano, facendo corrispondere direttamente i valori di  $j$  e  $i$  alle loro ascisse e ordinate, rispettivamente.

Per costruire i campioni di *una* di queste funzioni, procediamo così:

- prendiamo tre numeri “reali” (in virgola mobile, per intenderci del tipo **float** o **double** del linguaggio C):  $\omega$  (la pulsazione) scelto “a caso” nell’intervallo  $[0.15, 0.80]$ ,  $\varphi$  e  $\psi$  (le fasi) scelti, ancora casualmente, nell’intervallo  $[0.0, 1.0]$  (così suggerisce Pickover, ma forse sarebbe più efficace  $[0, 2\pi]$ );
- per ogni  $i$  e  $j$ , calcoliamo  $F_{ji} = \sin(\omega \cdot i + \varphi) + \sin(\omega \cdot j + \psi)$ .

A questo punto scegliamo a caso un altro numero reale  $\mu$  nell’intervallo  $[0.0, 20.0]$  e un numero intero  $\beta$  tra 128 e 256 (quest’ultimo intervallo è un nostro suggerimento) e infine calcoliamo

$$m(i, j) = \text{floor}((\beta + \mu) \cdot (1.0 + F_{ji}/2.0)) \bmod \beta$$

dove *floor* è la funzione che restituisce il massimo intero minore o uguale all’argomento (nel nostro caso comunque non negativo) e *mod* (“modulo”) il resto della divisione intera tra i suoi due operandi.

Otteniamo così una matrice  $m(i, j)$  di interi in  $[0, \beta - 1]$ ; poi ripetiamo questi passi altre due volte, per avere *tre* di queste matrici da usare come componenti per i tre colori di base dell’immagine finale.

In fondo al volume, nella seconda sezione delle tavole a colori, sono riportate alcune immagini ottenute col nostro programma.

Pickover non si è fermato qui: ha scritto un programma in C che impiega le librerie OpenGL per la grafica in tre dimensioni, in grado di presentare all'utente uno splendido uovo pasquale, rifinito di tutto punto, all'incirca ogni tre secondi. Naturalmente, per ottenere una maggiore varietà di motivi, ha provato a usare altre funzioni, come le sinusoidi la cui frequenza abbia a sua volta un andamento sinusoidale, combinate in vari modi.

E ha scritto pure che non si vedranno mai due uova uguali, ammirando la varietà delle forme che passano sullo schermo: semplicemente “girando una manopola” che controlla i diversi parametri, si genera infatti una varietà pressoché infinita di disegni affascinanti, e per di più questo risultato è raggiunto in modo relativamente semplice dal punto di vista computazionale. Questa è la ragione per cui i nostri progetti di grafica al computer potrebbero suscitare l'interesse di chi allestisce esposizioni didattiche o mostre presso musei, destinate sia ai bambini sia agli adulti.

L'articolo originale dell'autore, *A recipe for self-decorating eggs*, fu pubblicato su Computer Language, Vol. 4, No. 11, novembre 1987, pp. 55-58. Un breve resoconto si trova nella sezione speciale “Art and Biology” della rivista Leonardo, Vol. 31, No. 4, agosto-settembre 1998, p. 280: *Algorithmic Fabergé Eggs via Residue Analysis*.

### **Immagini ottenute con formule ricorrenti.**

Ritorniamo all'articolo di Dewdney. La seconda proposta che avanzava consisteva nella costruzione di immagini mediante formule ricorrenti; in particolare, riportava un algoritmo di Barry Martin della Aston University di Birmingham, ispirato dal procedimento iterativo per generare quelle meravigliose illustrazioni a colori che rappresentano approssimazioni dell'insieme di Mandelbrot. Per inciso, ormai da parecchi anni queste figure sono oggetto ricorrente di divulgazione, purtroppo non sempre chiara, malgrado la semplicità dell'idea che permette di ottenerle: vi accenneremo alla fine del capitolo.

L'algoritmo, che Martin chiamò *Hopalong*, consiste in questi passi:

1. dati:

- i valori di tre parametri reali ( $a$ ,  $b$  e  $c$ )
- il numero  $N$  di punti che si vogliono disegnare
- il colore iniziale da assegnare al piano cartesiano (tutto nero oppure tutto bianco)
- il punto di partenza (ad esempio l'origine), le cui coordinate sono assegnate rispettivamente alle variabili  $x$  e  $y$

2. ripete per  $N$  volte:

- disegna il punto  $(x, y)$  (in bianco oppure in nero, o in un altro opportuno colore)

- calcola  $nuovo\_x = y - segno(x) \cdot \sqrt{|b \cdot x - c|}$
- calcola  $nuovo\_y = a - x$
- assegna a  $x$  il valore di  $nuovo\_x$
- assegna a  $y$  il valore di  $nuovo\_y$

Una variante dell'algoritmo calcola come  $nuovo\_x$  il valore  $y - segno(x) \cdot |b \cdot x - c|$ , senza la radice quadrata, ed è quella da noi usata; un'altra, più semplice, il valore  $y - segno(x)$ . In quest'ultimo caso i parametri si riducono al solo  $a$ , al quale Martin consiglia di assegnare valori prossimi a  $\pi$ .

Ovviamente la funzione  $segno$  vale  $-1$  se l'argomento è negativo,  $+1$  altrimenti.

Per ottenere un'immagine a colori, si può decidere ad esempio di cambiare colore ogni  $n$  punti disegnati, oppure di ripetere una stessa sequenza di  $n$  colori ogni  $n$  punti disegnati.

Si può pensare di prendere anche qui una porzione di piano rettangolare da visualizzare, purché sia sufficientemente densa di punti disegnati, mappandola nella matrice di pixel; oppure, servendosi ad esempio dell'operazione “modulo”, spostare all'interno della regione da visualizzare i punti disegnati fuori da essa, o comprimere in qualche altro modo nella regione stessa quelle zone pur interessate dal processo che ne stanno al di fuori.

Anche con questo algoritmo abbiamo ottenuto delle immagini interessanti, come si può vedere nelle quattro figure su sfondo nero, riportate nella terza sezione delle tavole a colori in fondo al volume. Tanto per dare un'idea, esse sono state generate, nell'ordine, con i seguenti parametri:

$a$	$b$	$c$	$N$
-75	0.02	5	$10^6$
-100	0.9	5	75000
-10	-0.1	-5	$10^7$
-200	0.9	40	50000

Nell'ultima parte dell'articolo citato, Dewdney parla di disegni ottenuti mediante un semplice automa cellulare in grado di replicare sé stesso, inventato nel 1960 da Edward Fredkin del MIT, e di sue varianti più complesse; noi ci occuperemo di “automi cellulari” nel prossimo capitolo.

### Tappeti persiani “ricorsivi”.

I motivi usati per decorare pavimenti, ceramiche e stoffe – tanto per fare soltanto alcuni esempi – che magari provengono da epoche e civiltà lontane – si pensi allo stile moresco o a quello persiano – spesso seguono schemi in qualche modo simmetrici (o quasi) e ripetitivi (o quasi).

Uno straordinario volume che illustra il ruolo delle costruzioni geometriche nell'arte

islamica fu scritto parecchi anni fa da Issam El-Said e Ayse Parman: *Geometric Concepts in Islamic Art*, World of Islam Festival Publishing Company (Londra, 1976).

In tempi più recenti, Anne M. Burns, del Dipartimento di Matematica della Università di Long Island, New York, ha escogitato un semplice algoritmo ricorsivo per generare automaticamente motivi che ricordano assai vivamente quelli dei pregiati tappeti persiani. L'idea è contenuta in un suo articolo, intitolato appunto “*Persian*” Recursion, apparso nel numero di giugno del 1997 della rivista Mathematics Magazine (Vol. 70, No. 3, pp. 196-199); noi l'abbiamo seguita per scrivere il programma che adesso delineiamo, grazie al quale abbiamo ricavato le suggestive immagini contenute nell'ultima sezione in fondo al volume.

Partiamo con una matrice di pixel quadrata, con  $r = c = 2^d + 1$ : ad esempio, scegliendo  $d = 9$ , i pixel saranno  $513 \times 513$ . Assegniamo un colore a nostra scelta ai soli quattro pixel agli angoli, e poi inneschiamo il procedimento ricorsivo sull'intera matrice: in funzione dei colori agli angoli, calcoliamo un nuovo colore e lo assegniamo a tutti i pixel della riga e della colonna centrali; dopodiché ripetiamo lo stesso procedimento su ciascuno dei quattro quadranti.

Per far sì che ogni applicazione della procedura si trovi ad operare su una porzione quadrata della matrice, dovremo avere l'accortezza di comprendere in ciascun quadrante anche mezza riga e mezza colonna centrali, fino al pixel centrale incluso. Come terminerà il processo? Naturalmente, quando la porzione quadrata sarà di soli  $2 \times 2$  pixel, tutti e quattro avranno già il loro colore assegnato, e quindi non si dovrà continuare!

Alla fine, il disegno a colori risulterà simmetrico? Non proprio: la presenza di una qualche simmetria dipenderà dall'ordine in cui sono elaborati ricorsivamente i quattro quadranti...

Per chi vuol scendere nei dettagli, ecco le definizioni che possono servire, espresse in linguaggio C++:

```
#define size 513
    // lato della matrice quadrata, espresso in pixel

struct pixel { // un pixel è costituito da 3 byte:
    unsigned char r, g, b;
    // livelli dei colori di base (red, green, blue)
};

typedef pixel pixel_matrix[size][size];
    // Il nome "pixel_matrix" è attribuito agli array (statici)
    // di pixel, con due indici, entrambi da 0 a size-1.

void draw (pixel_matrix M, int i1, int i2, int j1, int j2) {
    // Colora la parte di matrice dalla riga di indice i1 a
    // quella di indice i2 e dalla colonna di indice j1 a quella
    // di indice j2, ricorsivamente. Precondizione:
    // i2-i1 == j2-j1 == 2^k per un qualche intero positivo k.
```

```

if (i2-i1 > 1) {
    // il lato della parte da colorare è di almeno 3 pixel
    pixel c = new_colour(M[i1][j1], M[i1][j2],
                          M[i2][j1], M[i2][j2]);
    // L'applicazione della funzione new_colour (da definire)
    // sceglie (e restituisce) il colore per la riga e la
    // colonna centrali, in base ai quattro pixel agli angoli;
    // tale colore è memorizzato nel pixel c ...
    int i, j, im = (i1+i2)/2, jm = (j1+j2)/2;
    for (i = i1; i <= i2; i++) M[i][jm] = c;
    for (j = j1; j <= j2; j++) M[im][j] = c;
    // Colora ricorsivamente le 4 suddivisioni di questa parte
    // di matrice:
    draw(M, i1, im, j1, jm); // riquadro in alto a sinistra
    draw(M, im, i2, jm, j2); // riquadro in basso a destra
    draw(M, i1, im, jm, j2); // riquadro in alto a destra
    draw(M, im, i2, j1, jm); // riquadro in basso a sinistra
    // Tale sequenza di chiamate garantisce comunque la simmetria
    // del risultato finale rispetto alle due diagonali.
}
}

```

Per quanto concerne la scelta del colore da attribuire alla riga e alla colonna centrali, noi abbiamo fatto così:

```

#define ratio 3
    // divisore della somma dei colori dei 4 pixel d'angolo
#define shift 64
    // termine poi sommato (modulo 256) per ottenere il colore

pixel new_colour (pixel p1, pixel p2, pixel p3, pixel p4) {
    // In base ai quattro pixel dati, calcola un nuovo colore.
    pixel c;
    c.r = ((p1.r + p2.r + p3.r + p4.r)/ratio + shift) % 256;
    c.g = ((p1.g + p2.g + p3.g + p4.g)/ratio + shift) % 256;
    c.b = ((p1.b + p2.b + p3.b + p4.b)/ratio + shift) % 256;
    return c;
}

```

A questo punto, per completare l'opera, basta scrivere la funzione principale, che dovrà allocare un *data object* di tipo `pixel_matrix`, chiamiamolo ancora `M`; quindi assegnare un colore iniziale ai quattro pixel d'angolo e innescare il procedimento ricorsivo con la chiamata

```

draw(M, 0, size-1, 0, size-1);
// disegna tutto il tappeto, ricorsivamente

```

Dopodiché non resterà che visualizzare in qualche modo il contenuto di `M` e/o salvarlo in un *file*.

Se la costante `size` è fissata a 513, scesi 8 livelli di chiamate si arriva a un quadrato

di  $3 \times 3$  pixel; in funzione dei colori agli angoli si assegna un colore (unico) agli altri cinque pixel, dopodiché ciascuna delle quattro chiamate successive (di livello 9) opererà su un quadrato di  $2 \times 2$  pixel, e pertanto non farà più nulla.

Giocando sul colore iniziale e sulle costanti `ratio` e `shift`, si può già ottenere una buona varietà di immagini.

Suggeriamo tuttavia un’alternativa: prima di effettuare la chiamata esterna alla procedura `draw`, si assegna uno stesso colore a tutti i pixel sul bordo della matrice, anziché soltanto ai quattro d’angolo, e si riscrivono i due cicli `for` nel corpo della procedura stessa in questo modo:

```
for (i = i1+1; i < i2; i++) M[i][jm] = c;  
for (j = j1+1; j < j2; j++) M[im][j] = c;
```

Così facendo, si escludono ogni volta dalla nuova colorazione i quattro pixel alle estremità della croce centrale, e quindi il risultato finale sarà perfettamente simmetrico rispetto sia alle due diagonali, sia agli assi orizzontale e verticale.

Naturalmente, ciascuno può apportare modifiche di propria invenzione alla funzione `new_colour` o addirittura all’algoritmo di base che abbiamo illustrato.

In verità, uno degli scopi di questo capitolo è la trascrizione scrupolosa di alcune (buone) “ricette” per creare immagini di sicuro effetto; l’estro dell’autentico cuoco deve poi intervenire, per dare un tocco di originalità e personalità al piatto da servire in tavola! E se ci siamo limitati alle due dimensioni, pazienza: anziché uova pasquali decorate, avremo comunque motivi da disegnare sulla carta per confezionarle o sulla tovaglia da stendere quando le mangeremo!

## Montagne “frattali”.

Torniamo di nuovo indietro nel tempo, agli anni ’80, quando si cominciò seriamente a pensare a lungometraggi generati dal computer. Uno dei primi algoritmi realizzati serviva a generare catene montuose “a prova di zoom”, cioè che mantenessero le loro caratteristiche (frastagliature, vette, picchi e avvallamenti) all’avvicinarsi anche improvviso della telecamera.

In verità, il modello cosiddetto “frattale” della natura implicherebbe un regresso *infinito* di dettaglio: soltanto così, infatti, si possono definire oggetti la cui dimensione non è intera. Per quanto riguarda le applicazioni grafiche al computer, è sufficiente che il paesaggio appaia dettagliato fino al massimo ingrandimento che si vuole raggiungere.

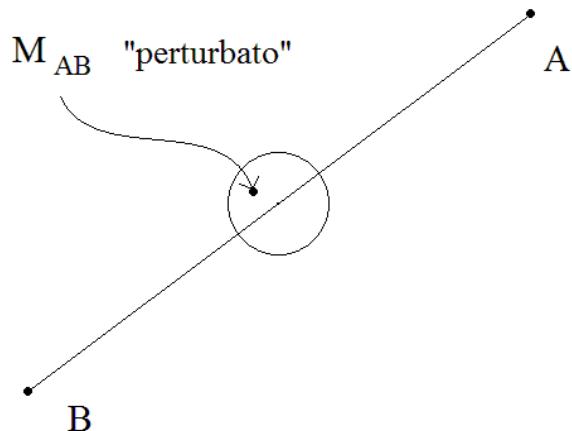
Per semplicità, vediamo come si può ottenere un’immagine in due sole dimensioni – una sorta di fotografia dal basso – di una montagna artificiale. Il procedimento ricorsivo non è poi tanto dissimile da quello che produce i tappeti persiani. Qui però si parte da tre punti, che rappresentano la vetta e gli estremi della base del nostro monte, e si stabilisce il livello massimo delle chiamate ricorsive da effettuare (che è strettamente legato al livello di dettaglio da raggiungere, per cui occorre fare

attenzione a che non venga poi superato il limite di risoluzione dello schermo): e questi saranno gli argomenti della chiamata esterna.

La procedura opera quindi su quattro dati: tre punti del piano cartesiano (A, B, C) e livello (L).

Se L vale zero, allora essa si limita a tracciare i lati del triangolo ABC; altrimenti:

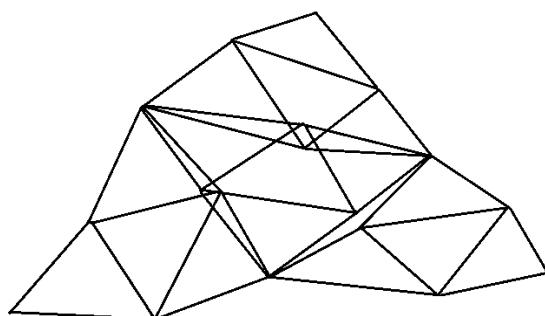
- calcola in  $M_{AB}$  il punto medio del segmento AB e poi lo “perturba”, ossia lo sposta a caso all’interno di un cerchio (oppure un quadrato) centrato in esso e con raggio pari a una (modesta) percentuale della lunghezza del segmento AB:



(in alternativa, la perturbazione può essere soltanto verticale, cioè riguardare l’ordinata ma non l’ascissa del punto medio);

- ripete la stessa operazione per i segmenti AC e BC, ottenendo i punti medi perturbati  $M_{AC}$  e  $M_{BC}$ ;
- ricorre (cioè chiama sé stessa) *quattro* volte, rispettivamente sulle terne di punti  $(A, M_{AB}, M_{AC})$ ,  $(M_{AB}, B, M_{BC})$ ,  $(M_{AC}, M_{BC}, C)$ ,  $(M_{AB}, M_{AC}, M_{BC})$ , e tutte le volte con livello  $L - 1$ .

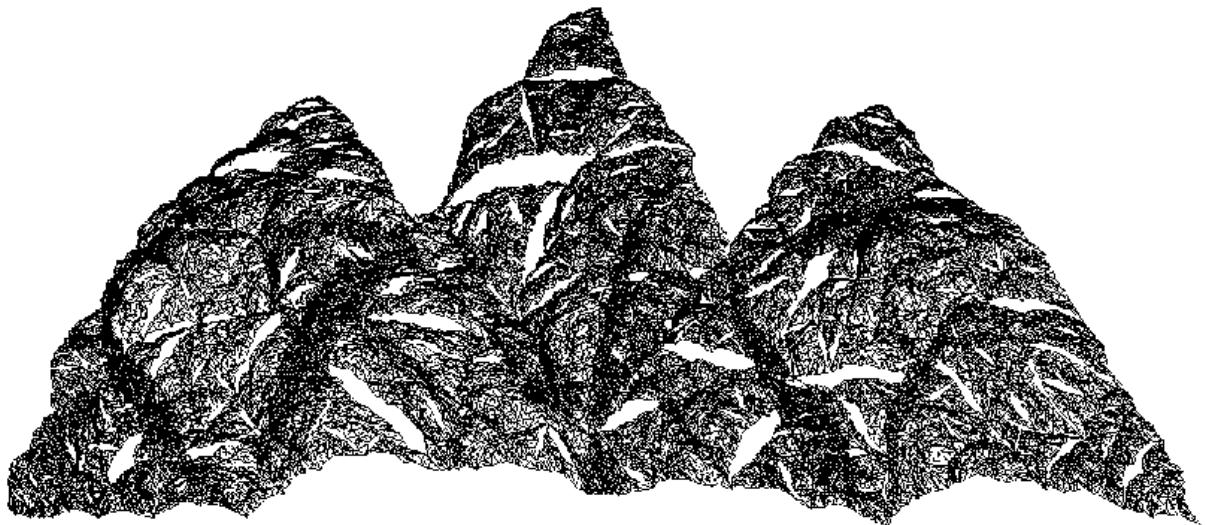
Ad esempio, fissando a 2 il livello massimo, si ottiene un risultato di questo tipo:



Si noti che in questa figura i triangoli disegnati sono 16. In generale saranno 4 elevato al livello massimo: aumentando tale livello, ci possiamo aspettare una figura assai più fine, come la seguente, costruita a livello 7, e dunque formata da 16384 triangoli.



L’algoritmo può essere adattato con facilità alla generazione di gruppi o intere catene montuose!



Un analogo procedimento può essere poi formulato per costruire montagne in tre dimensioni. Ad esempio, presi come dati iniziali un quadrato (la base del monte) e le quote in corrispondenza dei quattro vertici, la procedura ricorsiva compie le seguenti azioni:

- suddivide il quadrato in quattro quadranti;
- calcola la quota in corrispondenza del punto centrale, anche semplicemente come media – opportunamente “perturbata” – delle quote ai quattro vertici;
- se è giunta al livello finale, allora disegna quattro triangoli, tra la quota centrale e le quote agli estremi di ciascun lato del quadrato (oppure compie una diversa triangolazione tra le tante possibili);

- altrimenti calcola le quote in corrispondenza dei punti mediani dei lati e ricorre su ciascuno dei quattro quadranti. Nel punto a metà di un lato del quadrato si può assumere una quota pari a una funzione *prefissata* e *commutativa* delle quote ai due estremi del lato stesso; la loro media aritmetica, però, non va bene... ed è facile comprenderne la ragione.

Ci sarà infine bisogno di un processo di resa grafica (*rendering*) tridimensionale, non solo per visualizzare l'immagine ma anche per migliorare la qualità della rappresentazione.

Lasciamo al lettore la realizzazione di questi algoritmi o di loro varianti, raccomandandogli pure – se gli sarà possibile – di confrontare le nostre spiegazioni con quelle contenute nell'articolo di Dewdney “Montagne frattali, piante graftali e altra grafica al calcolatore della Pixar”, apparso su *Le Scienze* (edizione italiana) nel febbraio 1987. Come si evince dal titolo, questo articolo contiene spunti per altri esperimenti grafici assai interessanti!

## Sistemi dinamici “caotici”.

Dall'avvento del computer, la matematica è sempre più “applicata”, e anche “costruita”, con l'aiuto di questo prodigioso strumento. Ma in che cosa davvero ci aiuta il computer? Essenzialmente, in ogni attività che richiede l'analisi, il calcolo, l'elaborazione e la memorizzazione (permanente) di grandi quantità di dati (numerici o simbolici), il tutto in tempi accettabili. Proprio in questo senso, un apporto fondamentale è stato dato dal computer allo studio di quei *sistemi dinamici* che, pur sotto semplici apparenze, nascondono spesso una notevole complessità.

Sempre negli anni '80, l'evoluzione degli elaboratori elettronici – e in particolare delle loro potenzialità grafiche – segnò l'avvio di tecniche matematiche innovative nell'ambito di un filone di ricerca sperimentale e d'avanguardia: quello della *dinamica caotica*. Sebbene gran parte della matematica coinvolta in questo settore sia ardua e astratta almeno quanto quella in altri campi di ricerca, la bellezza intrinseca delle strutture che ne derivano può essere mostrata attraverso immagini spettacolari, prodotte direttamente su uno schermo da un calcolatore; sicché molte idee sono state perfino “rubate” alla matematica e all'ingegneria per essere usate nei film di fantascienza o di animazione!

Ma andiamo in ordine cronologico. Verso la metà dell'Ottocento, il matematico belga Pierre François Verhulst formulò un modello di sviluppo di una *popolazione con tasso di crescita variabile*. Se indichiamo con  $x_0$  la quantità iniziale di popolazione e con  $x_n$  la quantità di popolazione dopo  $n$  anni, allora il tasso di crescita durante l'anno  $(n+1)$ -esimo è

$$r = \frac{x_{n+1} - x_n}{x_n}$$

Se  $r$  si mantiene *costante* anno dopo anno, allora questa equazione è valida per ogni  $n$ , e quindi si ricava una *legge dinamica lineare*:

$$x_{n+1} = (1+r)x_n$$

(lineare in quanto  $x_{n+1}$  è proporzionale a  $x_n$ , qui moltiplicata per la costante  $1+r$ ). Ne deriva una crescita *esponenziale*, tipica di molti fenomeni reali (ma per un tempo limitato!); infatti:

$$\begin{aligned} x_1 &= (1+r)x_0 \\ x_2 &= (1+r)x_1 = (1+r)^2 x_0 \\ x_3 &= (1+r)x_2 = (1+r)^3 x_0 \\ &\dots \\ x_n &= (1+r)^n x_0 \\ &\dots \end{aligned}$$

Ad esempio,  $r = 1$  corrisponde a una crescita del 100% annuo, cioè la quantità di popolazione raddoppia ogni anno; ovviamente, se  $r$  è negativo la popolazione tenderà a estinguersi.

Il modello proposto da Verhulst tiene conto di una quantità massima possibile di popolazione, che indichiamo con  $X$ , e assume inoltre che il tasso di crescita sia *variabile*, dipendendo dalla quantità di popolazione, e precisamente sia espresso da

$$r\left(1 - \frac{x_n}{X}\right)$$

con  $r$  costante reale positiva; si osservi che il tasso di crescita tende a zero all'avvicinarsi della quantità di popolazione a  $X$ . La legge dinamica che si ottiene è

$$x_{n+1} = \left[1 + r\left(1 - \frac{x_n}{X}\right)\right]x_n$$

ossia:

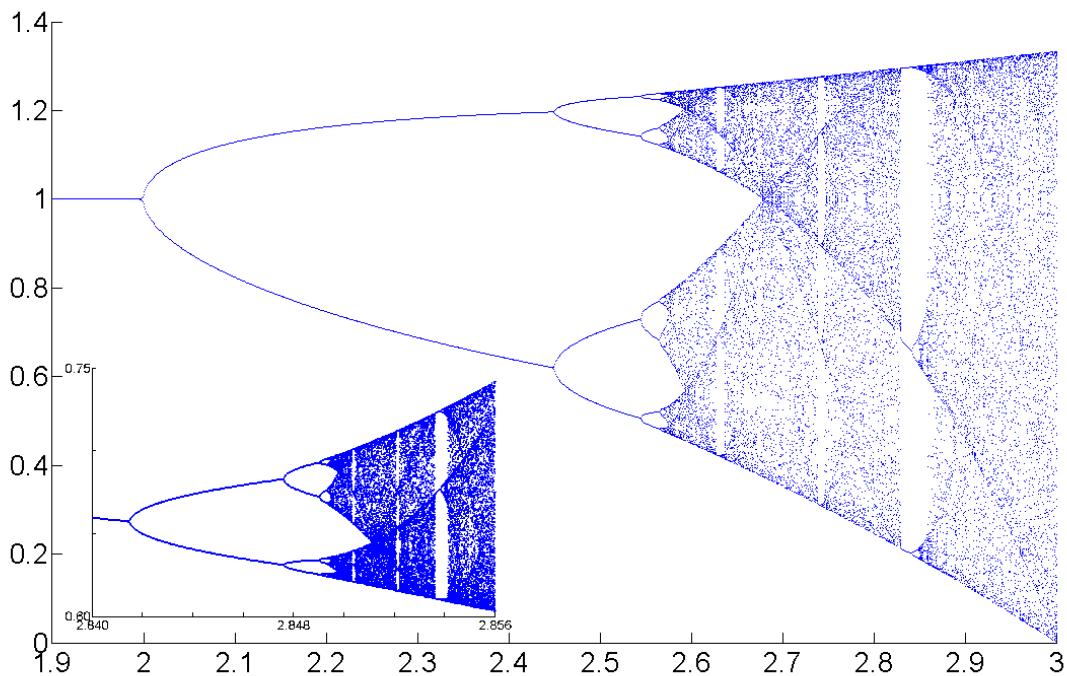
$$x_{n+1} = (1+r)x_n - \frac{r}{X}x_n^2$$

Questa equazione (ricorrente), detta *logistica*, è *non lineare*, poiché  $x_{n+1}$  dipende (anche) dal *quadrato* di  $x_n$ .

Se  $r < 2$  e  $x_0 < \frac{1+r}{r}X$ , allora la popolazione si evolverà fino a stabilizzarsi sulla

quantità  $X$ . Supponendo di *non* partire con  $x_0 = X$  (che è un *punto fisso* della trasformazione, l'altro è 0), proviamo ad aumentare il parametro  $r$ : se  $r$  non supera 1, l'evoluzione arriva velocemente al regime  $X$ ; quando  $r$  è compreso tra 1 e 2, vi giunge con fluttuazioni attorno a  $X$  stesso, e per  $r = 2$  la convergenza è assai lenta.

Nel 1963, il meteorologo Edward Norton Lorenz, famoso per il suo *attrattore strano*, si accorse che per  $r > 2$  il modello di Verhulst descrive certi aspetti dei flussi turbolenti; e proprio da qui si iniziano a osservare i fenomeni più interessanti...



Nel diagramma più grande, per il cui calcolo si è assunto  $X = 1$ , sono riportati (in ordinata) i valori sui quali la *densità* di popolazione  $x$  oscilla *a regime raggiunto*, al variare del parametro  $r$  (in ascissa) da 1.9 a 3.0. In breve, quando  $r$  supera 2 il processo si stabilizza su un'oscillazione regolare tra due valori (dipendenti da  $r$ , ma non dalla *condizione iniziale*  $x_0$ ), a circa 2.45 inizia una ricorrenza ciclica di quattro valori, a circa 2.545 incomincia un ciclo di otto valori, a circa 2.565 il ciclo raddoppia ancora giungendo a sedici valori; i raddoppiamenti continuano sempre più rapidamente, fintanto che, in prossimità di 2.57, l'effetto di duplicazione si verifica un numero *infinito* di volte: non vi sono più oscillazioni di periodo finito a regime, e il comportamento del sistema dinamico diviene *caotico*.

Tuttavia, all'interno del caos, inizia presto a riemergere un ordine, “auto-riproduzione” compresa: le “regioni caotiche” in cui vanno a cadere i valori della variabile  $x$  si dimezzano in numero – questa volta con frequenza sempre minore – all'aumentare di  $r$ ; inoltre, ad esempio, poco prima di 2.83, improvvisamente compare un ciclo di tre valori, e poi, attorno a ciascuno di essi, si ripete – in “scala” ridotta, ma evidente – un processo di raddoppiamento analogo a quello sopra descritto. Il diagramma più piccolo riportato in figura rappresenta un dettaglio (ingrandito) del primo (e cioè il rettangolo da 2.840 a 2.856, in ascissa, e da 0.60 a 0.75, in ordinata), allo scopo di illustrare questo fenomeno di “autoriproduzione”.

Un'osservazione doverosa sui valori numerici sopra espressi con tre o quattro cifre significative: la sensibilità alle condizioni iniziali, unita alle differenze di precisione tra le diverse macchine, fa sì che da computer diversi (pur con algoritmi ugualmente accurati o addirittura identici) sia assai improbabile ottenere lo stesso risultato!

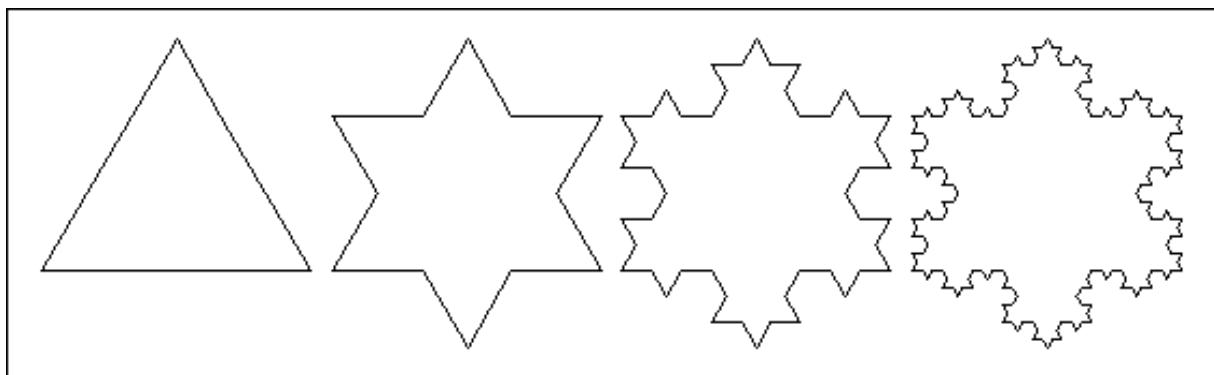
Val quindi la pena di rammentare che, in questo contesto, il termine “caotico” non ha nulla da spartire con “casuale”: indica semplicemente un’elevata sensibilità dell’evoluzione del sistema alle condizioni iniziali (il cosiddetto “effetto farfalla”), e dunque l’inevitabile, intrinseca inattendibilità della sua analisi al computer, almeno da un certo momento in poi, persino nel caso in cui il modello matematico rispecchi alla perfezione la realtà e la simulazione parta da dati precisi. È perciò facile intuire la difficoltà nonché la mole di calcoli che comportano, ad esempio, la messa a punto e l’esercizio di un sistema per le previsioni meteorologiche (sotto tali aspetti, tra i più complessi al mondo), e come queste – nel lungo periodo – diventino comunque inutili. (A riprova, nel sistema visto, si fissi  $r = 2.8$  e si confrontino le evoluzioni che partono da  $x_0 = 0.5$  e da  $x_0 = 0.500001$ , protraendosi per almeno 50 passi.)

## Frattali.

Sull’argomento “frattali”, nel gennaio del 2010 le Edizioni Kangourou Italia hanno pubblicato una monografia, corredata di puntuali spiegazioni e magnifiche illustrazioni; perciò, per conchiudere il discorso avviato nel precedente paragrafo, qui intendo soltanto riepilogare le nozioni essenziali e precisare qualcosa sul procedimento con cui si ottengono le belle immagini a colori relative all’insieme di Mandelbrot.

Nel 1890, Giuseppe Peano – al quale si deve, tra l’altro, l’assiomatizzazione dell’aritmetica, come ricordato a pagina 54 – definì una “curva” continua (di lunghezza infinita) che passa per tutti i punti di un quadrato, e quindi lo “riempie”: una tale curva, dunque, deve avere dimensione 2, la stessa del quadrato!

Nel 1904, lo svedese N. F. Helge von Koch descrisse invece una figura geometrica piana di area finita – anzi, esattamente calcolabile! – delimitata da una “curva” di lunghezza *infinita*. La sua costruzione inizia da un triangolo equilatero; ciascun lato è suddiviso in tre parti uguali e la parte centrale è sostituita da due segmenti, ciascuno della medesima lunghezza della parte sostituita, e formanti con questa ancora un triangolo equilatero, com’è mostrato in figura.



Per ottenere l’*isola* (o “*fiocco di neve*”) di *von Koch*, s’immagini di iterare questo procedimento all’infinito: la “linea costiera” dell’isola di von Koch è la “curva” perimetrale che corrisponde al limite della successione infinita di approssimazioni, le prime quattro delle quali sono riportate nella figura qui sopra.

Non è difficile provare che l'area della porzione finita di piano occupata dall'isola di von Koch è data dall'area del triangolo iniziale moltiplicata per 1.6 (il numero 1.6 si ottiene *esattamente* come uno più la somma di una serie:

$$1 + \sum_{n=0}^{\infty} \frac{4^n}{3^{2n+1}} = 1 + \frac{1}{3} + \frac{4}{27} + \frac{16}{243} + \frac{64}{2187} + \frac{256}{19683} + \dots$$

*ad infinitum*), mentre il suo perimetro è *infinito* (in quanto ognuno degli infiniti passi del processo di costruzione aumenta la lunghezza della linea costiera, moltiplicandola per 4/3).

Si può anche dimostrare che la curva perimetrale è continua, ma non ammette tangente in alcuno dei suoi punti (si potrebbe dire che vi sono infiniti “cambi di direzione” in uno spazio infinitesimo) e ha lunghezza infinita tra due suoi qualsiasi punti distinti.

Qual è la dimensione di questa “curva”? Secondo il matematico ebreo tedesco Felix Hausdorff – che nel 1919 estese la nozione di dimensione – una figura “autosimile” ha dimensione  $d$  quando è costituita da  $N^d$  parti simili (o congruenti) di grandezza (“lineare”)  $1/N$ . Nel nostro caso, ad ogni passo, tre parti di grandezza  $1/3$  ciascuna (da cui  $N = 3$ ) sono sostituite da 4 parti (congruenti), per cui  $3^d = 4$  e quindi  $d = \log_3 4 = 1.261859507\dots$ : una dimensione *non intera*, più vicina a quella di una linea retta o di una circonferenza (che è 1) che non a quella di una superficie piana o sferica (che è 2).

Gli oggetti che hanno una dimensione non intera – ma pure quelli che hanno una dimensione intera “insolita”, come la curva di Peano – sono stati chiamati *frattali*: questo termine fu coniato nel 1975 dal matematico di origine polacca Benoît B. Mandelbrot.

Già nel 1967 egli aveva pubblicato un articolo dal titolo efficace e lievemente provocatorio: “Quanto è lunga la linea costiera della Gran Bretagna?”. Alla luce di quanto sopra esposto – e senza la preoccupazione di dover procedere all’infinito – è chiaro che la risposta non può essere unica: dipende su quale “scala” e con quale strumento si eseguono le misurazioni! Ciò dovrebbe apparire evidente, sebbene la linea costiera della Gran Bretagna manifesti un andamento assai meno regolare della curva di von Koch, nella quale il processo di autoriproduzione è lo stesso ad ogni livello e l’osservazione ravvicinata di un particolare della figura per cogliere più dettagli non dà risultati inaspettati, bensì del tutto prevedibili: infatti la curva di von Koch è un frattale *invariante per trasformazioni lineari* (cioè cambiamenti di scala e traslazioni).

Alla fine degli anni ’70, Mandelbrot riprese alcuni studi, avviati una sessantina d’anni prima dai francesi Gaston Julia (che fu uno dei suoi maestri a Parigi) e Pierre Fatou: le loro ricerche non erano proseguiti più di tanto, forse proprio a causa dell’impossibilità, a quei tempi, di rappresentare in modo adeguato gli oggetti in esame. Mandelbrot, grazie all’insostituibile aiuto del calcolatore (all’epoca lavorava presso un centro di ricerca della IBM a Yorktown Heights, New York), iniziò a

esplorare frattali invarianti per trasformazioni più “complicate”, *non lineari*, in cui il modulo di riproduzione cambia continuamente (anche se spesso lo si può ancora chiamare “autoriproduzione”, come abbiamo fatto poc’ anzi parlando del diagramma di Verhulst): un oggetto frattale di questo tipo rispecchia nello spazio la complessità del comportamento dei sistemi caotici nel tempo, e la sua osservazione ravvicinata può dare esiti del tutto sorprendenti.

Senza pretesa di approfondimento, vediamo come Mandelbrot è arrivato a definire l’importante e meraviglioso insieme legato al suo nome, considerando la legge dinamica

$$x_{n+1} = x_n^2 + c$$

già studiata da Julia e Fatou (che fissavano  $c$  e facevano variare il punto di partenza  $x_0$ ).

Pur apparendo di forma un po’ più semplice rispetto al modello di Verhulst, in questa equazione ricorrente le lettere  $x$  e la lettera  $c$  (che, si noti, sono scritte in neretto) denotano *punti del piano*. In verità si tratta del piano *complesso*, ma possiamo tranquillamente pensare al più familiare piano cartesiano, a patto di definire nel seguente modo le operazioni di elevamento al quadrato e addizione per i punti del piano stesso:

$$(x, y)^2 = (x^2 - y^2, 2xy); \quad (x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2).$$

Naturalmente, la prima componente della coppia che rappresenta un punto è l’ascissa (reale), la seconda componente è l’ordinata (reale); e si osservi che l’addizione è semplicemente fatta componente per componente.

Un punto  $c$  appartiene all’*insieme di Mandelbrot* se e soltanto se, partendo con  $x_0 = \mathbf{0}$  (origine del piano) ovvero con  $x_1 = c$  stesso,  $x_n$  non diverge, cioè non “fugge” sempre più lontano dall’origine del piano, verso l’infinito, all’aumentare di  $n$ ; ciò significa che  $x_n$  non esce mai dal cerchio di raggio 2 centrato nell’origine del piano (se uscisse, allora “scapperebbe”). Quindi l’insieme di Mandelbrot è un *sottoinsieme* dei punti di questo cerchio, circonferenza compresa; in realtà, l’unico punto della circonferenza che vi appartiene è  $(-2, 0)$ , come si può facilmente verificare.

È stato provato che tale sottoinsieme è strettamente collegato con il comportamento di tutti – e non soli! – i processi dinamici della forma che abbiamo visto (equazione logistica),<sup>19</sup> in particolare per quanto concerne la stabilità e le evoluzioni caotiche; è compatto, è connesso (cioè “costituito da un solo pezzo”; A. Douady e J. H. Hubbard, 1982) ed è pure qualcosa di più, e la sua frontiera (cioè il suo “contorno”) è un frattale di dimensione 2 (M. Shishikura, 1991) – molto complicato ed

---

<sup>19</sup> Segnatamente, potete provare a mettere in corrispondenza biunivoca i punti dell’insieme di Mandelbrot che stanno sull’asse delle ascisse (precisamente sono quelli dell’intervallo chiuso  $[-2, 1/4]$ ) con la famiglia delle equazioni logistiche che si ottengono da quella vista nel precedente paragrafo, ponendovi  $X = r/(1+r)$  e facendo poi variare il parametro  $r$  nell’intervallo chiuso  $[0, 3]$ .

estremamente affascinante – che soltanto un elaboratore elettronico ci consente di esaminare e ammirare: la quantità e la qualità dei dettagli che si possono osservare dipendono sia dalla potenza e dalla precisione dell’hardware, sia dall’accuratezza del software. Le immagini meravigliose che spesso abbiamo occasione di vedere, su qualunque supporto siano esse riprodotte, chiaramente rappresentano soltanto una *approssimazione* dell’insieme di Mandelbrot. Il solito Dewdney contribuì a renderle popolari, con un articolo su *Scientific American* dell’agosto 1985, pubblicato in Italia nell’ottobre dello stesso anno: “Un microscopio al calcolatore per gettare uno sguardo sul più complesso fra gli oggetti della matematica”, in cui descrisse un programma per generarle, avvalendosi dell’aiuto di John H. Hubbard della Cornell University, un antesignano di queste applicazioni insieme col francese Adrien Douady.

Per scrivere un programma, a noi dovrebbero bastare le nozioni apprese nel presente capitolo, salvo che per la procedura di visualizzazione.

Come abbiamo visto nel primo paragrafo, ad ogni pixel della matrice (o dello schermo) corrisponde un punto del piano; i punti del piano considerati sono tanto più vicini tra loro quanto più eseguiamo uno zoom sull’immagine. Va da sé che la griglia di punti dovrà essere collocata su una parte significativa del cerchio di raggio 2; a mero titolo d’esempio, alcune regioni “di frontiera” (qui indicate quadrate) che meritano un’esplorazione sono caratterizzate dai seguenti valori:

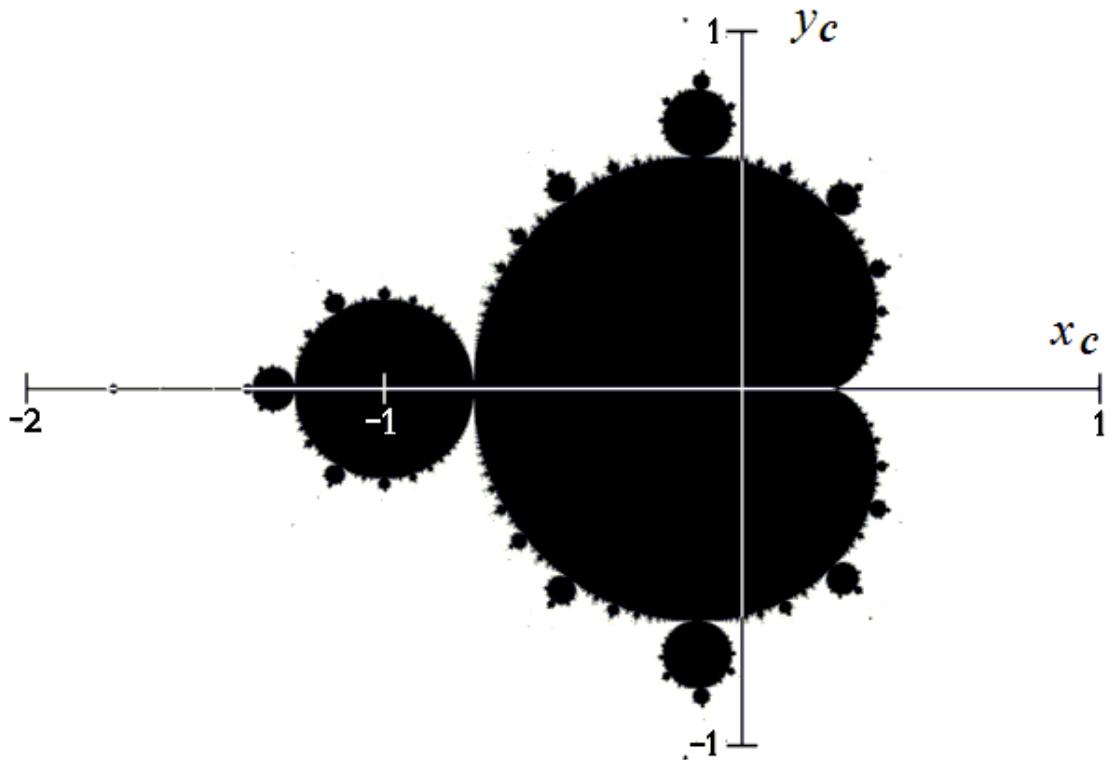
$xa$	$ya$	lato
-0.7	0.5	0.3
-0.66	-0.74	0.30
-0.95	0.22	0.09
-0.20	1.00	0.07
-1.79	-0.01	0.02
-1.40	-0.01	0.02
-0.35	-0.64	0.01
-0.56	-0.65	0.01
-0.76	0.10	0.02
-1.2551	0.0446	0.0040
-0.745444	0.112994	0.000030
-0.743196	-0.105621	0.000012
-0.743187	-0.105614	0.000002

Specialmente le ultime cinque riservano delle belle sorprese, se vi si entra ancora in maggior dettaglio! (Si noti che l’ultima è già contenuta nella penultima.)

Teniamo presente che comunque la macchina non ci permetterà di scendere al di sotto di una certa soglia, data la limitatezza delle rappresentazioni numeriche adottate; inoltre, le operazioni coinvolte (che comportano moltiplicazioni e addizioni tra numeri in virgola mobile) introducono errori di approssimazione, tipicamente ad ogni passo, i cui effetti si propagano ai passi successivi...

In genere, le immagini che vediamo sono così ottenute: un pixel rappresenta un punto  $c$ ; si itera l'equazione (partendo con  $x_0 = 0$ ), al più, per un numero di passi prestabilito, sufficientemente elevato; se prima di raggiungere questo numero massimo di passi il punto  $x_n$  esce dal cerchio di raggio 2, allora al pixel associato a  $c$  è assegnato un colore che dipende dal numero di passi effettuati, altrimenti gli si assegna il colore nero (con ciò assumendo implicitamente che  $x_n$  non esca più dal cerchio, e che quindi  $c$  appartenga all'insieme di Mandelbrot).

All'atto pratico, è come considerare una funzione di due variabili (le coordinate di  $c$ ) la cui quota sia data dal numero di passi compiuti – secondo la legge dinamica con  $c$  fissato – per uscire dal cerchio suddetto quando si parte dall'origine (o, se si preferisce, da  $c$  stesso, che è il secondo punto della successione); se è raggiunto il numero massimo di passi e il punto non è ancora uscito dal cerchio – e soltanto in questo caso – al pixel associato a  $c$  è assegnato il colore nero.



Anche se i punti che “fuggono” fuori dal cerchio fossero lasciati tutti in bianco – anziché colorati in varie tonalità a seconda del numero di passi impiegati per uscirne – le immagini ottenute produrrebbero un certo effetto in chi le guarda, specialmente grazie a quei software che permettono di “entrarvi” con uno zoom continuo...

È dunque quella regione che vediamo in *nero* (anche nelle figure altrove variegate) a rappresentare un'approssimazione dell'insieme di Mandelbrot. Non appena ne ingrandiamo un poco una parte – ad esempio quella più a sinistra, come è mostrato nella figura alla pagina successiva – ci accorgiamo che compaiono alla vista sottoinsiemi sempre più piccoli, di analoga forma, che prima neanche si vedevano o sembravano soltanto dei puntolini neri, e che in realtà sono ramificazioni del corpo principale.



In conclusione, queste immagini sono sì il risultato dell'applicazione di nozioni matematiche piuttosto semplici, tuttavia l'analisi di ciò che vi sta dietro richiede strumenti e metodi decisamente sofisticati. E così, d'altra parte, l'analisi dettagliata di sistemi complessi – resa possibile anche e soprattutto grazie alla simulazione della loro dinamica mediante calcolatore – rientra in un settore della matematica che può legittimamente ritenersi figlio dell'era dei computer.

A questo proposito Corrado Böhm (1923-2017), che fu uno dei pionieri italiani dell'informatica tanto teorica (lambda-calcolo, logica combinatoria) quanto applicata (compilatore in grado di compilare sé stesso, programmazione strutturata), una volta affermò che la matematica è al tempo stesso madre e figlia dell'informatica: se da un lato l'informatica ha fornito alla matematica potenti mezzi sia di calcolo sia – più in generale – di espressione, dall'altro essa trova nella matematica e nella logica la ragione, la causa teorica del suo stesso potenziamento.

**Parole chiave:** matrice di pixel, funzione di due variabili, partizione del piano, formule ricorrenti, procedura definita ricorsivamente, sistema dinamico caotico, modello di Verhulst (equazione logistica a tempo discreto), dimensione non intera, oggetto frattale, insieme di Mandelbrot, grafica al calcolatore.

## **Parte seconda**

**Da zero a due giocatori:  
divertimenti con il computer**



## 6. Automi cellulari e macchine di Turing

Nella gara *Kangourou dell'Informatica* di marzo 2010 fu proposto un quesito dal titolo “Salvaschermo lampeggiante”, in cui appariva una piccola configurazione del più famoso automa cellulare, *Life*, inventato dal matematico inglese John Horton Conway verso la fine degli anni ’60 del secolo scorso. Nel libretto, commentando la soluzione, raccomandammo di cercare in internet certe figure iniziali che generano delle stupefacenti evoluzioni, come la locomotiva a vapore che lascia sbuffi di fumo dietro di sé, oppure il “cannone di Gosper” che spara alianti, o altri schemi ancor più semplici e pur dotati di capacità di crescita infinita... Confidando che qualcuno dei lettori l’abbia fatto, e che quindi abbia avuto occasione di divertirsi a sufficienza, dedichiamo questo capitolo ad altri automi cellulari – anche elementari – e alle macchine di Turing in due dimensioni.

### Che cos’è un automa cellulare?

Senza dare definizioni formali, né classificazioni, richiamiamo alcune nozioni essenziali per capire di che cosa stiamo parlando. Possiamo dire che un *automa cellulare* sia un sistema discreto formato da un numero finito di elementi di uno stesso tipo (le *cellule*) che interagiscono tra loro in base a certe *regole* prefissate.

Di solito, lo scenario in cui “vivono” queste cellule è una *griglia* illimitata: in una sola dimensione (ossia costituita da celle quadrate allineate su una retta), in due dimensioni (in tal caso le celle, tutte di una stessa forma, possono essere quadrate, triangolari o esagonali, in modo da ricoprire il piano) o anche in più dimensioni.

Nella configurazione iniziale che si sceglie, ciascuna cellula occupa una propria cella; successivamente le cellule possono scomparire oppure cambiare colore, e altre ne possono nascere, a seconda dello stato delle celle che si trovano in un opportuno intorno.

In maniera equivalente, senza neppur coinvolgere le cellule, si può pensare più semplicemente a una griglia di celle che cambiano colore o, meglio, stato. Il sistema evolve a intervalli di tempo discreti: ciascuna cella si troverà al tempo  $i + 1$  nello stato stabilito dalle regole di transizione (che sono le stesse per tutte le celle), in base al proprio stato e agli stati delle celle intorno ad essa al tempo  $i$ .

La principale fonte di varietà è costituita dalle regole di transizione, per determinare il prossimo stato di una cella in funzione dell’attuale configurazione dell’intorno che la comprende: infatti, se  $k$  sono i possibili stati di una cella (ogni cella ha uno stesso numero finito di stati possibili), e  $n$  celle (essa compresa) formano l’intorno che può influire sul suo stato successivo, allora le possibili regole di transizione – che danno origine ad altrettanti automi cellulari diversi – sono  $k$  elevato all’esponente  $k^n$ .

Vediamo subito un esempio, il più semplice che si possa fare: gli automi cellulari *elementari* sono unidimensionali; due sono gli stati possibili per una cella ( $0$  = bianco,  $1$  = nero), e lo stato di una cella al tempo  $i + 1$  è stabilito dallo stato della cella stessa e delle due adiacenti (una a destra e una a sinistra) al tempo  $i$ . Esistono

quindi soltanto  $2^8 = 256$  automi elementari, applicabili – s'intende – a un'infinità di configurazioni iniziali.

Adottando la notazione di Stephen Wolfram, che ha studiato le sorprendenti proprietà di questi oggetti, il nome dell'automa elementare (o dell'insieme di regole, o più semplicemente della *regola*, che lo governa) è quel numero decimale che, espresso in notazione binaria usando 8 cifre, fornisce la sequenza di regole di evoluzione nell'ordine discendente da 111 a 000.

Ad esempio, l'automa (o regola) 110 =  $(01101110)_2$  corrisponde alla tabella:

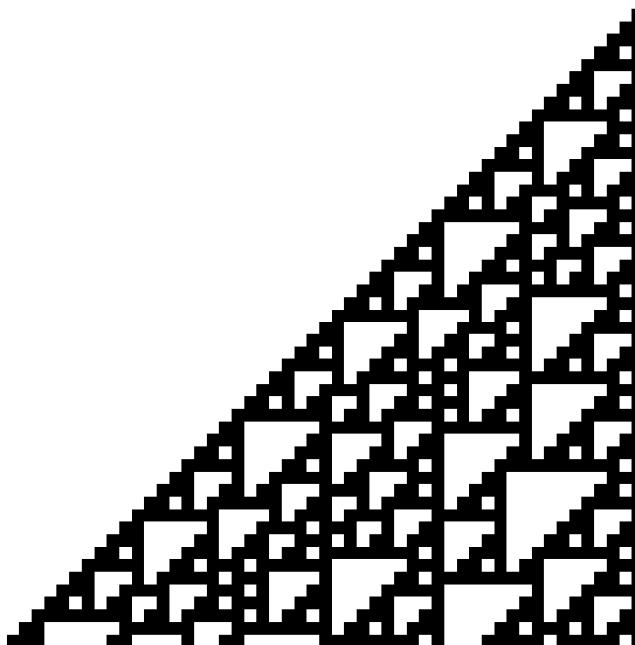
configurazione di tre celle adiacenti	111	110	101	100	011	010	001	000
nuovo stato per la cella centrale	0	1	1	0	1	1	1	0

Notiamo che soltanto in tre casi su otto (da sinistra: il primo, il terzo e il settimo, evidenziati in giallo) la cella centrale cambia stato; si potrebbe anche descrivere questo automa a parole nel seguente modo:

- se una cellula è stretta tra due, allora scomparirà;
- in una cella vuota che abbia una cellula a destra nascerà una nuova cellula;
- non ci saranno altri cambiamenti di stato nel prossimo intervallo di tempo.

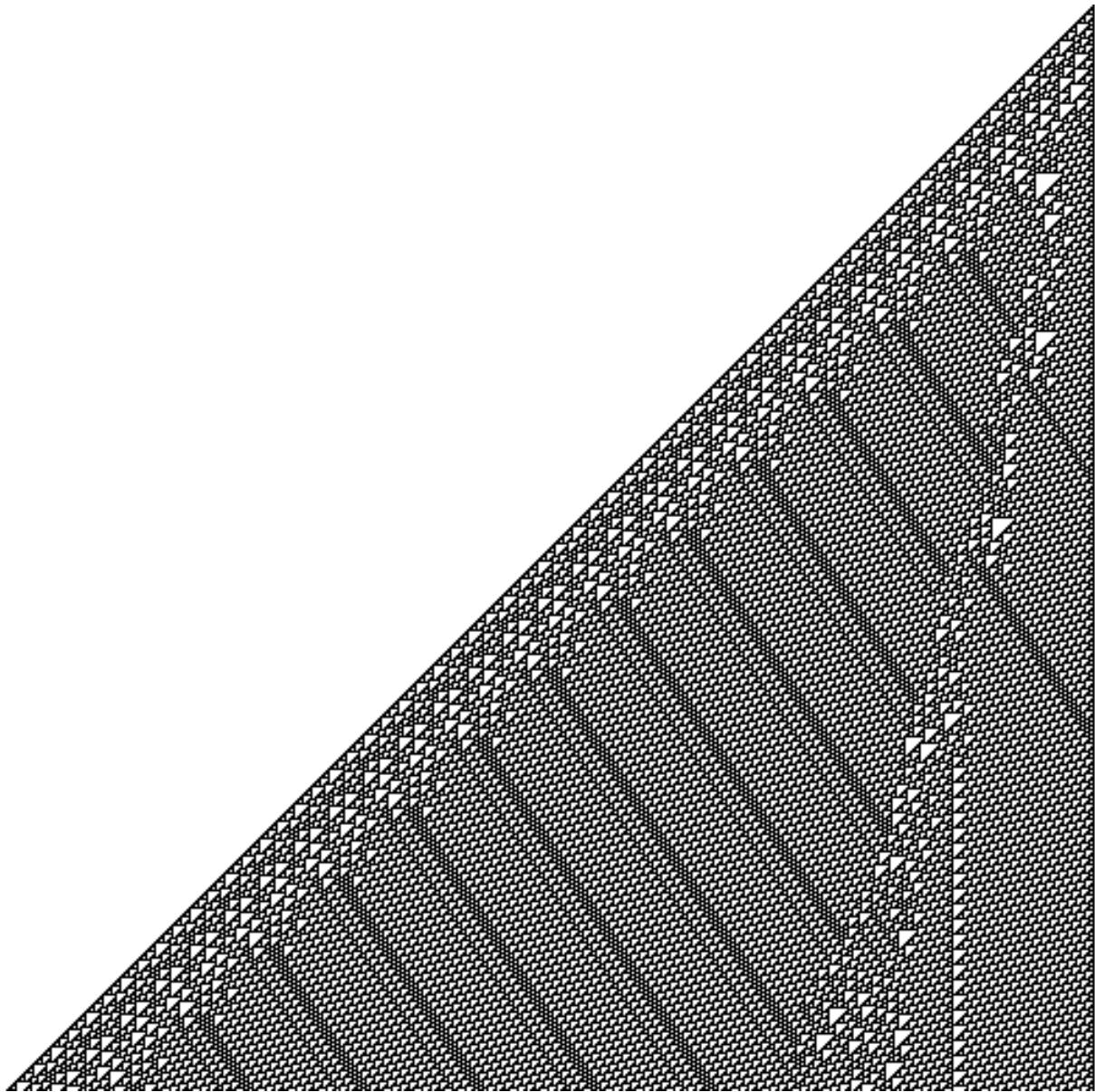
Usiamo una fila di celle, che disegneremo ad ogni intervallo di tempo; tracciando generazioni successive una sotto l'altra si formerà così un diagramma, con un asse dello spazio in orizzontale e un asse del tempo in verticale, orientato verso il basso. Ciò aiuterà ad avere un'idea dell'evoluzione, con un solo colpo d'occhio.

Se applichiamo la regola 110 alla configurazione iniziale costituita da una sola cellula (posta al centro della fila) per 50 intervalli di tempo (o passi), otteniamo la sequenza mostrata in figura:



Questa figura è stata ottenuta con un programma che usa un array di  $N = 103$  bit,

inizialmente col bit centrale posto a valore 1 e tutti gli altri a 0; l'array iniziale è stampato sulla prima riga in alto. In un array ausiliario dello stesso tipo è calcolata la configurazione al tempo successivo, poi copiata nel primo array e stampata; e questo passo è iterato per  $(N - 3)/2$  volte. Notiamo che, in questo caso, tutte le celle a destra di quella centrale sono rimaste vuote, cioè bianche: la ragione si spiega facilmente... La figura seguente riporta il risultato che si ottiene con 500 passi:



Questo automa è importante perché, come già provato per *Life* (John H. Conway, 1982), anch'esso è capace di *computazione universale*, vale a dire che ha le stesse potenzialità di calcolo di una macchina di Turing universale (Stephen Wolfram, 2002; Matthew Cook, 2004).

Negli esempi che seguono consideriamo le regole pari (in una cella vuota situata tra due celle vuote non nasce nulla) e immaginiamo di applicarle sempre alla configurazione iniziale costituita da una sola cellula.

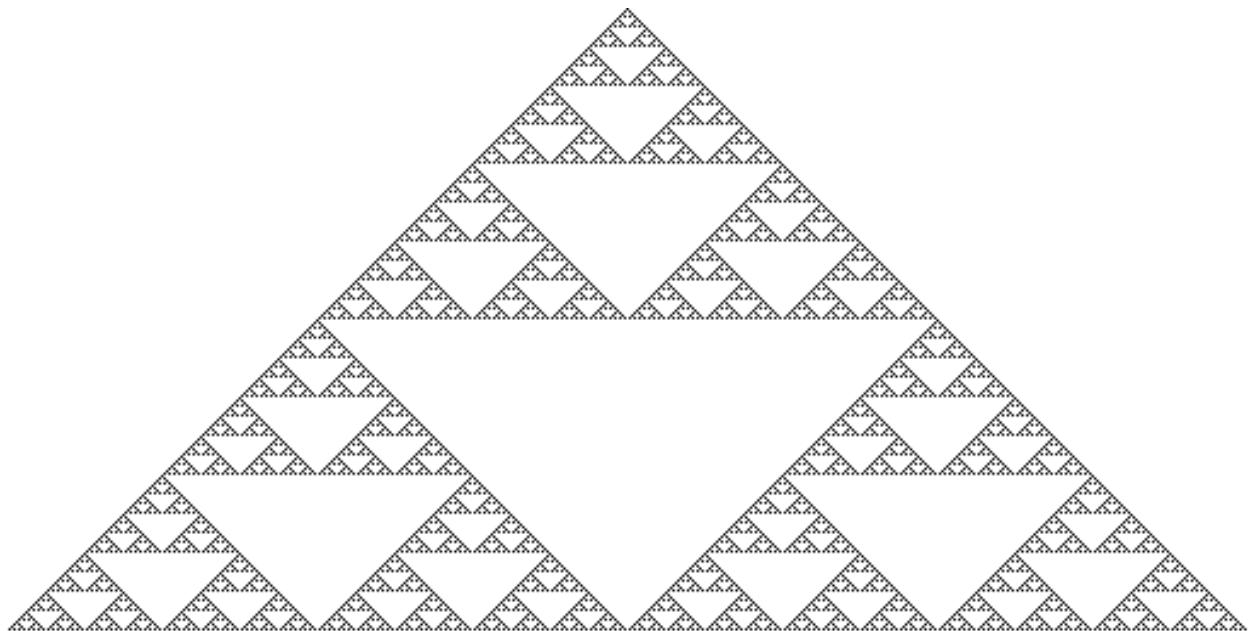
Esiste una regola che produce la sequenza esattamente speculare a quella ora vista, rispetto all'asse verticale: qual è?

[Risposta: la regola  $124 = (01111100)_2$ .]

Ci sono soltanto altre due regole che danno origine ad altrettante sequenze “irregolari” tra loro simmetriche: quali sono?

[Risposta: la regola  $30 = (00011110)_2$  e la regola  $86 = (01010110)_2$ .]

Tra le rimanenti, assai interessante è la regola  $18 = (00010010)_2$ , che evidentemente ha qualche legame col famoso *triangolo di Sierpiński* (si veda anche il quesito “Un’insegna a LED”, nel libretto del 2015). Qui essa è applicata per 255 passi:



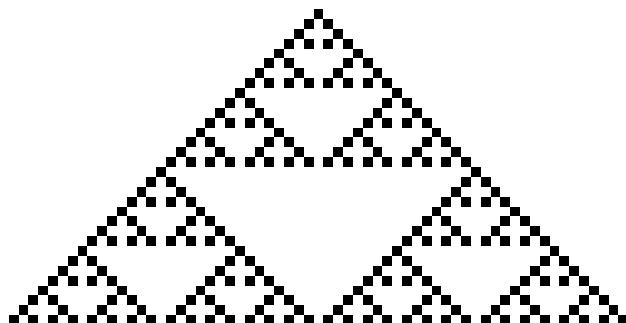
... e la stessa precisa immagine si ottiene costruendo il triangolo di Tartaglia, in modo tale che ciascun numero sia la somma dei due in diagonale sulla riga soprastante, e assegnando poi il colore nero ai numeri dispari.

Tuttavia la 18 non è la sola regola che la genera: ben altre sette regole producono la stessa identica sequenza. Quali sono?

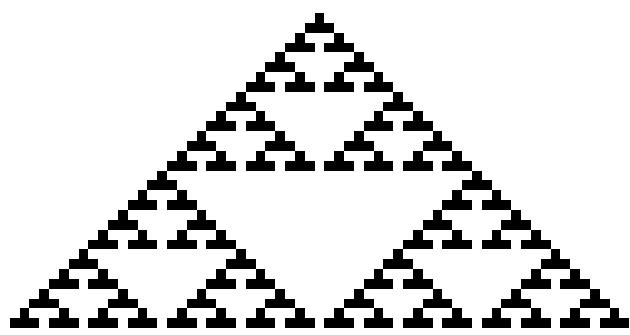
[Risposta: le regole 26, 82, 90, 146, 154, 210 e 218.]

La regola  $182 = (10110110)_2$  produce lo stesso triangolo in negativo, mentre le regole 22 e 126 originano un motivo un po’ diverso rispetto alla 18, come si vede nelle seguenti immagini – limitate a 31 passi:

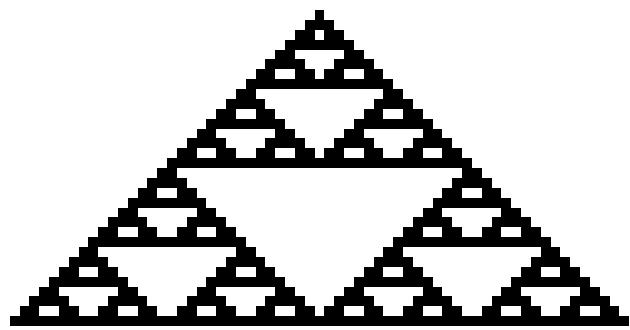
regola 18:



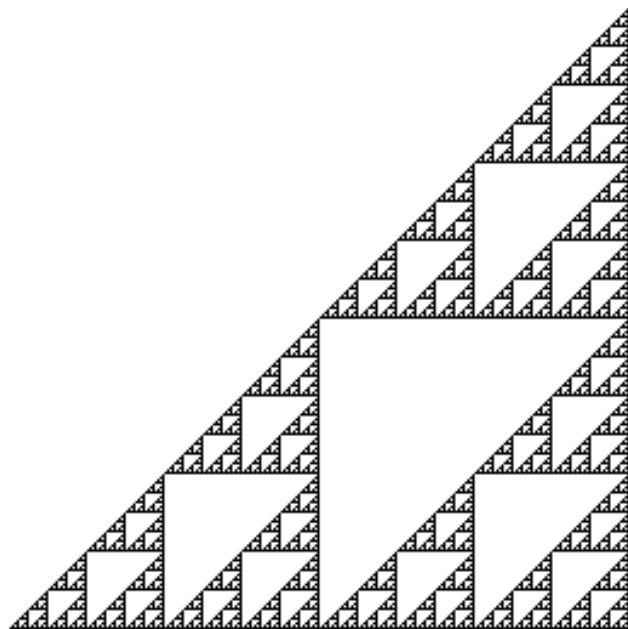
regola 22:



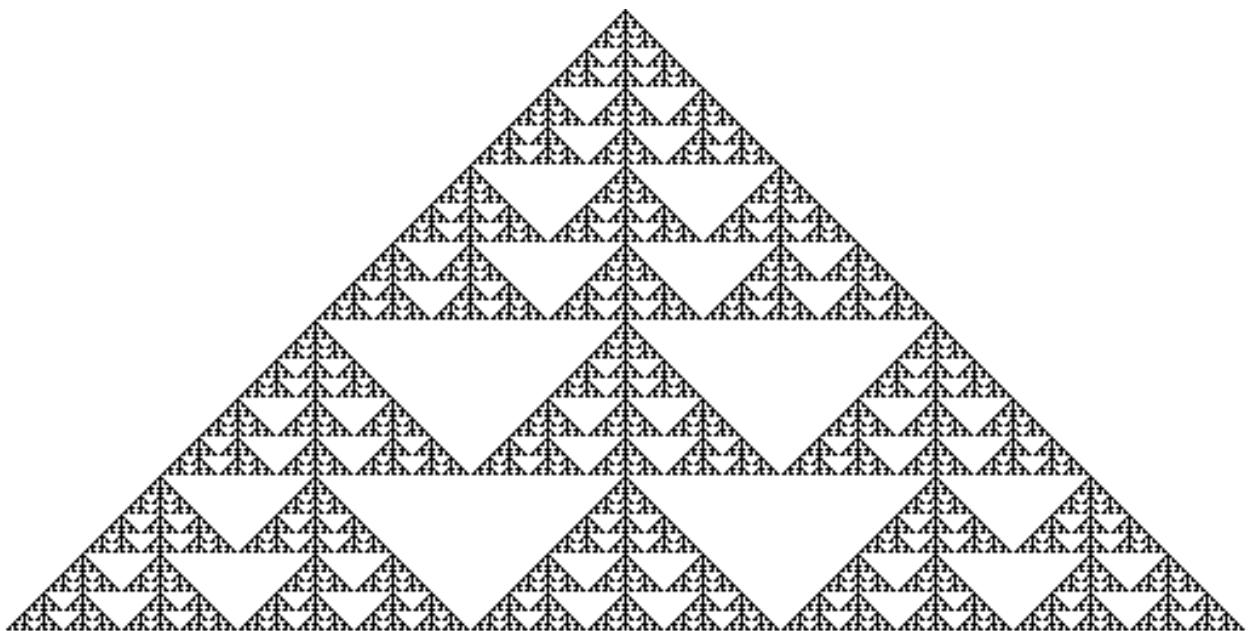
regola 126:



Il “tema Sierpiński” ricorre pure con la regola 102 – qui applicata per 255 passi:



e con la sua speculare 60; ma la regola da cui scaturisce la variazione più bella è la 150 – come ci mostra la figura alla pagina successiva, ov’è applicata di nuovo per 255 passi.



### Un automa di Ulam.

I primi automi cellulari furono ideati intorno alla metà del Novecento da due eminenti matematici statunitensi, entrambi attivamente impegnati nel mondo della computazione: Stanisław (Stan) Ulam (1909-1984) di origine polacca e John von Neumann (nato János Lajos Neumann, 1903-1957) di origine ungherese, uno dei padri dei moderni calcolatori elettronici. Da allora, specialmente grazie all’impiego dei computer, questi automi – che aiutano a esplorare la complessità di sistemi superiori – sono stati studiati a fondo e utilizzati in diversi campi, soprattutto per modellare e simulare (in modo discreto) vari fenomeni naturali che dipendono da leggi “locali”: ad esempio il comportamento dei gas perfetti, l’evoluzione di una popolazione o di un ecosistema eccetera. E non va dimenticato che pure in natura si trovano “tracce” che sembrano lasciate da qualche automa cellulare: ad esempio il motivo mostrato da certe conchiglie marine, come quelle del *Conus textile* (la fotografia qui sotto è di un esemplare proveniente dal Madagascar).



Ma passiamo ora a un semplice automa in due dimensioni, proposto proprio da Ulam nel 1962. Questa volta ne definiremo le regole di evoluzione mediante una procedura che usa due matrici di bit, A e B (B come ausiliaria, per memorizzarvi la configurazione successiva a quella contenuta in A), entrambe con  $r$  righe numerate da 0 a  $r - 1$  e  $c$  colonne numerate da 0 a  $c - 1$ . Ecco la procedura, agevolmente traducibile in un linguaggio di programmazione:

1. assegna alla matrice A la configurazione iniziale (ad esempio, assegna a tutti i suoi elementi il valore 0 = bianco, tranne a quello centrale, al quale assegna il valore 1 = nero) e la stampa;
2. **per ogni**  $i = 1, \dots, r - 2$ :  
**per ogni**  $j = 1, \dots, c - 2$ :  
**se**  $A(i, j)$  vale 1 oppure la somma dei quattro bit adiacenti ortogonalmente ad  $A(i, j)$  vale 1 [ciò significa che  $A(i, j)$  vale 1 oppure uno e uno soltanto tra i quattro elementi  $A(i - 1, j)$ ,  $A(i + 1, j)$ ,  $A(i, j - 1)$  e  $A(i, j + 1)$  vale 1],  
**allora** assegna a  $B(i, j)$  il valore 1,  
**altrimenti** assegna a  $B(i, j)$  il valore 0;
3. copia la matrice B (che adesso contiene la configurazione successiva) nella matrice A e la stampa;
4. **se** si desidera continuare, **allora** ritorna al punto 2.

Partendo dalla configurazione iniziale suggerita nell'esempio e supponendo  $r$  e  $c$  entrambi  $\geq 11$ , provate a simulare a mano l'esecuzione dei punti 2 e 3 per quattro volte.

Può una cella (cioè un elemento della matrice A) passare dallo stato (valore) 1 allo stato 0?

[*Risposta:* no! Perché?]

Come può essere espressa con semplici parole, ma in maniera precisa, la regola di cambiamento di stato di una cella?

Si darà prima o poi il caso in cui è inutile continuare, perché la matrice A non sarà più modificata alle successive iterazioni?

[*Risposta:* sì, in virtù della precedente risposta e poiché la matrice è limitata.]

Servirsi di due matrici, mantenendole in memoria, è certamente la maniera più semplice per simulare l'evoluzione di un generico automa bidimensionale. Tuttavia un automa cellulare può essere realizzato su un *microchip* specializzato, le cui celle lavorino in parallelo in modo sincrono, per garantire la massima efficienza.

## Neve aliena.

Questo nome è stato dato a un altro semplice automa cellulare bidimensionale, ideato da Clifford Pickover e descritto nel suo già citato *Wonders of Numbers*.

La procedura da seguire non è tanto dissimile da quella descritta nel paragrafo precedente:

1. assegna alla matrice A la configurazione iniziale (ad esempio, assegna a tutti i suoi elementi il valore 0 = bianco, tranne a quello centrale, al quale assegna il valore 1 = nero) e la stampa; inoltre assegna il valore 0 alla variabile intera *passo* e a tutti gli elementi della matrice B;
2. **per ogni**  $i = 1, \dots, r - 2$ :
 

**per ogni**  $j = 1, \dots, c - 2$ :

**se**  $A(i, j)$  vale 0 **allora**:

**se** *passo* vale 0 o un multiplo di 6,

**allora** assegna alla variabile intera *somma* la somma dei bit adiacenti ortogonalmente ad  $A(i, j)$  (al più, varrà 4),

**altrimenti** assegna alla variabile intera *somma* la somma dei bit confinanti con  $A(i, j)$  (al più, varrà 8);

**se** *somma* vale esattamente 1,

**allora** assegna a  $B(i, j)$  il valore 1,

**altrimenti** assegna a  $B(i, j)$  il valore 0 (\*);
3. copia la matrice B (che adesso contiene la configurazione successiva) nella matrice A e la stampa;
4. incrementa *passo* di un’unità;
5. **se** *passo* non ha raggiunto il massimo stabilito, **allora** ritorna al punto 2.

In sostanza, ogni 6 passi (a partire dal primo) ciascuna cella a valore 0 cambia stato se sopra o sotto o a sinistra o a destra vi è una sola cella a valore 1; ad ogni altro passo ciascuna cella a valore 0 cambia stato se tutto intorno ad essa vi è una sola cella a valore 1.

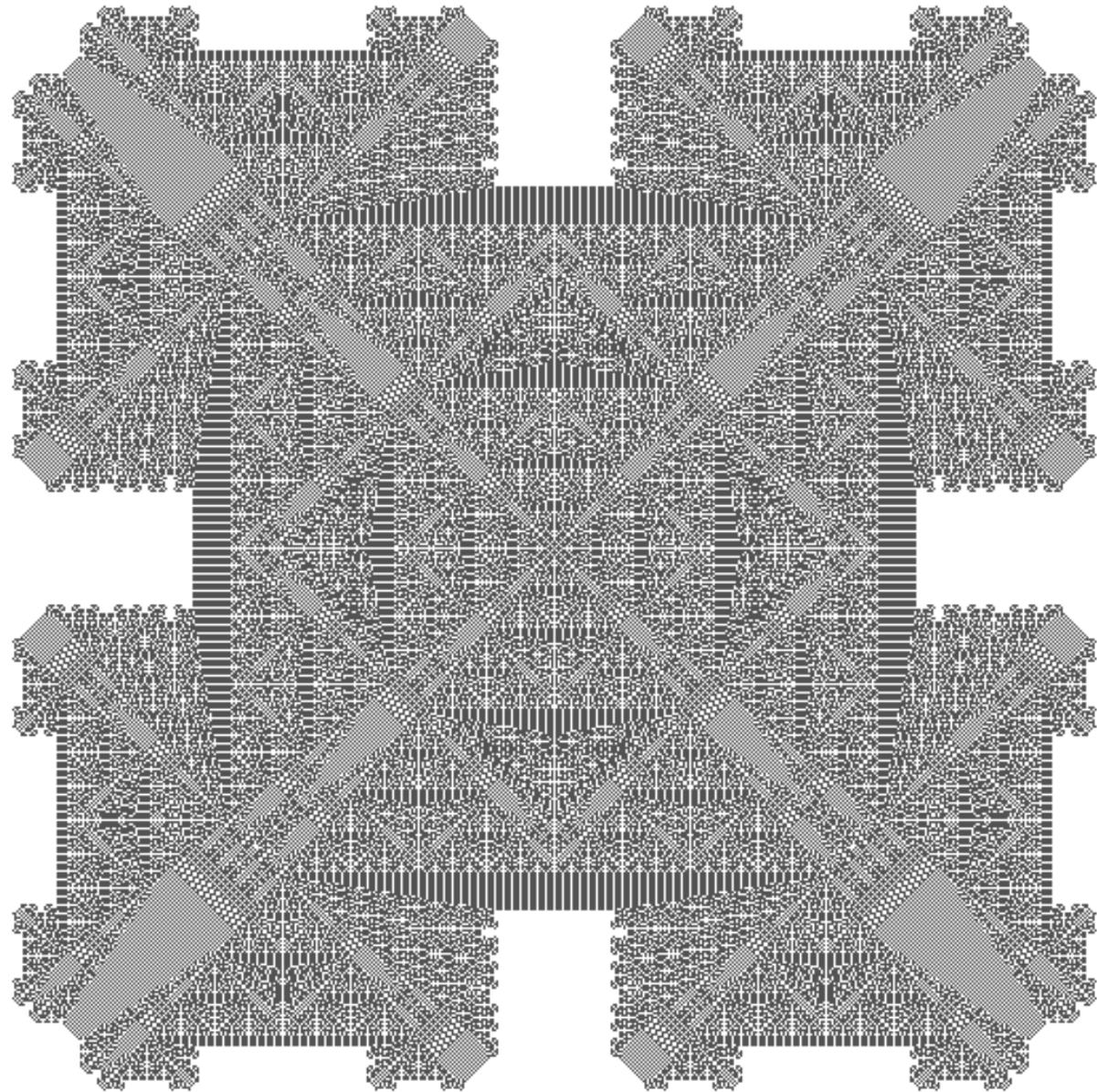
Considerando inizialmente una matrice quadrata  $513 \times 513$  con la sola cella centrale nera, dopo 255 passi si ottiene la splendida configurazione mostrata alla pagina successiva... che ricorda appunto uno strano fiocco di neve!

Varianti possibili: ogni  $m$  passi, ad esempio 2, ... anziché 6, oltre naturalmente all’introduzione di altri “difetti” nella configurazione iniziale, attorno ai quali si formerà il fiocco di neve, ad esempio tre celle nere disposte a triangolo intorno al centro, anziché quella centrale soltanto...

Potete anche apportare delle modifiche in altri punti della procedura, e poi osservare come cambierà l’evoluzione: ad esempio, omettere la linea (\*), oppure assegnare il valore 0 a tutti gli elementi della matrice B all’inizio di ciascun passo, anziché *una tantum* al punto 1.

Quanto al numero di passi, conviene sempre tenersi al di sotto della minima distanza di una cellula dal bordo nella configurazione iniziale. Tuttavia un’interessante alternativa consiste nell’immaginare il mondo (cioè la matrice) di forma *toroidale*, ossia “a ciambella”: sotto l’ultima riga si ritrova la prima riga e a destra dell’ultima colonna si ritrova la prima colonna. Ad esempio, a nord-est della cella di riga  $i$  e colonna  $j$  si troverà la cella di riga  $(i + r - 1)$  *modulo*  $r$  e colonna  $(j + 1)$  *modulo*  $c$

(come sappiamo, si chiama *modulo* l'operazione che produce il resto della divisione intera) e analogamente per le altre sette direzioni. Allora l'automa potrà avanzare per un maggior numero di passi, e ovviamente i risultati cambieranno... Che cosa accadrà?



### A che cosa può servire un automa cellulare?

Abbiamo accennato alle prime ricerche di John von Neumann all'inizio degli anni '50, animate dall'idea di definire dei sistemi in grado di riprodursi, come fa un organismo vivente.

Un semplice automa bidimensionale che *replica sé stesso* fu escogitato nel 1960 dal fisico Edward Fredkin del MIT. Le celle sono binarie e l'intorno di una cella è costituito dalle sole quattro ad essa adiacenti ortogonalmente; se a valore 1 ce ne sono un numero dispari (una o tre) allora il suo stato successivo sarà 1, altrimenti sarà 0.

Provate a realizzarlo: vi accorgerete che, dopo un certo numero di passi, compariranno quattro copie della configurazione originale; in seguito, ciascuna copia sarà a sua volta replicata per quattro volte, e così via. Vi suggeriamo anche un altro esperimento: provate a considerare un intorno di  $5 \times 5$  celle, questa volta includendo la cella centrale, e ad applicare la stessa regola di disparità...

Dagli anni '70 in poi, sulla scia del "gioco" *Life* di Conway, moltissimi automi cellulari – sebbene costituiscano una frazione insignificante della loro moltitudine! – sono stati studiati da diversi ricercatori. In particolare, restando su una matrice illimitata di celle binarie quadrate, il gruppo del MIT formato da Edward Fredkin, insieme con Norman Margolus, Tommaso Toffoli e Gérard Y. Vichniac, focalizzò l'attenzione su regole per cui il nuovo stato di una cella dipende soltanto dal numero di adiacenti a valore 1 nello stato precedente, ma non dalla loro posizione nell'intorno considerato. (Quali, tra quelli di cui abbiamo parlato, rientrano in questa classe?)

Un'ulteriore possibile specificazione: alla cella centrale è assegnato il valore 1 se e soltanto se il numero di celle a valore 1 nel suo intorno raggiunge o supera una certa soglia. Se l'intorno considerato è costituito dalle quattro adiacenti ortogonalmente più la stessa cella centrale, e la soglia da raggiungere o superare è 3, allora nel corso dell'evoluzione si possono formare delle catene continue di 1, ammesso che la "concentrazione" iniziale di 1 sia sufficiente (pari almeno alla metà delle celle).

La formazione di un percorso ininterrotto attraverso uno spazio è detta *percolazione* ed è un fenomeno importante in fisica (per quanto concerne, ad esempio, la conducibilità delle leghe o la struttura di alcuni polimeri) e pure in altri campi (si pensi alla propagazione degli incendi nei boschi o alla diffusione delle malattie infettive).

Per far sì che la matrice tenda a riempirsi di 1, la soglia dev'essere abbassata a 2, ed è sufficiente una concentrazione iniziale inferiore: un tale automa può modellare il processo di *nucleazione*, anch'esso di rilievo nella fisica dello stato solido (ad esempio, dà inizio alla crescita di un cristallo).

Un automa cellulare è detto *reversibile* se da una qualsiasi configurazione raggiunta durante il processo evolutivo può tornare indietro fino a quella iniziale, purché siano applicate le regole "alla rovescia". Ciò implica che le regole, una volta invertite, rimangano deterministiche: in altre parole, ogni stato deve avere non soltanto un unico successore, ma anche un unico predecessore. (Ci sono automi reversibili tra quelli di cui abbiamo parlato?)

Un automa reversibile non può avere stati "di attrazione", non può entrare in un ciclo di stati o in uno stato stabile, né uscirvi. Si può dire che la quantità di informazione di un automa reversibile rimane costante nel tempo: i "calcoli" possono essere rovesciati poiché non v'è perdita di informazione!

Sulla base di questa idea, il gruppo del MIT sviluppò modelli di calcolo realizzabili con automi reversibili che non consumano energia, almeno in linea di principio. È rimasto famoso il modello cosiddetto "a palle di biliardo", in cui i bit 1 si muovono come in un sistema meccanico ideale, senza attriti o altre dissipazioni e con urti

perfettamente elastici; ad esempio, poiché l'informazione si conserva, dai risultati di un calcolo è possibile risalire ai dati iniziali.

In generale, per far sì che un automa abbia determinate proprietà, come ad esempio la reversibilità, può essere opportuno o addirittura necessario considerare non soltanto lo stato attuale dell'intorno, per calcolare il prossimo stato di una cella, ma anche lo *stato precedente* (o alcuni stati precedenti) dello stesso intorno.

Per simulare certi fenomeni del mondo reale, nella definizione del comportamento di un automa possono essere introdotte *regole probabilistiche*: infatti, se possono intervenire elementi di casualità, sarà necessario legare il verificarsi di un certo evento a una qualche probabilità (rimanendo nell'ambito della fisica, si pensi alla descrizione di materiali ferromagnetici).

Spesso, per far apparire un modello matematico il più possibile aderente alla realtà fisica, si prevedono grandezze variabili con continuità in funzione sia del tempo sia dello spazio, pure continui; per contro, un automa cellulare è un sistema *discreto* in tutti questi aspetti. Ciò non toglie che talvolta (e c'è chi sostiene, addirittura, sempre) gli automi cellulari possano essere assai utili per modellare validamente certi fenomeni naturali, in particolare laddove una configurazione su larga scala si sviluppa completamente dalle interazioni “a breve raggio” di molte unità identiche; si pensi alle molecole d'acqua che arrivano a comporre un fiocco di neve o alle concrezioni calcaree che danno origine a certe conchiglie: il modello computazionale della loro crescita è proprio un automa cellulare. Ma potremmo spingerci oltre, e pensare a sistemi fisici assai più grandi, come le galassie...

A parte le applicazioni in campo informatico, alle quali abbiamo accennato, dal nostro punto di vista gli automi cellulari costituiscono un “mondo digitale” che senza dubbio merita di essere esplorato di per sé.

Studi sistematici e approfonditi sugli automi cellulari sono stati compiuti da Stephen Wolfram a partire dagli anni '80. Uno dei suoi primi lavori, in cui ne propose già una classificazione, fu discusso in un seminario tenuto nel 1983 al Los Alamos National Laboratory; si può trovare in rete al seguente indirizzo:

<http://www.stephenwolfram.com/publications/academic/cellular-automata.pdf>.

All'epoca, Wolfram svolgeva attività di ricerca presso l'Institute for Advanced Study di Princeton e iniziò a indagare sulle relazioni tra la descrizione del comportamento degli automi cellulari e i linguaggi formali, nonché alcune forme d'arte tra cui la musica. Una gigantesca raccolta di risultati e scoperte è presentata nel suo libro *A New Kind of Science* (Wolfram Media, 2002). In rete, partendo dal sito <http://www.stephenwolfram.com/>, si trova parecchio di questo materiale.

Inoltre, all'indirizzo <http://tones.wolfram.com/generate/> è offerta la divertente opportunità di trasformare una sequenza di passi di un automa in note musicali, generando così delle melodie.

Non possiamo dimenticare il ricchissimo sito “CelLab”, assai ben curato da Rudy Rucker e John Walker (<http://www.fourmilab.ch/celldab/>), contenente storia, teoria e applicazioni a non finire con le quali ci si può divertire, ottenendo spesso risultati strabilianti (si consulti la *User Guide*).

La teoria degli automi cellulari è immensamente ricca e, come in parte possiamo intuire dai pochi esempi visti, anche con regole e configurazioni iniziali assai semplici si può produrre una grande varietà di comportamenti, spesso inaspettati. Lo stesso Edward Fredkin – che già negli anni ’70 si occupò di visione artificiale e di scacchi al computer – continuò a studiarli. Un suo più recente articolo, scritto con Daniel B. Miller presso la Carnegie Mellon University e pubblicato nei *Proceedings of the 2<sup>nd</sup> Conference on Computing Frontiers* del 2005, descrive un nuovo automa cellulare in tre dimensioni (ma sempre con celle a due stati) reversibile e capace di computazione universale; si trova al seguente indirizzo:

<http://arxiv.org/ftp/nlin/papers/0501/0501022.pdf>.

L’ingegnere tedesco Konrad Zuse (del quale abbiamo già parlato) scrisse tra il 1967 e il 1969 il lavoro che diede l’avvio alla fisica digitale, *Rechnender Raum*, “il cosmo che calcola”, in cui avanzò l’ipotesi che l’evoluzione dell’universo intero sia calcolata da un “dispositivo” discreto, il cui contenuto informativo non può aumentare. Sia Fredkin sia Wolfram concordano con lui, ritenendo che l’universo fisico possa proprio essere il prodotto di un automa cellulare in azione!

Ribadiamo un paio di concetti, già espressi nel libretto del 2010. Un automa cellulare può essere considerato un *gioco senza alcun giocatore*, se si escludono chi detta le regole e chi escogita la situazione iniziale: stabilite infatti le regole e la configurazione di partenza, si può soltanto assistere all’evoluzione del sistema, senza poter più intervenire.

Infine, come già detto anche in questa sede, certi automi cellulari rivestono pure un particolare interesse per l’informatica teorica, poiché hanno le potenzialità di un computer universale: è stata dimostrata l’esistenza di *automi cellulari universali*, anche semplici come la regola 110 e *Life*, capaci di emulare il comportamento di qualsiasi altro automa cellulare e di qualsiasi macchina di Turing...

A proposito di *macchine di Turing*: nel prossimo paragrafo impareremo finalmente a usarle... in due dimensioni, anziché in una soltanto, per rendere così l’approccio più divertente e produrre ancora qualche immagine suggestiva! Dovrebbe essere superfluo precisare che una dimensione in più non aggiunge nulla alle capacità di calcolo delle classiche macchine di Turing.

## Macchine di Turing bidimensionali.

Riprendiamo il discorso avviato nel secondo capitolo. Negli anni ’30 del Novecento, quando ancora non esistevano i computer, Alan M. Turing ideò un formalismo per definire un qualsiasi algoritmo (o almeno così pare, visto che nessuno è mai riuscito a descrivere sulla carta un qualche procedimento di computazione che non possa essere espresso in tale contesto): fu uno dei primi e più usati modelli di calcolo, assai elegante per la sua semplicità. In un memorabile lavoro, dato alle stampe tra il 1936 e il 1937, Turing stesso se ne servì per dimostrare che le future macchine non avrebbero mai potuto decidere o calcolare tutto ciò che può essere immaginato! Nel contempo, provò che una delle sue macchine, dotata in teoria di memoria illimitata,

poteva costituire un *algoritmo universale*, in grado di eseguire un qualsiasi algoritmo, opportunamente codificato, su un qualsiasi suo input.

Il progettino qui presentato, con riferimento a più moderne metodologie di programmazione (può essere sviluppato ad esempio in linguaggio C++), trae ispirazione dai lavori di Christopher Langton, Allen H. Brady e Greg Turk, risalenti alla seconda metà degli anni '80, sulle *macchine di Turing in due dimensioni* (MdT2D), poi chiamate *turmites* (“turmiti”): le termiti di Turing!

Cominciamo col descrivere precisamente le *classi* di oggetti che serviranno al nostro scopo.

Una MdT2D è un automa in grado di muoversi su una tavola piana, suddivisa in celle quadrate di vari colori – i quali, in numero finito, giocano il ruolo dei simboli dell’alfabeto di una classica macchina di Turing (MdT). Pensiamo a una grande scacchiera con le case colorate, una delle quali è occupata da un automa che si può muovere di una sola casa per volta (ortogonalmente), magari dopo aver cambiato il colore della casa lasciata. In apparenza lo scenario è un tantino diverso rispetto a quello di un automa cellulare bidimensionale, perché qui è la “macchina” che – seguendo le proprie regole, in base al proprio stato attuale e al colore della casa in cui si trova – cambia tale colore, si sposta e cambia stato...

Poiché una classica MdT si muove lungo un nastro *illimitato* (ovvero lo sposta sotto di sé), analogamente la nostra tavola dovrebbe estendersi senza limiti, ricoprendo interamente un piano infinito; tuttavia, noi supponiamo che abbia, al più, MAX\_RIGHE righe e, al più, MAX\_COLONNE colonne, essendo MAX\_RIGHE e MAX\_COLONNE due costanti prefissate.

Un’osservazione importante: l’input di una computazione è sempre rappresentabile in uno spazio finito (così come l’output, se il processo ha termine); per avere un modello di calcolo universale basta immaginare di poter aggiungere *ad libitum* nuove celle, ognqualvolta servano.

Ogni cella ha un *colore*, rappresentato da un intero  $\geq 0$  e  $<$  num\_colori, essendo num\_colori ( $\geq 2$  e  $\leq$  MAX\_COLORI, costante prefissata) il numero di colori riconosciuti dalla macchina.

Ad ogni passo della computazione, prima di fermarsi, la macchina si trova in uno *stato*, rappresentato da un intero  $\geq 0$  e  $<$  num\_stati, essendo num\_stati ( $\leq$  MAX\_STATI, costante prefissata) il numero di stati “attivi” possibili per la macchina; ogni eventuale stato  $\geq$  num\_stati sarà considerato “finale”: se e quando la macchina lo raggiunge, si ferma.

La macchina si trova inizialmente, per convenzione, nello *stato zero* (0), in una data cella della tavola, e da lì inizia la computazione, seguendo le regole espresse dalla sua *tabella di triple* (di num\_stati righe per num\_colori colonne) che costituisce l’*algoritmo*: se la macchina è nello stato  $s$ , e il colore della cella in cui si trova è  $c$ , e la tripla di riga  $s$  e colonna  $c$  in questa tabella è  $(c1, sposta, s1)$ , allora la macchina cambia in  $c1$  il colore della cella in cui si trova, poi si sposta come stabilito da *sposta* (= 0 sta ferma, = 1 si muove di una cella verso l’alto, = 2 si muove di una

cella a sinistra, = 3 si muove di una cella verso il basso, = 4 si muove di una cella a destra) e infine si porta nello stato *s1*.

La computazione termina quando si verifica almeno una delle seguenti condizioni per la macchina:

1. arriva in uno stato  $\geq \text{num\_stati}$ ;
2. “esce” dalla tavola;
3. ha eseguito `MAX_PASSI` (costante prefissata) passi di computazione.

Dunque, a differenza delle classiche MdT (che si spostano lungo un nastro unidimensionale ma potenzialmente infinito, e senza alcun limite al numero di passi), qui la computazione termina sempre. Quando ciò avviene, la tavola (che costituisce il risultato della computazione) può essere visualizzata come immagine bidimensionale, ad esempio facendo corrispondere una cella a un pixel e mappando il numero che rappresenta il colore della cella in una opportuna tavolozza di colori (o, più semplicemente, scala di grigi).

Diamo ora qualche suggerimento per realizzare un programma che permetta di simulare il comportamento di una MdT2D su una tavola, assumendo – come *precondizione* – che i colori iniziali delle celle (tra i quali di certo vi saranno 0 e 1) siano riconosciuti dalla macchina. A questo scopo abbiamo definito due *classi*, una per rappresentare le MdT2D e l’altra per le tavole.

Chi non è interessato ai dettagli può limitarsi ad osservare alcuni dei risultati che abbiamo ottenuto con opportune macchine – vale a dire algoritmi.

Definiamo dapprima una classe `MdT2D`, con

- tre *campi privati*:
  - `tabella`, matrice (di `MAX_STATI` righe per `MAX_COLORI` colonne) di *triple* (come illustrato sopra);
  - `num_stati` e `num_colori`, che dicono, rispettivamente, quante righe e colonne (a partire da 0) di tale matrice sono effettivamente usate,
- un *costruttore*, che permetta di inizializzare in modo appropriato tali campi, ad esempio leggendo i dati da un *file* di testo specificato (si veda l’esempio sotto riportato),
- e due *metodi getter*:
  - uno che restituisce il valore del campo `num_stati`;
  - l’altro che, dati lo stato corrente *s* e il colore della cella corrente *c*, restituisce la tripla di riga *s* e colonna *c* in `tabella`.

L’importante è che in `tabella` siano assegnate *tutte* le triple utili, in numero uguale a `num_stati`  $\times$  `num_colori`.

Sicché si può pensare che il programma possa disporre dei dati necessari in un *file* di testo – ad esempio con le *nove* linee scritte nel riquadro a sinistra della figura che segue, avendo posto sulla prima linea i valori per `num_stati` e `num_colori` – e

che se ne serva per costruire l’istanza della classe MdT2D – che in questo caso rappresenta la MdT2D con quattro stati (0, 1, 2, 3), due colori (0, 1) e la tabella 4×2 le cui triple sono elencate, per righe, a iniziare dalla seconda linea del *file* stesso (si veda il riquadro a destra in figura).

4 2 1 1 1 0 3 3 1 2 2 0 4 0 1 3 3 0 1 1 1 4 0 0 2 2	colore stato	0	1
	0	( 1, 1, 1 )	( 0, 3, 3 )
	1	( 1, 2, 2 )	( 0, 4, 0 )
	2	( 1, 3, 3 )	( 0, 1, 1 )
	3	( 1, 4, 0 )	( 0, 2, 2 )

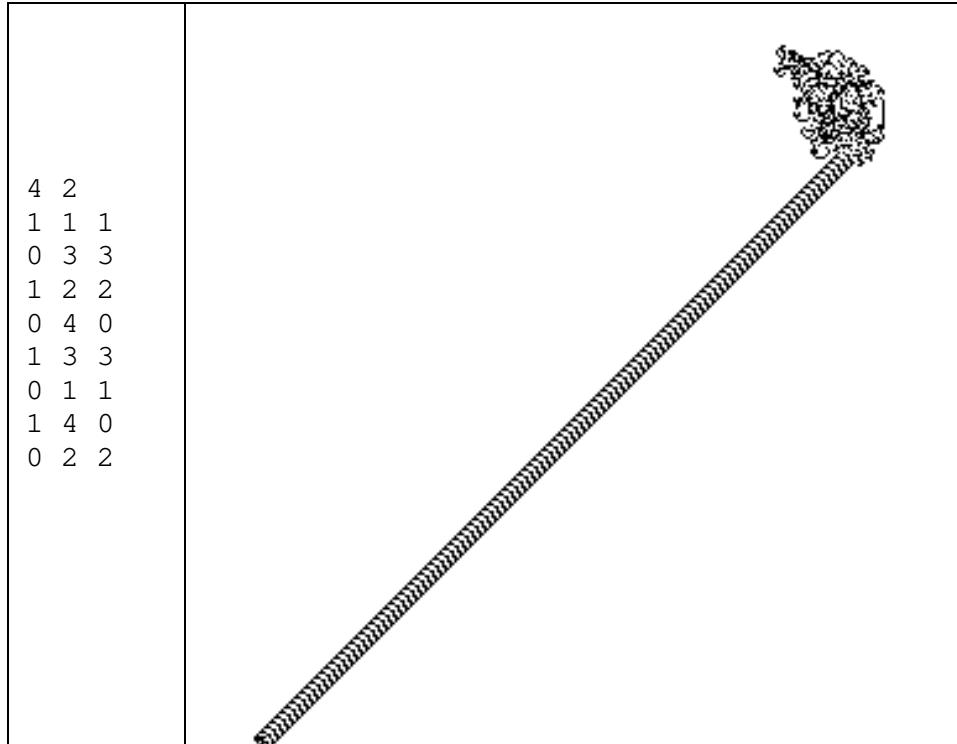
Pertanto, se questa macchina si venisse a trovare nello stato 3 sopra una cella di colore 0, allora cambierebbe il colore di tale cella in 1, si sposterebbe di una cella a destra (poiché la seconda componente della tripla è 4) e passerebbe nello stato 0... Si noti che qui le triple non prevedono stati “finali” ( $\geq 4$ ): infatti ciascuna tripla rimanda in uno degli stati “attivi” (da 0 a 3), e quindi resta esclusa la prima delle tre possibilità di arresto.

Definiamo poi un’altra classe, *Tavola*, con

- tre *campi privati*:
  - *tavola*, matrice (di MAX\_RIGHE righe per MAX\_COLONNE colonne) di *celle colorate* (come illustrato sopra);
  - *num\_righe* e *num\_colonne*, che dicono, rispettivamente, quante righe e colonne (a partire da 0) di tale matrice sono effettivamente considerate,
- un *costruttore*, che permetta di inizializzare in modo appropriato tali campi; per quanto riguarda le celle, allo scopo di fare qualche prova già significativa è sufficiente assegnare il colore 0 a tutte quelle usabili,
- e due *metodi*:
  - uno, con prototipo **void** *applica* (MdT2D *m*, **int** *i*, **int** *j*), che inneschi sulla tavola la computazione della macchina *m* (nello stato 0) a partire dalla cella di riga *i* e colonna *j* (purché sia dentro alla tavola!) e termini quando si verifica almeno una delle condizioni già illustrate sopra (abbiamo ipotizzato che la macchina riconosca i colori iniziali delle celle);
  - un altro metodo che permetta di visualizzare la tavola (e/o di salvarla su *file*) come immagine bidimensionale (anche soltanto mappando i colori delle celle su una scala di grigi).

(In C++, il costruttore di copia di una classe provvede a inizializzare un parametro di quel tipo classe con una copia dell’argomento: si vedano i due capitoli successivi.)

Come primo esempio di risultato ottenuto, mostriamo l'immagine prodotta dall'applicazione della MdT2D relativa al caso sopra considerato, nota come *formica di Langton*, a una tavola quadrata  $300 \times 300$  inizialmente tutta bianca (0 = bianco, 1 = nero), a partire dalla cella di riga 47 e colonna 253 (le righe sono numerate progressivamente dall'alto e le colonne da sinistra, sempre partendo da zero).

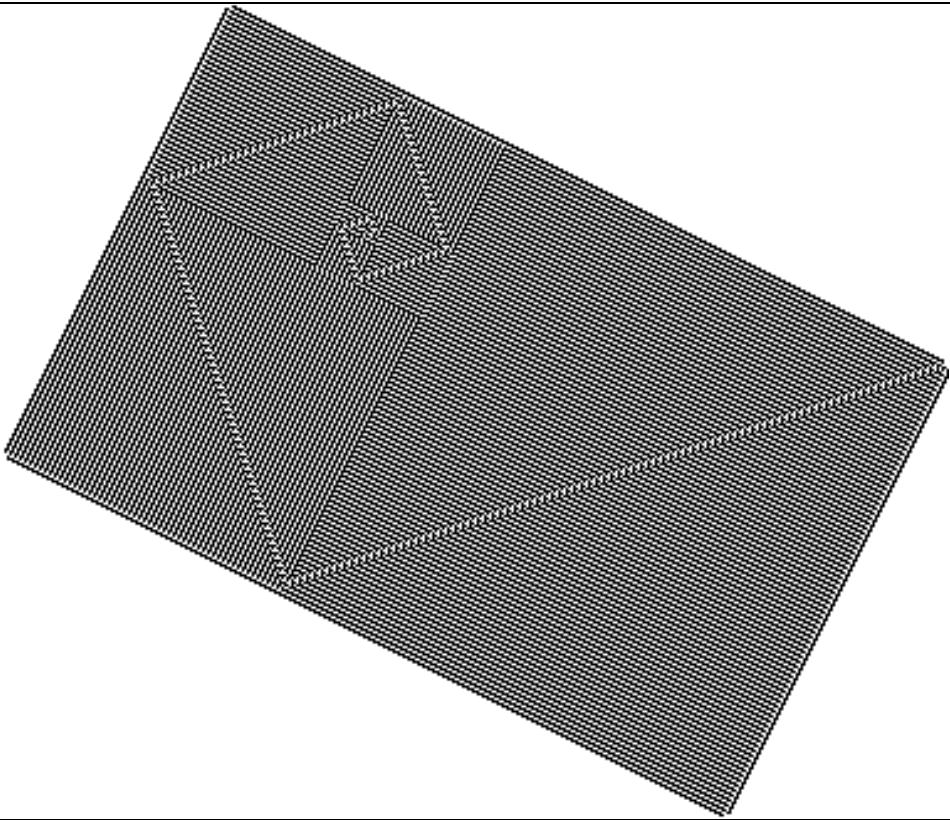


Dopo aver disegnato una “nuvola” di quadratini, la macchina si dirige verso sud-ovest, lasciando lungo il suo percorso la trama di un’intricata struttura: una improvvisa risolutezza dopo un lungo vagabondare senza meta, in apparenza casuale ma in realtà assolutamente deterministico! In un qualche punto del processo, compiuti poco più di 10000 passi, una certa configurazione fa sì che la macchina entri in una successione ripetitiva di movimenti che genera quella particolare struttura, che proseguirebbe indefinitamente... Nel caso illustrato, invece, la macchina esce dalla tavola dopo aver compiuto 22173 passi.

Nel secondo esempio, abbiamo considerato una macchina che genera una configurazione “a spirale”, mettendo in evidenza le diagonali consecutive di quadrati disposti via via in maniera ordinata intorno al punto di partenza: come si può notare dalla prossima figura, pare proprio che le lunghezze dei lati di questi quadrati seguano la successione di Fibonacci!

La MdT2D in questione – che ha otto stati, tutti “attivi” – è specificata dal *file* di testo riportato sempre nel riquadro a sinistra, ed è applicata a una tavola  $330 \times 385$  inizialmente tutta bianca, a partire dalla cella di riga 93 e colonna 150; l’immagine è ottenuta in 160741 passi, dopodiché anche in questo caso la macchina esce dalla tavola.

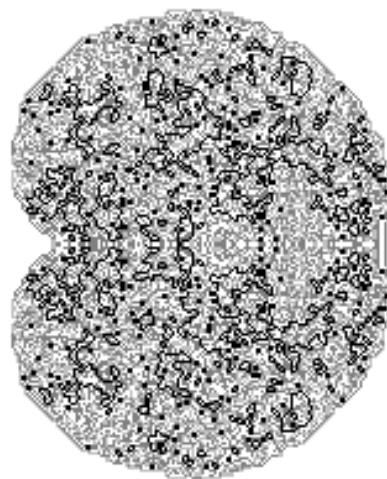
8	2	
1	2	1
0	1	4
1	3	2
0	2	5
1	4	3
0	3	6
1	1	0
0	4	7
1	4	3
1	4	3
1	1	0
1	1	0
1	2	1
1	2	1
1	3	2
1	3	2



Se cambiassimo da 1 in 0 la prima componente delle triple nona, undicesima, tredicesima e quindicesima, otterremmo un altro disegno che presenta delle regolarità...

Nell'ultimo esempio di MdT2D che proponiamo, la macchina si è fermata per aver raggiunto il massimo numero di passi, fissato a 9 milioni: il contorno dell'immagine risultante ricorda una cardioide.

4	4	
1	4	3
2	4	3
3	2	1
0	2	1
1	1	0
2	1	0
3	3	2
0	3	2
1	2	1
2	2	1
3	4	3
0	4	3
1	3	2
2	3	2
3	1	0
0	1	0



Questa volta i colori sono quattro (tradotti in altrettante tonalità di grigio: 0 = bianco, 1 = grigio chiaro, 2 = grigio scuro, 3 = nero), la tavola inizialmente bianca è quadrata  $255 \times 255$  e la macchina parte al centro.

Sarebbe interessante prevedere l'opzione di osservare sullo schermo l'immagine man mano che si forma, mentre la macchina esegue, a velocità adeguata, i propri passi di calcolo!

Concludiamo questo argomento raccomandando, come al solito, qualche ulteriore ricerca in rete<sup>20</sup> e rammentando l'esistenza – e talvolta la convenienza – di una descrizione alternativa delle MdT2D che specifica il “movimento relativo” della macchina, così come potrebbe fare la *tartaruga Logo*, dotata di una testa e quindi di un orientamento; ad esempio: 0 = nessuna rotazione, 1 = si gira a sinistra, 2 = si gira a destra, 3 = si volta indietro... e, dopo l'eventuale rotazione, avanza di una cella.

Spesso questa descrizione risulta più compatta: nel caso della spirale di Fibonacci vista sopra, si riduce a due soli stati, con le quattro triple

1 1 0, 0 0 1, 1 2 0, 1 2 0

mentre la formica di Langton, nella sua descrizione alternativa, ha addirittura un solo stato – e, per giunta, è pure capace di computazione universale, com'è stato provato nel 2000.

Chi fra i lettori riuscirà a costruire il programma seguendo le indicazioni date in questo paragrafo, sarà certamente in grado di generare – e poi provare – in modo automatico alcune macchine, che abbiano un numero limitato di stati e di colori, e magari rispettino certe simmetrie. Gli consigliamo anche di far partire la formica di Langton all'interno di un rettangolo di celle nere o di spargere dapprima sulla tavola, a caso, una discreta quantità di celle nere, procedendo poi per un buon numero di passi...

Quesiti ispirati dalla tartaruga si trovano nel libretto *Kangourou dell'Informatica* del 2015 (“Rettangoli” e “L'ape robotica”) e pure nella gara finale del 2014 (“Minirobot”), mentre nel libretto del 2013 è compreso “Contenitori Turing”, un simpatico problema sulle classiche macchine di Turing in una dimensione, ivi illustrate.

**Parole chiave:** automa cellulare, spazio e tempo discreti, regole di transizione, configurazione iniziale, reversibilità, modello di calcolo universale, macchina di Turing (in due dimensioni), matrici (*array* a due indici), paradigma di programmazione *object oriented*.

---

<sup>20</sup> Si può partire, ad esempio, dalle pagine <http://mathworld.wolfram.com/Turmite.html> e [http://www.mathpuzzle.com/MAA/21-2D%20Turing%20Machines/mathgames\\_06\\_07\\_04.html](http://www.mathpuzzle.com/MAA/21-2D%20Turing%20Machines/mathgames_06_07_04.html) (la prima si trova nel sito curato da Wolfram, che è un'autentica miniera di informazioni per tutti gli appassionati della matematica o dell'informatica).

Non possiamo dimenticare che Stephen Wolfram è tra i creatori dell'ambiente di calcolo (simbolico) *Mathematica* e pure del *Progetto R*, che è un software libero specializzato per la statistica e la grafica.

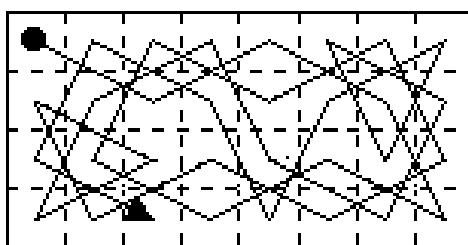
## 7. La corsa del cavallo sulla scacchiera

In questo primo capitolo sui rompicapi, o *puzzle*, ci occuperemo di quei giochi (a un solo giocatore) per i quali una soluzione (se c'è) è *tutta contenuta nello stato finale*, e non nella sequenza di mosse che è stata fatta per ottenerlo. Inoltre, ci limiteremo per ora a quei giochi in cui – grazie a un semplice accorgimento, se serve – non sussiste il rischio di incorrere in *cicli*, ossia di ritornare in uno stato già visitato durante la ricerca di una soluzione.

Ad esempio, appartengono a questa categoria di puzzle il completamento di uno schema di *Sudoku* (che, su griglie arbitrariamente grandi, è un problema complesso quanto trovare un *tour* ottimo per il commesso viaggiatore) e la disposizione di più regine su una scacchiera dimodoché non si minaccino, ma anche – tenendo traccia del percorso – il problema dell'uscita da un labirinto e quello del giro del cavallo: qui vedremo, in generale, com'è fatto un algoritmo di esaurimento per risolverli.

### In breve, una lunga storia.

Quello di far percorrere al cavallo le 64 case della scacchiera, tocando ciascuna casa esattamente una volta, è un problema antichissimo, presumibilmente coevo alla nascita di questo pezzo degli scacchi e del suo particolare movimento “a elle”. Tale problema, già studiato dai giocatori di scacchi arabi del IX secolo – come esercizio di tipo matematico-scacchistico è esposto infatti in numerosi antichi manoscritti, conservati presso diverse biblioteche del mondo, ad esempio a Istanbul e a Manchester – era pure noto al poeta indiano Rudrata, il quale usò un giro di cavallo (aperto) sulla metà superiore della scacchiera come motivo per comporre un poema in sanscrito, verso la fine di quello stesso secolo...<sup>21</sup>

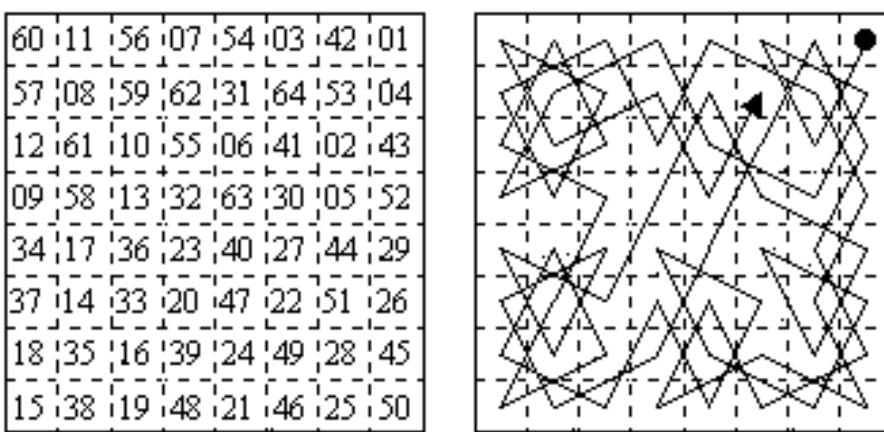


Come si può vedere dalla traccia lasciata dal cavallo, sarebbe bastato traslarne una copia nella metà inferiore della scacchiera, e prevedere un'ulteriore mossa nel mezzo che congiungesse le due parti, per ottenere un giro (aperto) sull'intera scacchiera! Una precisazione tecnica si rende subito opportuna: il giro si dice *chiuso* se la casa di arrivo è a distanza di salto di cavallo dalla casa di partenza (e su mezza scacchiera

<sup>21</sup> Raccomandiamo di approfondire le notizie, non soltanto storiche ma anche matematiche e geometriche, alla ricchissima pagina web <http://www.mayhemantics.com/t/t.htm> curata da George P. Jelliss, dalla quale abbiamo tratto alcuni diagrammi.

questo è impossibile), *aperto* in caso contrario. Sicché, se si trova un giro aperto, in realtà si trova anche quello che si sviluppa in senso opposto (dalla casa di arrivo a quella di partenza), più quelli ottenibili con rotazioni e/o ribaltamenti della scacchiera<sup>22</sup>; ma se si trova un giro chiuso, il cavallo può iniziare a percorrerlo da una qualsiasi casa, ancora procedendo nell'uno o nell'altro dei due possibili sensi di marcia: non ricorda un po' il problema del commesso viaggiatore? Torneremo presto a riconsiderare la questione sotto questo aspetto.

Ora facciamo un breve *excursus* storico, a cominciare dalla più antica soluzione ragionevolmente databile, che ci è pervenuta grazie a un manoscritto della metà del Trecento; eccola riprodotta:



Il diagramma in forma numerica sopra riportato (a sinistra) fu composto a Baghdad, intorno all'anno 840, dal filosofo al-Adli ar-Rumi, che scrisse un libro sul *Shatranij*: con questo nome gli Arabi chiamarono il gioco – da cui hanno avuto origine gli attuali scacchi – che appresero dai persiani probabilmente intorno alla metà del VII secolo. Come si vede nel diagramma a destra, ove il tracciato è completabile con un ultimo segmento tra la casa di arrivo e quella di partenza, si tratta di un giro chiuso, ma non simmetrico: ruotando il disegno di 180 gradi, non si sovrappone a sé stesso.

Una piccola digressione: al-Adli formulò uno stimolante problemino, con quattro cavalli su una scacchiera 3×3, reperibile – insieme con un giro del cavallo su una semiscacchiera (incluso pure, specchiato, nelle raccolte del *Civis Bononiae*) e oltre una settantina di problemi di scacchi secondo le regole antiche – in un codice cartaceo compilato dall'architetto forlivese Paolo Guarino nel 1512 (il manoscritto originale è conservato nella biblioteca pubblica di Cleveland, Ohio, dove si trova una delle più importanti collezioni al mondo di libri sugli scacchi).

La posizione iniziale è raffigurata nel diagramma che segue. I due cavalli bianchi devono prendere i posti dei due neri, e viceversa, nel minor numero di mosse.

<sup>22</sup> Si può dire quanti sono? Un cammino aperto sulla scacchiera 8×8 può presentare delle simmetrie? In uno dei suoi quesiti, contenuto nella storiella “Il campo di grano del fattore Lawrence”, il grande enigmista inglese Henry E. Dudeney – del quale parleremo nel prossimo capitolo – mostrò un bel giro aperto su una scacchiera 7×7 simmetrico rispetto al centro, che si sviluppa da uno degli angoli a quello opposto... E, approssimo, un giro chiuso su sei scacchiere 8×8 disposte sulle facce di un cubo!



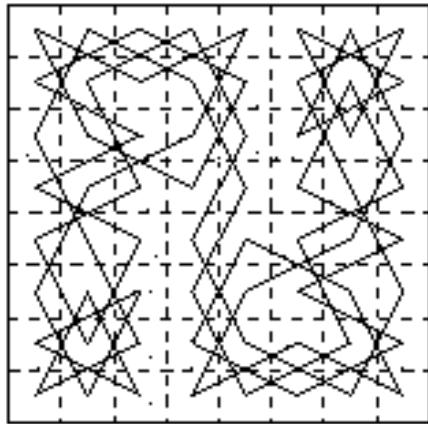
[*Suggerimento:* si scambieranno la posizione i cavalli agli angoli opposti, e ciascun cavallo farà quattro salti. Numerando i cavalli da 1 a 4 in senso orario a partire da quello nero in alto a sinistra, una soluzione prevede che dapprima ciascuno compia nell'ordine tanti salti quanti il proprio numero, dopodiché il cavallo 1 farà tre salti, 2 compirà gli altri due, e infine 3 farà l'ultimo salto. Si noti altresì che su una scacchiera  $3 \times 3$  un singolo cavallo piazzato in una casa laterale o d'angolo può visitare tutte le altre, tranne la centrale, e infine ritornare in quella di partenza tracciando esattamente una “stella” di otto mosse.]

Per quanto ne sappiamo, i moderni studi sul problema del giro del cavallo – che appassionò non solo molti scacchisti ma anche non pochi matematici – sono iniziati nella prima metà del XVIII secolo, in Francia, mancando tuttavia la conoscenza dei lavori di epoca medievale, ad eccezione forse del giro sulla semiscacchiera contenuto nell’opera del Guarino.

Nell’edizione del 1725 della già popolare raccolta di Jacques Ozanam *Récréations Mathématiques et Physiques*, furono pubblicati tre giri aperti dovuti ad altrettanti illustri matematici dell’epoca: Pierre Rémond de Montmort, Abraham de Moivre e Jean-Jacques de Mairan. Fu quest’ultimo, allora direttore dell’Accademia delle Scienze di Parigi, a consegnarli qualche tempo prima all’editore parigino Claude Jombert, che suddivise la nuova edizione (postuma) dell’opera di Ozanam in ben quattro volumi.

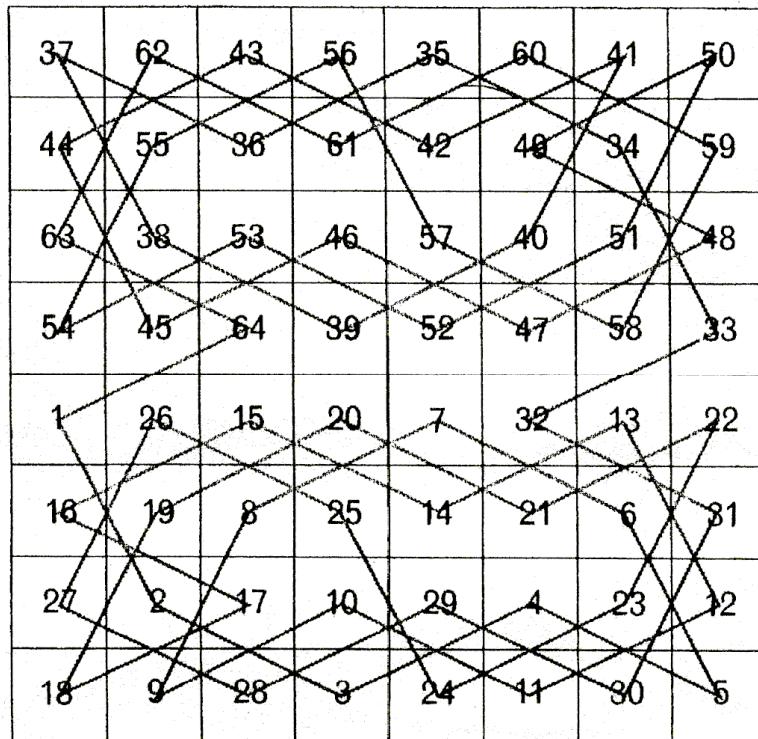
Il primo articolo sull’analisi del problema fu presentato dal più fecondo matematico del Settecento, Eulero, all’Accademia delle Scienze di Berlino, nell’anno 1759, e pubblicato poi nel 1766. Vi sono compresi, come esempi, cinque giri (chiusi) simmetrici: presentano infatti una simmetria rotazionale di ordine 2, vale a dire che il tracciato chiuso si sovrappone a sé stesso quando è ruotato di 180 gradi. Si tratta dei primi diagrammi con questa peculiarità che siano stati composti, se si esclude quello di Bhatta Nilakant-ha, il quale scrisse un’opera encyclopedica, contenente una sezione dedicata appunto agli scacchi, di datazione incerta, collocabile tra il 1600 e il 1700, ma giunta in Francia dalle Indie Orientali soltanto nel 1776.

Trovo che il disegno, in forma di spezzata chiusa, generato da questo giro “esotico”, sia estremamente evocativo, sicché lo riporto qui di seguito, senz’altro commento.

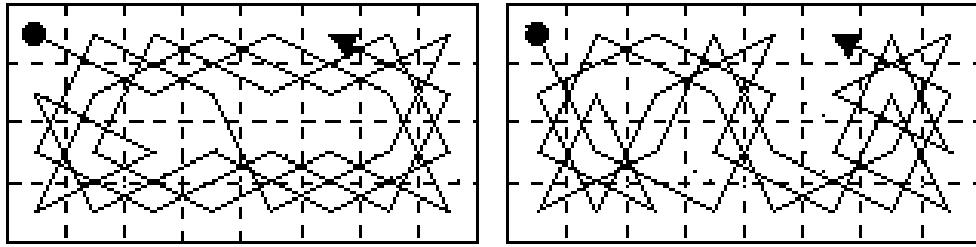


Comunque Eulero fu il primo ad accorgersi che la simmetria dei percorsi chiusi è caratterizzata da una particolare proprietà: la differenza tra i numeri presenti nelle case diametralmente opposte è sempre 32. Provate a mettere i numeri da 1 a 64 nel diagramma di Nilakant-ha, partendo da una casa a vostra scelta e procedendo nell'uno o nell'altro senso, e a verificare poi questa proprietà!

Nella stessa memoria, Eulero presentò altri cinque esempi di giri ottenuti per simmetria, prendendo un giro opportuno su una semiscacchiera e attaccandolo a una sua copia – ruotata di 180 gradi – mediante l'aggiunta di due mosse per chiudere il ciclo; eccone uno:



Il fatto curioso è che, già molto tempo addietro, erano conosciuti alcuni giri aperti su mezza scacchiera che avrebbero potuto servire a questo scopo: due di essi, che hanno le stesse case di partenza e di arrivo della metà inferiore del diagramma composto da Eulero, sono rappresentati qui di seguito.



Quello a sinistra (che segue il percorso di Guarino per le prime 18 mosse, ma poi se ne discosta) conclude una raccolta di studi stampata a Torino nel 1597, che costituisce il più antico testo italiano sulla teoria del gioco: il *Libro nel quale si tratta della Maniera di Giuocar'à Scacchi, Con alcuni sottilissimi Partiti*,<sup>23</sup> di Horatio Gianutio della Mantia, nobiluomo di probabile origine calabrese; quello a destra compare in un manoscritto fiorentino ancor di un secolo precedente. Ma nessuno dei due corrisponde alla metà inferiore di quello dianzi mostrato (benché il primo sia assai simile), né di uno degli altri quattro trovati da Eulero che presenta le case terminali speculari rispetto ai giri di Gianutio e del manoscritto fiorentino.

Con un nostro programma abbiamo calcolato che – non contando come diversi i tracciati orientati o ruotati o ribaltati – sono 986 i giri chiusi di questa natura, cioè ottenuti congiungendo due percorsi identici su una semiscacchiera, l'uno ruotato rispetto all'altro di 180 gradi; in particolare, le case terminali si trovano sempre sulla stessa traversa (o riga) in alto (considerando la metà inferiore della scacchiera): in prima e sesta colonna (come sopra) per 502 giri, in seconda e quinta colonna per 276 giri, e in terza e quarta colonna per 208 giri. Non è possibile, invece, ottenerne per ribaltamento!

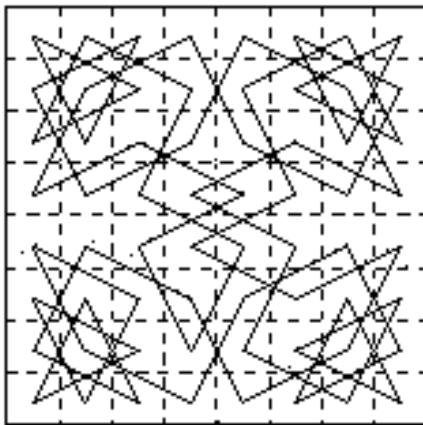
Ancora italiana è la prima pubblicazione dove i giri del cavallo – in tutto 19, tra cui i tre apparsi nella raccolta di Ozanam – sono presentati non soltanto in forma numerica da 1 (= casa di partenza) a 64 (= casa di arrivo), ma anche in forma di eleganti diagrammi «in lineette tirate su lo Scacchiere», non specificando il senso di marcia, e per di più – nel caso dei 6 giri chiusi – completando il disegno con l'ultimo segmento, senza indicare le case di partenza o di arrivo: la *Corsa del Cavallo per tutt'i scacchi dello Scacchiere*, stampata a Bologna nel 1766, presso la tipografia che era stata di Lelio dalla Volpe.

Alla pagina 197 ne riproduciamo il frontespizio, per gentile concessione della Biblioteca dell'Archiginnasio in Bologna.

Alexandre-Théophile Vandermonde presentò un proprio contributo all'Accademia delle Scienze di Parigi nel 1771, pubblicato tre anni dopo: partendo da uno pseudo-ciclo con simmetria biassiale, formato da quattro sotto-cicli che toccano 16 case ciascuno, costruì un giro chiuso, dal tracciato suggestivo, che tuttavia non preserva la simmetria, come si può notare dalla figura che segue.

---

<sup>23</sup> “Partiti” erano detti i problemi.



## Come si può trovare una soluzione?

In molti cercarono di escogitare una semplice regola che, seguita passo dopo passo, permettesse di generare uno o più giri. Ad esempio, nello stesso manoscritto medievale che riporta il giro di al-Adli se ne trova un altro (aperto, attribuito ad Ali ibn Mani, presumibilmente coevo) che, analogamente a quello ideato da de Moivre nel 1722, segue la regola di partire da un angolo e di tenersi, quanto più possibile, vicino ai lati della scacchiera, ruotando sempre nello stesso senso, prima di passare a coprire l'area centrale.

Riveste una notevole importanza storica, ed è tuttora la più conosciuta, la regola altrettanto semplice ma più rigorosa e generale, formulata dal matematico tedesco H. C. von Warnsdorf nel 1823 (*Des Rösselsprunges einfachste und allgemeinste Lösung*, Schmalkalden): ad ogni passo, si deve muovere il cavallo in una casa non ancora visitata che permetta il *minor numero* di possibili mosse successive, vale a dire in cui il cavallo controlli il minor numero di case non ancora visitate. In caso di parità tra diverse case, la scelta fra queste può essere fatta arbitrariamente (mutuando l'espressione dal gergo tennistico, oggi gli inglesi dicono *random tie-break*, ossia spareggio casuale)... Così almeno sostenne Warnsdorf.

In realtà, questo è un metodo *euristico* (cioè che *cerca di trovare* una soluzione basandosi su regole empiriche, dettate dal buon senso) plausibile per il problema in esame (infatti tende a percorrere dapprima i bordi, ossia le case più isolate, che, lasciate per dopo, potrebbero più facilmente diventare delle trappole), ma che non garantisce il reperimento di una soluzione in tutti i casi (cioè qualunque sia la scelta fatta in caso di parità).

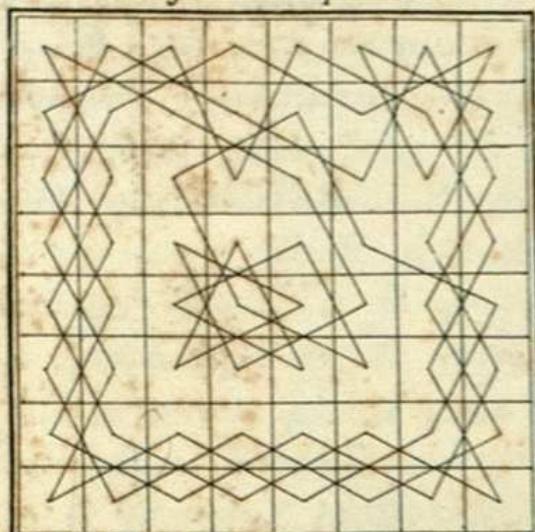
Innanzi tutto, si dovrebbe parlare del *minor numero positivo* di possibili continuazioni (a meno che non si tratti della mossa finale), altrimenti il rischio di finire in un “vicolo cieco” aumenta sensibilmente! Ma non basta ancora: se si è massimamente sfortunati si faranno soltanto 45 mosse, dopodiché si finirà in un vicolo cieco, come si può osservare ad esempio nello schema alla pagina 198 (prendendo come casa di partenza una delle quattro case centrali della scacchiera, si possono presentare 10 differenti di queste eventualità).

C O R S A  
D E L  
C A V A L L O  
P E R T U T T' I S C A C C H I  
D E L L O  
S C A C C H I E R E.



*O curas hominum! quantum est in rebus inane.  
Quandoque tamen et sapientibus placent.*

*Et semel a quovis*



*Cuncta attingit Equus.*

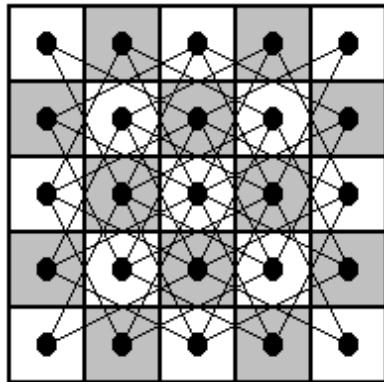
In Bologna per Lelio dalla Volpe 1766. Con licenza de'Superiori.

-	38	25	20	13	16	3	18
24	21	46	39	2	19	12	15
37	40	23	26	45	14	17	4
22	27	42	1	-	11	44	-
41	36	31	-	43	-	5	10
28	-	34	-	8	-	-	-
35	32	-	30	-	-	9	6
-	29	-	33	-	7	-	-

Scelte sfortunate possono comunque essere fatte (e manifestare il loro effetto negativo dopo un maggior numero di mosse) qualunque sia la casa di partenza sulla scacchiera  $8 \times 8$ . L'algoritmo di Warnsdorf è stato provato su scacchiere  $n \times n$  ed è stato appurato che la probabilità di insuccesso aumenta piuttosto rapidamente con  $n$ , quando  $n$  supera 50; ciò è anche dovuto al fatto che tende a cercare soluzioni in un sottoinsieme estremamente piccolo dell'insieme – enorme! – di tutte quante le soluzioni.

L'aspetto allettante resta la complessità rispetto al tempo, che è dell'ordine di  $n^2$  – ciò significa che, all'aumentare di  $n$ , il tempo impiegato dall'algoritmo cresce approssimativamente in modo proporzionale al quadrato di  $n$ , cioè al numero di case della scacchiera – ed è ovvio che a una complessità di ordine inferiore è impossibile scendere, data la natura del problema in sé che prevede la visita di tutte le case: senza dubbio un bel risparmio a confronto del metodo di “forza bruta” che le tenta tutte, praticabile – pur con l'ausilio del computer – soltanto per valori di  $n$  assai modesti!

Inoltre, la regola di Warnsdorf può essere immediatamente generalizzata per la ricerca di un *cammino hamiltoniano* (cioè che tocca una e una sola volta ciascun nodo) in un grafo qualsiasi (problema che, nella sua formulazione decisionale, fa parte della classe degli NP-completi, come si è detto): a meno che non si possa passare direttamente all'ultimo nodo, si passa a uno dei nodi adiacenti non ancora visitati, scelto tra quelli con il minor numero (positivo) di archi in uscita verso nodi a loro volta non ancora visitati – sicché tende a visitare prima i nodi “meno connessi”. In effetti, il problema del giro del cavallo ne costituisce un caso particolare: si tratta di trovare un cammino hamiltoniano sul grafo non orientato i cui nodi e archi corrispondono rispettivamente alle case della scacchiera e alle mosse lecite per un cavallo. Ad esempio, nel caso di una scacchiera  $5 \times 5$ , che è la più piccola scacchiera quadrata per cui – a partire da certe case, ma non da tutte! – il problema ammette soluzione, il grafo ha 48 archi (non orientati, cioè percorribili in entrambi i sensi), come si evince dalla seguente figura.



Nel caso  $8 \times 8$  gli archi sono 168; in generale, per  $n = 3, 4, 5, 6, \dots k$ , gli archi sono 8, 24, 48, 80, ...  $4 \cdot (k - 2) \cdot (k - 1)$ , cioè 8 volte i *numeri triangolari* 1, 3, 6, 10, ... che sono quelli della forma  $m \cdot (m + 1)/2$ , con  $m \geq 1$ . I giri chiusi corrispondono ai *cicli hamiltoniani* – quelli da considerare nel problema del commesso viaggiatore! – in quanto la mossa finale porta il cavallo in una casa che si trova alla distanza di una mossa lecita dalla casa di partenza.

In epoca più recente sono stati apportati dei miglioramenti all'algoritmo di Warnsdorf, proprio pensando alle applicazioni al caso più generale.

Si cercarono anzitutto delle regole di spareggio ragionevoli. Nel 1967 Ira Pohl introdusse questa: se più mosse hanno lo stesso minor numero di possibili continuazioni, allora si fa ancora un passo, andando a scegliere tra quelle che a loro volta hanno una mossa successiva col più piccolo numero (positivo) di possibili continuazioni (a meno che non si raggiunga la fine del giro). In caso di ulteriore parità, si sceglie casualmente.

Nel 1999 Arnd Roth propose quest'altra, più semplice, per il giro del cavallo: in caso di parità, si sceglie la casa più lontana (in senso euclideo) dal centro della scacchiera, e di nuovo in maniera casuale se ci fosse ancora parità.

Entrambe queste regole garantiscono sempre l'ottenimento di una soluzione sulla scacchiera  $8 \times 8$ , ma possono fallire su scacchiere sufficientemente grandi!

Nella regola di Roth non va bene considerare la “distanza Manhattan” (che può fallire quando il cavallo parte da una delle quattro case centrali), ancor peggio il numero minimo di mosse che un Re dovrebbe fare per giungere dal centro alla casa in questione; c’è invece un modo efficace per “approssimare” con i numeri naturali la distanza euclidea dal centro, com’è illustrato nello schema seguente:

9	8	7	6	6	7	8	9
8	5	4	3	3	4	5	8
7	4	2	1	1	2	4	7
6	3	1	0	0	1	3	6
6	3	1	0	0	1	3	6
7	4	2	1	1	2	4	7
8	5	4	3	3	4	5	8
9	8	7	6	6	7	8	9

Può rivelarsi un esercizio assai istruttivo la realizzazione dell'algoritmo di Warnsdorf-Roth per mezzo di un programma scritto in uno dei linguaggi noti. Qualche ricercatore ha escogitato metodi ancor più fini, ad esempio combinare in modo appropriato differenti ordinamenti (nelle diverse regioni della scacchiera) delle mosse da tentare... ma nessuno sa con certezza se funzioneranno sempre, su scacchiere comunque grandi!

È doveroso notare che sia la regola di Warnsdorf sia le sue varianti, sebbene si pongano l'obiettivo di *generare* giri del cavallo, non sono in realtà immuni da una sorta – seppur limitata – di *backtracking*, la tecnica di esplorazione in avanti e di ritorno sui propri passi in caso di fallimento, che viene impiegata nella ricerca *esauriente* – vale a dire “a tappeto” – di una o di tutte le soluzioni. Vedremo presto come questa tecnica può essere realizzata, sebbene la sua applicazione a uno specifico problema possa spesso rivelarsi proibitiva a causa dell'eccessivo tempo richiesto: anche questo è pur sempre un esercizio utile, anzi raccomandabile a chiunque intenda studiare un po' d'informatica!

Ricordiamo ancora che, per la ricerca di un cammino hamiltoniano su un grafo, sono stati messi a punto altri algoritmi euristicci – la cui spiegazione, non certo facile, esula dai nostri scopi – sempre con complessità temporale lineare nel numero dei nodi del grafo: alcuni sono basati su reti neurali, alcuni procedono seguendo criteri simili a quello adottato da Eulero quando aveva affrontato il problema del giro del cavallo, scomponendo la scacchiera in parti più piccole (eventualmente non quadrate) per le quali siano note soluzioni. In particolare, per scacchiere  $n \times n$  (o  $m \times n$ , purché ammissibile) arbitrariamente grandi, sono stati ideati algoritmi di ordine  $n^2$  (o  $m \cdot n$ , dunque lineari nell'area) che trovano giri del cavallo, aperti o chiusi, operando per scomposizione e ricomposizione (possibilmente in parallelo).

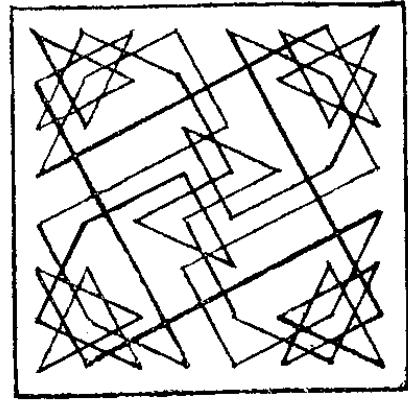
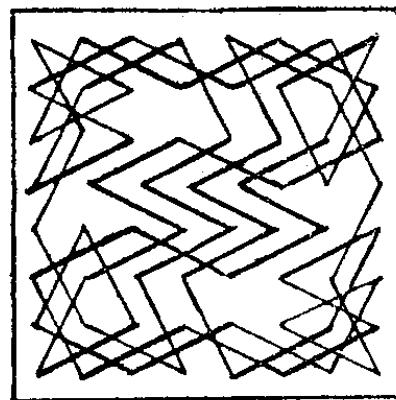
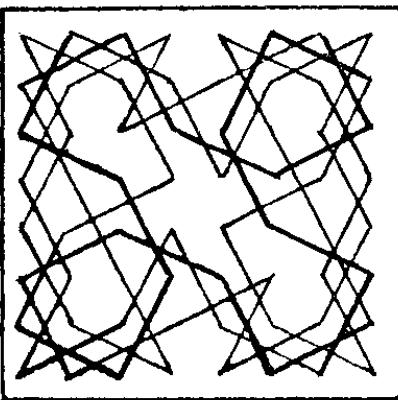
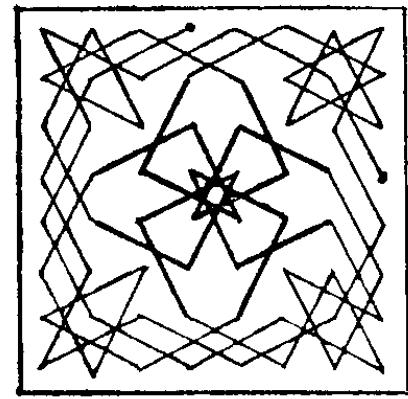
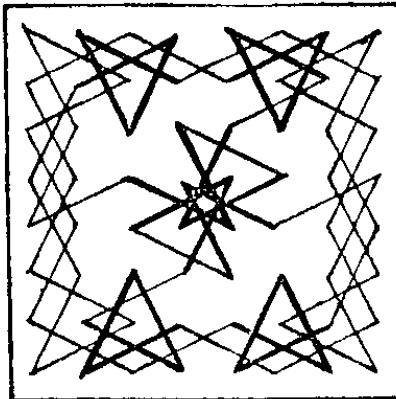
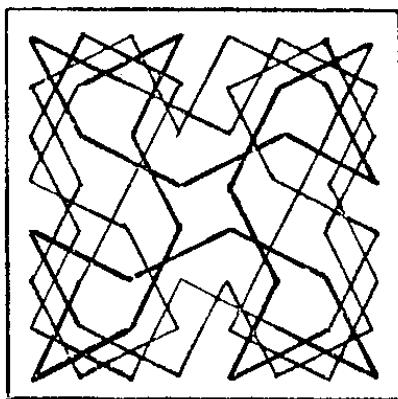
In genere, un algoritmo euristico per una certa classe di problemi giunge, salvo in rari casi, a fornire una soluzione ammissibile (se c'è), magari non la migliore; come abbiamo visto nel primo e nel quarto capitolo, certi metodi arrivano anche a dare una garanzia di approssimazione, ovvero una misura di quanto la soluzione trovata può essere lontana dalla migliore. I metodi più semplici usano tecniche *greedy*, cioè “ingorde” (che sono *costruttive*) oppure tecniche basate su ricerche locali (che rientrano nelle cosiddette *tecniche evolutive*).

## Quante sono le soluzioni?

Senza considerare alcuna simmetria, il numero esatto di soluzioni sulla scacchiera  $8 \times 8$  è stato calcolato nel 2014, ed è risultato poco meno di  $2 \cdot 10^{16}$ . Il numero di giri *geometricamente distinti* (vale a dire contando come uno soltanto i giri ottenuti l'uno dall'altro mediante rotazioni e/o ribaltamenti del tracciato non orientato) è  $1/16$  di tale numero, dunque dell'ordine di  $10^{15}$  (un milione di miliardi). Da tempo si sapeva con precisione quanti tra questi sono i giri chiusi, prescindendo pure sia dalla casa di partenza, sia dall'orientamento o senso di percorrenza: non arrivano a 1700 miliardi, dei quali sono simmetrici poco più di 600 mila... Trovare e disegnare anche soltanto

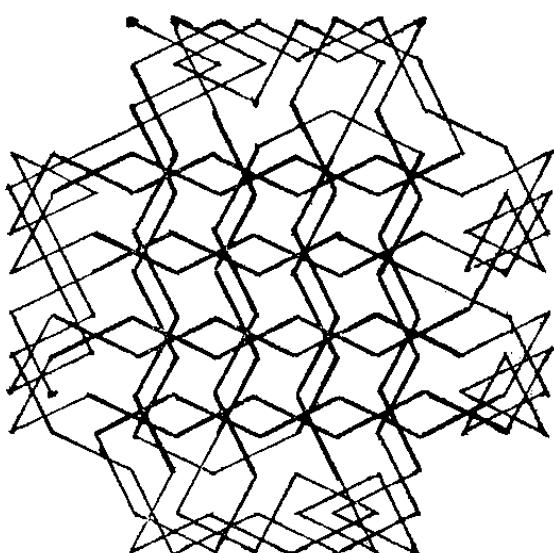
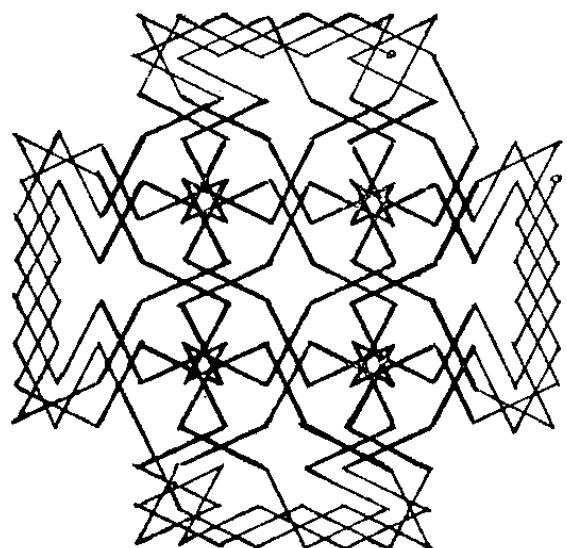
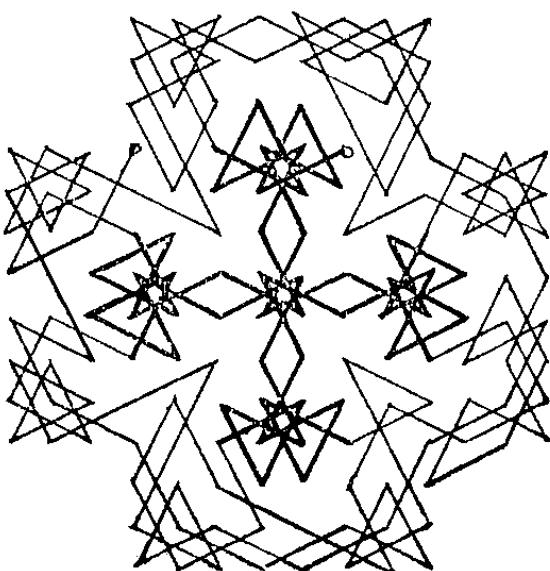
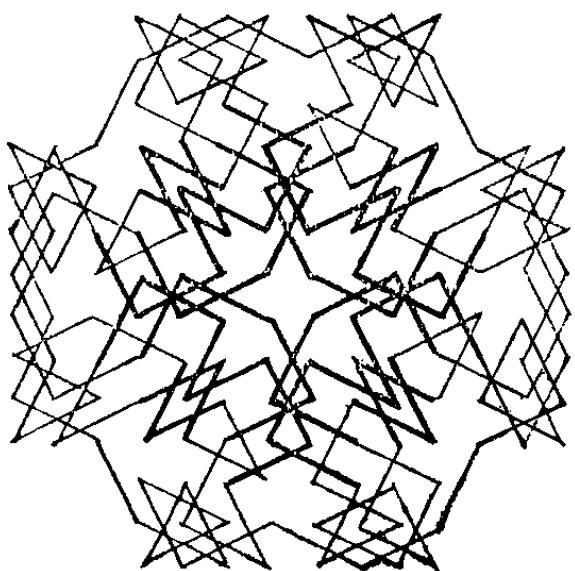
questi ultimi è un discreto compito perfino per un veloce elaboratore elettronico! Il problema del giro del cavallo ammette soluzioni per tutti gli  $n \geq 5$ , ma esistono giri chiusi se e soltanto se  $n$  è pari; è ovvio che se  $n$  è dispari allora lo è anche il numero di case della scacchiera, e quindi un giro chiuso è impossibile perché la casa dove si trova il cavallo cambia colore ad ogni mossa. Con il programma che impareremo presto a progettare, che enumera tutte le soluzioni, abbiamo verificato che sulla scacchiera  $6 \times 6$  i giri chiusi non orientati – *senza* però considerare rotazioni né ribaltamenti – sono 9862: in quante classi di equivalenza sono ripartiti?<sup>24</sup> Con un simile programma, non è consigliabile tentare il calcolo di quelli sulla scacchiera  $8 \times 8$ , il cui numero già supera i 13 mila miliardi!

È davvero stupefacente la quantità di soluzioni che si conoscevano oltre un secolo fa. Edward Falkener, nel suo bellissimo libro *Games Ancient and Oriental and How to Play Them*, pubblicato in prima edizione a Londra nel 1892, diede una raccolta di forme di giri sia aperti sia chiusi (in prevalenza non simmetrici), anche su una scacchiera non rettangolare ottenuta attaccando a ciascuno dei quattro lati un rettangolo  $8 \times 3$ : come osserva l'autore, su questo tavoliere i disegni sono ancor più regolari e sorprendenti nel mostrare l'ordine armonioso della marcia del cavallo; ce ne possiamo rendere conto anche noi, da alcuni esempi che qui ho voluto riprodurre (i giri chiusi da me scelti sono tutti simmetrici).



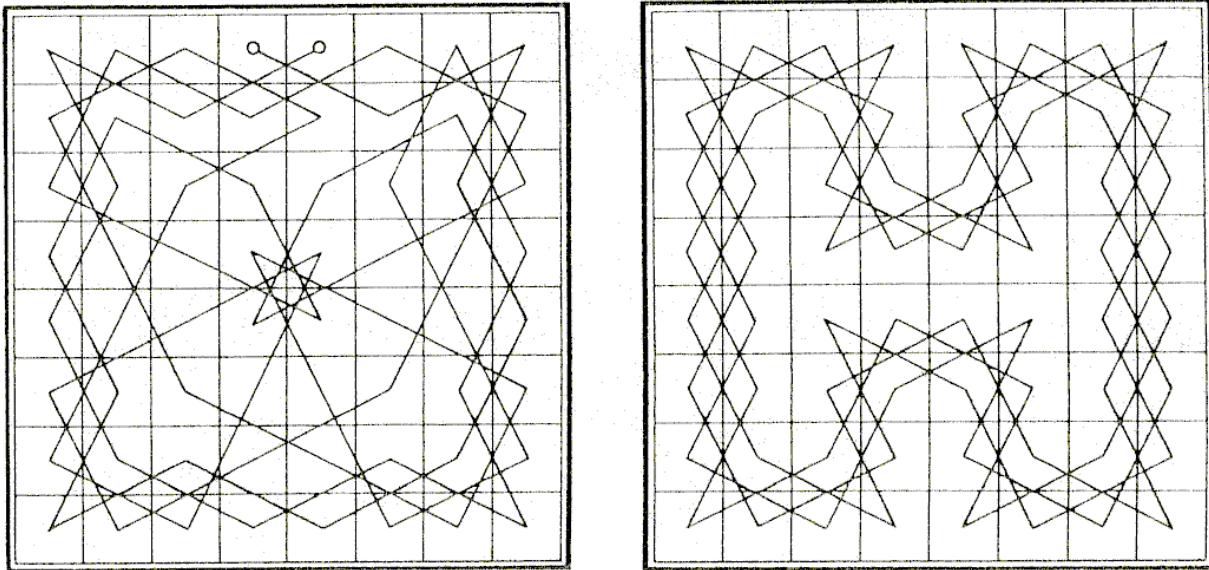

---

<sup>24</sup> Ve ne sono 1245 geometricamente distinti, di cui 17 con simmetria rotazionale di ordine 2, ma 5 con simmetria rotazionale di ordine 4, non ottenibile sulla scacchiera  $8 \times 8$ : per questi ultimi, basta una rotazione di 90 gradi affinché il tracciato chiuso si sovrapponga a sé stesso.



E, per concludere, alla pagina successiva in alto riportiamo ancora due esempi di bei tracciati: a sinistra uno aperto, a destra uno chiuso e simmetrico, e non solo rispetto al centro ma anche ai due assi orizzontale e verticale! Quanti altri ce ne saranno con questa notevole proprietà?

La risposta, che richiede un minimo di attenzione, sarà data alla fine del capitolo...



## Giri semimagici.

A iniziare dagli anni '20 dell'Ottocento fu tentata l'impresa di trovare un giro di cavallo "magico" sulla classica scacchiera  $8 \times 8$ : ciò vuol dire che la somma dei numeri su ciascuna traversa, su ciascuna colonna e su ciascuna delle due diagonali maggiori deve essere la stessa, ossia 260; infatti, com'è noto, la somma dei numeri da 1 a 64 è  $64 \cdot 65 / 2 = 2080$ , e va divisa in parti uguali fra le otto traverse (o colonne). Il primo che scoprì un giro *semimagico* (mancava il requisito sulle diagonali) fu William Beverly, e lo pubblicò nell'agosto del 1848 su *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*.

1	30	47	52	5	28	43	54
48	51	2	29	44	53	6	27
31	46	49	4	25	8	55	42
50	3	32	45	56	41	26	7
33	62	15	20	9	24	39	58
16	19	34	61	40	57	10	23
63	14	17	36	21	12	59	38
18	35	64	13	60	37	22	11

Come si vede, non è chiuso ma ha altre proprietà degne di nota:

- in ciascuno dei quattro quadranti in cui può essere suddivisa la scacchiera, la somma dei numeri è 520 e in ciascuna riga e colonna è 130;

- scomponendo poi ciascun quadrante in quattro quadranti  $2\times 2$ , la somma dei numeri in ciascuno di essi è ancora 130.

Se si numerano le case in ordine inverso, seguendo il percorso a ritroso, si ottiene un quadrato semimagico con tutte le medesime proprietà.

L'anno successivo fu trovato il secondo, chiuso e simmetrico: lo pubblicò Carl Wenzelides su *Schachzeitung*.

2	11	58	51	30	39	54	15
59	50	3	12	53	14	31	38
10	1	52	57	40	29	16	55
49	60	9	4	13	56	37	32
64	5	24	45	36	41	28	17
23	48	61	8	25	20	33	42
6	63	46	21	44	35	18	27
47	22	7	62	19	26	43	34

La corrispondente, graziosa figura geometrica è tra quelle riprese dal libro di Falkener nel paragrafo precedente (alla pagina 201): ed è facile trovarla fra le cinque col percorso chiuso!

Una parentesi curiosa: proprio in quegli anni, in Francia, comparvero i primi *crittogrammi a giro di cavallo*, in cui le 64 parole o sillabe di una frase sono stampate nelle case della scacchiera e possono essere lette nella giusta successione percorrendo appunto un giro di cavallo. Per rendere il puzzle più interessante, di solito non è nemmeno segnalata la casa di partenza!

Da allora la ricerca di un giro magico divenne una grande sfida...

Saltiamo, come il cavallo, fino ai nostri giorni. Si sapeva da tempo che giri magici non sono possibili su scacchiere  $n\times n$  con  $n$  dispari e che invece sono possibili su tutte le scacchiere  $4k\times 4k$  con  $k > 2$ . Ma non si sapeva se ne esistessero sulla solita scacchiera  $8\times 8$  fino all'agosto del 2003: a distanza di 155 anni esatti dalla pubblicazione del primo giro semimagico, l'enumerazione completa delle possibilità, fatta mediante un software scritto da Jean-Charles Meyrignac, terminò con una risposta negativa. Furono trovati soltanto 140 giri semimagici (se si contano i cammini, lasciandoli tutti comunque aperti, modulo simmetrie), di cui 63 chiusi (ma non tutti geometricamente distinti, qualora si richiuda il percorso con l'ultimo segmento) e 77 aperti; essi sono ancora disponibili in rete, al seguente indirizzo:

<http://magictour.free.fr/>.

## Un algoritmo di esaurimento.

Lasciando come compito ai lettori interessati la realizzazione di un programma che trovi una soluzione al problema del giro del cavallo mediante la regola di Warnsdorf, magari con la variante di Roth, passiamo ad illustrare – più in generale – il progetto di un programma (in linguaggio C++) che risolva un puzzle seguendo un’idea più ovvia e “brutale”: esplorare via via tutte le possibili sequenze di mosse, fino a trovarne una risolutiva, se c’è; se non ce n’è alcuna, segnalare questo evento con un opportuno messaggio e fermarsi comunque. Volendo, trovata una soluzione, si può continuare, allo scopo di cercarne altre.

Una tal procedura può fare una ricerca in profondità (*depth first*) sul grafo orientato delle possibili transizioni da uno stato del gioco a un altro, impiegando la tecnica del *backtracking*: così sfrutta lo *stack* del sistema, e il codice sorgente (ricorsivo all’interno di un ciclo) è assai conciso.

L’idea del *backtracking* è semplice, e implicitamente l’abbiamo già usata nella specifica della procedura ricorsiva TSP per risolvere il problema di ottimizzazione del commesso viaggiatore: durante l’esplorazione, se giungiamo in un vicolo cieco, torniamo indietro al primo bivio e tentiamo un’altra strada. L’unico pericolo consiste nei “circoli viziosi”: se esiste la possibilità di cadervi, allora per uscirne dobbiamo ricordare gli scenari già visti – il mito di Arianna e la favola di Pollicino insegnano pur qualcosa! Nel caso del giro del cavallo non sussiste questa eventualità (proprio perché numeriamo le case della scacchiera via via che sono percorse dal cavallo), ma per tanti altri giochi sì, come vedremo nel prossimo capitolo.

Adesso procediamo così: illustriamo un algoritmo generale per stampare (ed eventualmente conservare) soltanto il *primo* stato finale di successo che il procedimento di calcolo sia riuscito a individuare (se c’è), poi lo dettagliamo per il particolare problema del giro del cavallo, e infine lo modifichiamo per trovare e stampare a uno a uno *tutti* gli stati finali di successo.

Definiamo pertanto una *classe* *puzzle*, ciascuna istanza della quale sarà un *first-class object* adatto a rappresentare, in ogni momento della sua “*vita*”, uno qualsiasi tra tutti i possibili stati del puzzle; a tale oggetto dovranno essere applicabili le operazioni lecite, a seconda dello stato rappresentato.

```
class puzzle {
    private:
        // dichiarare qui i campi che rappresentano lo stato del gioco
    public:
        // l'interfaccia verso l'esterno è costituita
        // dalle operazioni, o metodi:
        puzzle(); // costruttore senza parametri: servirà in seguito
        puzzle (const puzzle & rhs); // costruttore di copia
        puzzle & operator = (const puzzle & rhs); // overloaded
        bool success () const; // lo stato è finale (è una soluzione)
        bool is_legal (int i) const; // la mossa i è lecita
        void make_move (int i); // esegue la mossa i (se è lecita)
        void print () const; // stampa in output lo stato e va a capo
```

```

// aggiungere le funzionalità (o prototipi) degli altri metodi
// che servono:
// search_one e/o search_all,
// un costruttore con parametri per rappresentare lo stato
// iniziale (se questo può essere scelto dall'utente),
// il distruttore ...
};


```

Vediamo subito come può essere scritto un programma di prova, avendo anzitutto incluso <iostream>:

```

int main () {
    puzzle p; // istanza statica inizializzata dal primo costruttore
    // oppure p(...); p rappresenta lo stato iniziale del gioco
    if (! p.search_one())
        std::cout << "Non esiste soluzione." << std::endl;
    return 0;
}

```

Per certi giochi, il costruttore invocato potrà avere degli argomenti: si pensi, in particolare, alla posizione iniziale del cavallo sulla scacchiera.

Il metodo a valori *booleani* search\_one cerca una soluzione a partire dallo stato rappresentato dall'oggetto al quale è applicato, e ha come effetto collaterale (*side-effect*) la stampa in *output* della soluzione trovata, ma soltanto nel caso in cui abbia avuto successo (cioè quando restituisce **true**):

```

bool puzzle :: search_one () {
    if (success ()) {
        // trovata una soluzione!
        print ();
        return true;
    }
    for (int i = 1; i <= N; i++) // enumera le mosse da tentare
        if (is_legal (i)) {
            // se la mossa numero i è lecita, la esegue su una copia
            // dello stato corrente ...
            puzzle c = *this;
            c.make_move (i);
            // ... e se da qui arriva al successo,
            // lo comunica a sua volta al chiamante
            if (c.search_one ()) { *this = c; return true; }
        }
    return false;
    // se giunge a restituire false, vuol dire che a partire
    // dallo stato corrente non è stata trovata alcuna soluzione:
    // nessuna mossa ha portato al successo!
}

```

Qui N denota una costante, caratteristica del puzzle, che è il massimo numero di mosse che possono presentarsi in uno stato del gioco (ad esempio sarà 8 per il giro

del cavallo, che deve tentare appunto otto mosse, al più: quelle lecite cadranno all'interno della scacchiera e su una casa non ancora visitata); per altri giochi, come il Sudoku, converrà invece specializzare l'istruzione **for** ...<sup>25</sup> Notiamo che i metodi `is_legal` e `make_move` sono lasciati “pubblici”, ma la loro applicazione dall'esterno implica la conoscenza della numerazione delle mosse!

Qualche osservazione più tecnica:

a) se e quando un'applicazione del metodo `search_one` ha successo, la soluzione trovata non soltanto è stampata ma è poi propagata a ritroso, e infine rimane memorizzata nell'oggetto al quale abbiamo applicato (all'esterno) il metodo `search_one` (nell'esempio di programma `main` scritto poc'anzi, tale oggetto ha nome `p`): e questo è l'altro effetto collaterale!

Così siamo riusciti a sfruttare (bene) lo *stack* di sistema.

b) In casi semplici, come per il giro del cavallo, può essere più conveniente – per motivi di efficienza – sostituire l'**if** che costituisce il corpo del ciclo **for** con il seguente:

```
if (is_legal(i)) {
    // esegue la mossa numero i su questo stesso oggetto ...
    make_move(i);
    if (search_one()) return true;
    // ... e, se non ha portato al successo, la ritira
    retract_move(i);
}
```

Ovviamente bisogna aggiungere il metodo `void retract_move (int)`. Così facendo, non solo si evita l'uso di un'istanza d'appoggio della classe `puzzle`, ma – almeno per il momento – si possono eliminare dalla definizione della classe stessa i due metodi che rendono *first class* le sue istanze: il costruttore di copia e l'operatore di assegnazione (*overloaded*).

Tuttavia, per certi giochi (come il Sudoku) può rivelarsi assai più difficile definire un metodo che ritiri una mossa (nel Sudoku, ciò vuol dire che annulli la scelta di un numero da scrivere in una casella, operazione che – come sappiamo – avrà implicato la cancellazione di quel numero da tutte le altre caselle della stessa riga, della stessa colonna e dello stesso riquadro, che nello stato precedente lo contenevano...).

---

<sup>25</sup> Rappresentiamo uno schema di Sudoku con una *matrice di insiemi di interi*; ciascun insieme contiene i numeri che possono stare nella corrispondente casella. Se nello schema corrente esiste una casella dove non può stare alcun numero (insieme vuoto), allora tale schema è di fallimento; se invece in ciascuna casella può stare esattamente un numero, allora è di successo.

Negli altri casi possiamo pensare di provare a fissare, uno alla volta, tutti i numeri che possono stare in uno degli insiemi che abbiano almeno due elementi (scelto magari fra i più piccoli); fissare un numero implica toglierlo dagli insiemi relativi alle caselle di stesso riquadro o riga o colonna della casella corrispondente all'insieme considerato e, ricorsivamente, ogni volta che resta un solo numero in un insieme, esso va tolto dagli insiemi relativi alle altre caselle di stesso riquadro o riga o colonna.

c) Se non interessa conservare nell'oggetto lo stato finale di successo (se trovato), si può eliminare l'istruzione `*this = c;` (e la definizione dell'operatore di assegnazione) e dichiarare `const` il metodo `search_one` (poiché non modifica più lo stato dell'oggetto al quale è applicato).

Se viceversa interessa conservarlo, ma non stamparlo immediatamente, basta eliminare l'istruzione `print();` dal corpo del primo `if`. Questa, in effetti, è la scelta più elegante: la decisione circa le successive operazioni da eseguire sull'oggetto in questione è demandata al chiamante esterno (nel nostro esempio la funzione `main`).

## Campi e metodi per il giro del cavallo.

Vediamo quali sono le componenti specifiche per realizzare il puzzle del giro del cavallo nella maniera meno sofisticata, tenendo presenti le osservazioni appena fatte. Innanzi tutto i campi, che costituiscono la struttura di dati “privata”, non direttamente accessibile dall'esterno: si può prevedere una matrice statica `b` (*array* a due indici) di interi (inizialmente tutti a valore zero), formata da `m` righe per `n` colonne (con `m` ed `n` costanti da fissare, uguali se si vuole una scacchiera quadrata); inoltre, sebbene non necessari (perché si possono derivare dalla matrice), saranno assai utili altri tre campi interi: gli indici di riga `r` e di colonna `c`, che individuano l'ultima casa raggiunta dal cavallo, e il numero `k` da scrivere nella casa che sarà raggiunta con la mossa successiva (`k` sarà quindi uguale a `b[r][c]` aumentato di un'unità).

Quando creiamo un'istanza della classe, vogliamo fissare la casa da cui parte il cavallo assegnando il valore 1 al corrispondente elemento della matrice. Possiamo allora fare a meno del costruttore senza parametri, ma dobbiamo prevederne uno con due parametri, e cioè gli indici di riga (tra 0 e `m-1`) e di colonna (tra 0 e `n-1`) della casa di partenza:

```
puzzle :: puzzle (int i1, int j1) {
    // stato iniziale, con 1 in riga i1 e colonna j1, 0 altrove
    // (se i1 o j1 cadono fuori, si parte di default da riga 0
    // e colonna 0)
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            b[i][j] = 0;
    if (i1 < 0 || i1 >= m || j1 < 0 || j1 >= n)
        { r = 0; c = 0; }
    else
        { r = i1; c = j1; }
    b[r][c] = 1; // messo 1 nella casa di partenza,
    k = 2;         // 2 è il prossimo numero da scrivere
}
```

Allo scopo di evitare la ricompilazione quando si vogliono cambiare le dimensioni della scacchiera (il numero delle traverse o il numero delle colonne), dovremmo stabilire con una costante la dimensione massima e lasciare poi all'utente, di volta in

volta, la scelta delle dimensioni reali  $m$  e  $n$ : allora dell'intera matrice statica (quadrata) sarà usato un rettangolo  $m \times n$ ;  $m$  e  $n$  saranno ora due campi, e il costruttore avrà due parametri in più per assegnarne i valori (previo controllo che non superino la dimensione massima). Ancor meglio, prevedendo di non rendere pubblico il software sorgente: non stabiliamo alcuna dimensione massima, e il costruttore alloca *dinamicamente* un array (a un indice) di  $m \times n$  interi in cui mappare esattamente la matrice che rappresenta la scacchiera; in tal caso, costruttore di copia e operatore di assegnazione devono essere definiti in modo *deep* (si vedano le pagine 240-243).

Comunque sia, per il problema in esame, anche le altre operazioni sono facili da definire; basta decidere un ordine di esplorazione (ossia una numerazione) per le mosse: ad esempio potremmo partire da quella più in alto a destra (due righe in meno e una colonna in più) e procedere in senso orario. Una soluzione sarà trovata se e non appena  $k$  supererà il prodotto  $m \times n$ . (Per cercare un giro chiuso sarà sufficiente cambiare la definizione del metodo `success`: come?)

```

bool puzzle :: success () const {
    return k > m*n;
}

bool puzzle :: is_legal (int i) const {
    int i1, j1;
    switch (i) {
        case 1: i1 = r-2; j1 = c+1; break;
        case 2: i1 = r-1; j1 = c+2; break;
        case 3: i1 = r+1; j1 = c+2; break;
        case 4: i1 = r+2; j1 = c+1; break;
        case 5: i1 = r+2; j1 = c-1; break;
        case 6: i1 = r+1; j1 = c-2; break;
        case 7: i1 = r-1; j1 = c-2; break;
        case 8: i1 = r-2; j1 = c-1;
    }
    return i1 >= 0 && i1 < m && j1 >= 0 && j1 < n && b[i1][j1] == 0;
    // il valore di verità restituito è quello della proposizione
    // "con la mossa numero i, il cavallo non esce dalla scacchiera
    // e la casa dove si porta non è stata ancora visitata"
}

void puzzle :: make_move (int i) { // Precondizione: is_legal(i)
    switch (i) {
        case 1: r -= 2; c += 1; break;
        case 2: r -= 1; c += 2; break;
        case 3: r += 1; c += 2; break;
        case 4: r += 2; c += 1; break;
        case 5: r += 2; c -= 1; break;
        case 6: r += 1; c -= 2; break;
        case 7: r -= 1; c -= 2; break;
        case 8: r -= 2; c -= 1;
    }
    b[r][c] = k; k++;
}

```

```

void puzzle :: retract_move (int i) {
    // ripristina lo stato precedente l'ultima mossa tentata,
    // con numero d'ordine i
    k--; b[r][c] = 0;
    switch (i) {
        case 1: r += 2; c -= 1; break;
        case 2: r += 1; c -= 2; break;
        case 3: r -= 1; c -= 2; break;
        case 4: r -= 2; c -= 1; break;
        case 5: r -= 2; c += 1; break;
        case 6: r -= 1; c += 2; break;
        case 7: r += 1; c += 2; break;
        case 8: r += 2; c += 1;
    }
}

```

Lasciamo come compito la definizione di un efficace metodo `print`, che magari disegni anche il tracciato, ricordando che nel programma di prova visto al paragrafo precedente basterà scrivere, ad esempio, `puzzle p(2, 0)`; per creare un'istanza (statica) `p` in cui il cavallo parta dalla casa in terza riga, prima colonna.

## Ricerca esauriente di tutte le soluzioni.

Per trovare *tutte* le soluzioni, si deve forzare il *backtracking* anche quando si trova uno stato di successo – attenzione alla complessità, che di solito è esponenziale!

Qui è necessario stampare a una a una le soluzioni, man mano che vengono trovate; e se non ne esistono, non è stampato nulla. In alternativa, si dovrà prevedere la loro memorizzazione in una struttura di dati adeguata: svilupperemo questa idea, migliore, nel prossimo capitolo. Un brevissimo (da scrivere!) programma di prova è:

```

int main () {
    puzzle p; // oppure p(...); come sopra
    p.search_all();
    return 0;
}

```

Questa applicazione del metodo `search_all` avrà come effetto la stampa di tutti gli stati finali di successo ottenibili a partire da `p`; qui comunque, alla fine, `p` rimane esattamente com'era all'inizio.

Il metodo `search_all` si scrive ancor più semplicemente di `search_one`:

```

void puzzle :: search_all () const {
    if (success()) print(); // trovata una soluzione!
    else
        for (int i = 1; i <= N; i++)
            if (is_legal(i)) {
                puzzle c = *this;
                c.make_move(i);
                c.search_all();
            }
}

```

Analogamente a quanto già osservato, in certi casi – come per il giro del cavallo – può essere più conveniente sostituire il corpo dell'**if** all'interno del ciclo **for** con:

```
make_move(i);  
search_all();  
retract_move(i);
```

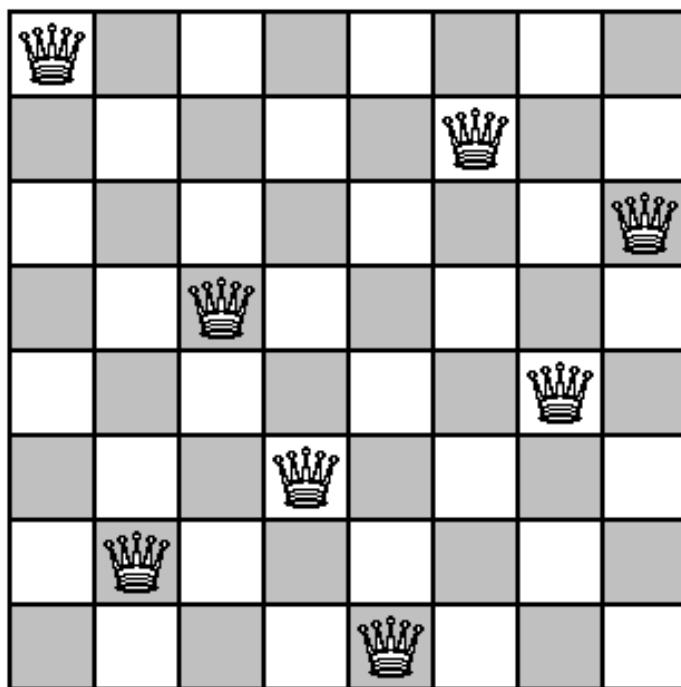
Allora, va da sé, bisogna togliere **const** nella definizione del metodo `search_all`, poiché ora l'oggetto viene – seppur temporaneamente – modificato.

Lasciamo ai lettori il compito di apportare al codice gli appropriati cambiamenti allo scopo di contare le soluzioni trovate.

## Il problema delle $n$ regine.

Un altro famoso rompicapo è la ricerca di soluzioni per il problema delle  $n$  regine: si devono disporre  $n$  regine su una scacchiera  $n \times n$  facendo sì che nessuna di esse si trovi sulla linea d'azione di un'altra.

Fu proposto dallo scacchista tedesco Max Bezzel nel settembre del 1848, su *Schachzeitung*, per il caso classico  $n = 8$ . Una possibile disposizione è riportata in figura.



Ricordiamo che una regina domina in orizzontale, in verticale e lungo le diagonali alle quali appartiene la casa in cui si trova: pertanto due regine non potranno stare su una stessa traversa, né su una stessa colonna, né su una stessa diagonale. È immediato constatare che, su una scacchiera  $n \times n$ , più di  $n$  regine non possono trovar posto, e che non vi sono soluzioni per  $n$  uguale a 2 o 3.

Si tratta di un caso particolare del problema dell'*insieme indipendente massimo* (in un grafo non orientato, trovare un sottoinsieme di nodi a due a due non adiacenti, che sia il più grande possibile: si confronti col problema 3 a pagina 103), che è NP-equivalente; e qui aggiungiamo soltanto che è stato ideato un algoritmo *ad hoc* che trova una soluzione al problema delle  $n$  regine addirittura in tempo pseudo-lineare.

Come può essere rappresentata una soluzione? Ad esempio, poiché le regine devono occupare colonne diverse, si può caratterizzare una soluzione con una sequenza di  $n$  interi  $[r_1, r_2, \dots, r_n]$ , ove ciascun  $r_j$  indica l'indice di riga in cui è collocata la regina della colonna di indice  $j$ ; così, la soluzione raffigurata alla pagina precedente è espressa dalla sequenza  $[0, 6, 3, 5, 7, 1, 4, 2]$ .

Il procedimento di ricerca “a tappeto” inizierà piazzando una regina nella prima colonna, e si baserà sul calcolo delle case della prossima colonna libera minacciate dalle regine già sistematiche nelle colonne precedenti. Si potrà scoprire così che, nel caso  $n = 8$ , delle  $8! = 40320$  permutazioni degli indici di riga da 0 a 7, soltanto 92 descrivono posizioni che soddisfano il requisito: questo risultato fu ottenuto dal matematico Franz Nauck nel 1850.

Passando dalla tradizionale scacchiera  $8 \times 8$  a una scacchiera ridotta  $6 \times 6$ , si scende da 92 soluzioni a 4 soltanto. Queste si riducono ancora a 12 e 1, rispettivamente, se si considerano le simmetrie (rotazioni e/o ribaltamenti). (Perché allora, non considerando le simmetrie, sono 92, che *non* è il prodotto di 12 per 8?)

Nel caso  $n = 8$ , quale casa della scacchiera (modulo simmetrie) sarà certamente occupata da una regina? Fu il prolifico creatore di giochi Sam Loyd – uno dei grandi nomi nella storia dei puzzle: parleremo di lui nel prossimo capitolo – ad osservare che vi è una sola casa occupata da una regina che sia comune a tutte le possibili soluzioni: è la casa d1, cioè quella occupata dalla regina bianca all'inizio di una partita a scacchi, o equivalentemente una delle sette ad essa simmetriche (nell'esempio dato in figura è occupata la casa e1, speculare a d1 rispetto alla mediana verticale).

Un altro problema con le regine consiste nel disporne cinque sulla scacchiera  $8 \times 8$  in modo tale che tutte le 64 case siano controllate o occupate: cinque regine sono necessarie e sufficienti a tale scopo. Lasciamo scoprire al lettore quante sono le soluzioni che soddisfano anche il precedente requisito: le cinque regine non devono attaccarsi a vicenda.

*Idem* con i cavalli: qual è il numero massimo di cavalli che si riescono a disporre sulla scacchiera evitando che si minaccino l'un l'altro? E qual è invece il numero minimo di cavalli che permettono la copertura di tutte le case?

## Il problema dei sei cavalli.

Più complesso del problemino di al-Adli, si ritrova in varie ambientazioni, più o meno moderne: anche sulla scacchiera  $4 \times 3$  rappresentata nella figura alla pagina successiva, i tre cavalli bianchi devono scambiarsi il posto con i tre neri, e mai deve accadere che un cavallo di un colore ne minacci uno dell'altro colore.

Quanti salti dovranno compiere?



[*Suggerimento:* analogamente al gioco dei quattro cavalli, non importa quale sia il colore che dà inizio alla danza, che qui sarà perfettamente speculare rispetto alla mediana orizzontale! Due cavalli – uno bianco e uno nero, posti su uno stesso lato – faranno 3 salti ciascuno, gli altri quattro cavalli ne faranno 4 ciascuno; in totale, quindi, sono richiesti 22 salti.]

### I problemi del labirinto.

La seguente figura illustra un esempio.

*	*	*	*	*	*	*	*	*	*	*
*		P	□	□	*		*		*	*
*	*	■	*	□	*		*		*	*
*	■	■	*	□	□	□	*		*	*
*	■	*	*	*	*	□	□	□	*	*
■	■	*			*	*	*	□	*	*
*	*	*		*	*			□	*	
*			Q		*			□	□	
*	*	*	*	*	*	*	*	*	*	*

Un labirinto può essere rappresentato da una matrice di caratteri, con opportune convenzioni: nel caso di figura, un asterisco indica una casella “siepe” (non percorribile) mentre uno spazio bianco indica una casella “vuota” (percorribile).

Dato un labirinto e dati gli indici di riga e colonna della casella vuota di partenza, si deve decidere se esiste un cammino che conduca da tale casella a una casella “di uscita”, cioè una casella vuota posta su uno dei lati del labirinto.

Il cammino deve essere costituito esclusivamente da caselle vuote, l'una adiacente *ortogonalmente* alla successiva; non è consentito procedere in diagonale.

In caso affermativo, si dovrà esibire (in forma grafica) un cammino (non ridondante, cioè che non attraversi più volte una stessa casella e, possibilmente, che non possa essere abbreviato) che giunga fino a una casella di uscita.

In figura sono riportati due diversi cammini (non ridondanti) che permettono di uscire dal labirinto a partire dalla casella P; partendo invece dalla casella Q non si può uscire.

Un secondo problema richiede come ulteriori dati gli indici di riga e colonna di un'altra casella, e consiste nel trovare, se c'è, un cammino che unisca le due caselle indicate. Nell'esempio, tra P e Q non esiste alcun cammino.

Si può procedere come per il giro del cavallo. In ogni casella (vuota) all'interno del labirinto sono al massimo quattro i passi da tentare; onde evitare di ritornare sui propri passi cadendo in circoli viziosi, bisognerà ricordare le caselle già percorse, marcandole con un carattere apposito, e assegnarvi nuovamente lo spazio bianco, qualora si dovesse tornare indietro...

Come promesso, concludiamo il capitolo con la risposta al quesito (lasciato in fondo alla pagina 202) sui cicli del cavallo, simmetrici non soltanto rispetto al centro ma anche ai due assi orizzontale e verticale. Quello che avevamo presentato... *non* è un giro chiuso, ma uno *pseudo-ciclo*: il diagramma in realtà mostrava *quattro sotto-cicli* intrecciati, ma distinti, ciascuno dei quali tocca 16 case! Sulla scacchiera 8×8, cicli con quel tipo di simmetria non ce ne sono: in effetti, l'unica simmetria possibile è quella rotazionale di ordine 2 (D. E. Knuth, 1996).

**Parole chiave:** cammini e cicli *hamiltoniani*, algoritmo di Warnsdorf (euristico) e suoi miglioramenti, quadrati semimagici, algoritmo di esaurimento con *backtracking* (per la ricerca in profondità di una o di tutte le soluzioni di un *puzzle*), definizione di una classe in C++ con *campi* e *metodi*.

## 8. La soluzione sta nella traccia!

Passiamo ora a considerare quei puzzle per i quali è necessario stampare, o conservare, la sequenza completa di stati intermedi, o di mosse in qualche modo rappresentate, che ha portato alla prima soluzione trovata (se c'è) oppure a ciascuna soluzione possibile o – spesso più realisticamente – a ciascuna soluzione che soddisfi certi requisiti (ad esempio, la minimalità del numero di mosse). In molti giochi, infatti, è già conosciuto (del tutto o quasi) lo stato finale da raggiungere, e il problema consiste proprio nel determinare una sequenza di mosse risolutiva.

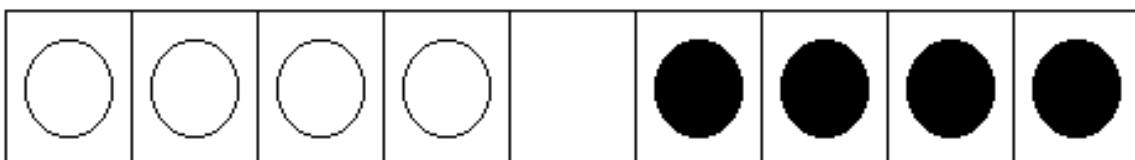
In verità, un esempio era già celato tra gli ultimi quesiti proposti nel capitolo precedente, quello dei sei cavalli. Il lettore più attento avrà subito notato che, in casi come quello, quando si dovrà progettare un programma al computer occorrerà prestare attenzione a che la sua esecuzione eviti il pericolo di ritornare in uno stato già visitato durante la ricerca di una soluzione, ossia che non corra il rischio di cadere in *cicli* (senza terminare). Di solito questo pericolo sussiste, ma ci sono dei puzzle che fanno eccezione, poiché, ad esempio, le pedine non possono retrocedere o ad ogni mossa ne è mangiata una.

Inizieremo da quest'ultimo caso, più semplice, per poi vedere in generale come trattare la possibilità di cicli e trovare le soluzioni più brevi (che richiedono cioè il minor numero di mosse) grazie a un algoritmo di esaurimento che fa uso di una struttura di dati “astratta” – con le sue operazioni peculiari – realizzata mediante una classe *parametrica* (o *generica*): la *pila* di un qualche tipo di oggetti.

### Rane e rospi.

Nella gara *Kangourou dell'Informatica* di marzo 2009 fu proposto un quesito su un gioco che risale almeno agli anni '70 dell'Ottocento; era sì presentato in versione ridotta, ma poi nel libretto delle soluzioni fu spiegato come affrontarlo nel caso generale. Diversi sono i nomi con i quali è noto: là fu chiamato *Hop!* ma in origine era detto *Frogs and Toads* (rane e rospi).

Consiste nel trovare una sequenza di mosse per portare le pedine nere (i rospi) al posto di quelle bianche (le rane) e viceversa. Il posto centrale è inizialmente libero, e tale deve ovviamente rimanere alla fine del gioco; le pedine bianche si spostano soltanto verso destra, quelle nere soltanto verso sinistra; una pedina può spostarsi di un posto in avanti, se il posto davanti è libero, oppure può scavalcare una pedina dell'altro colore, se il posto successivo a questa è libero.



La domanda era: in quante mosse può essere risolto il gioco?

Se  $n$  è il numero di pedine di ciascun colore, e quindi il tavoliere è costituito da una fila di  $2n + 1$  caselle, e si stabilisce anche il colore che inizia a muovere, allora la soluzione è *unica* e richiede esattamente  $n \cdot (n + 2)$  mosse. Naturalmente, se inizia l'altro colore, la soluzione è simmetrica.

Ad esempio, se le pedine sono quattro per ciascun colore, come in figura, la soluzione comporta 24 mosse.

È interessante notare che il numero di stati di cui è composta la sequenza risolutiva (compresi l'iniziale e il finale), che è chiaramente uno in più del numero di mosse, si può esprimere come il quadrato di  $(n + 1)$ .

Se numeriamo le caselle da sinistra a destra con  $-n, -(n - 1), \dots, -2, -1, 0, 1, 2, \dots, (n - 1), n$ , e senza ambiguità indichiamo ogni volta il numero della casella in cui si trova la pedina che si muove, allora, nel caso in cui inizi a muovere il bianco, le sequenze di mosse risolutive sono, al variare di  $n$ , le seguenti:

- per  $n = 1$ :  $-1, 1, 0$
- per  $n = 2$ :  $-1, 1, 2, 0, -2, -1, 1, 0$
- per  $n = 3$ :  $-1, 1, 2, 0, -2, -3, -1, 1, 3, 2, 0, -2, -1, 1, 0$
- per  $n = 4$ :  $-1, 1, 2, 0, -2, -3, -1, 1, 3, 4, 2, 0,$   
 $-2, -4, -3, -1, 1, 3, 2, 0, -2, -1, 1, 0$
- ...

Che cosa si nota in queste sequenze? Se escludiamo l'ultima mossa (l'ultimo 0), ciascuna di esse è *speculare* rispetto al centro, a condizione che siano cambiati i segni: ciò rispecchia il fatto che la configurazione iniziale del tavoliere di gioco è speculare rispetto a quella finale, la configurazione *dopo* la prima mossa è speculare rispetto a quella che si ha *prima* di fare l'ultima mossa, la configurazione *dopo* la seconda mossa è speculare rispetto a quella che si ha *prima* di fare la penultima mossa, e così di seguito.

Inoltre, come si può osservare, ciascuna sequenza risolutiva è ottenuta dalla precedente (quella corrispondente a  $n$  diminuito di un'unità) aggiungendovi opportune mosse “al centro” (in numero dispari, via via crescente: dapprima 5, poi 7, poi 9, poi 11, ...): come si determinano?

In una configurazione *legalmente raggiungibile* di questo gioco, quante sono, al più, le mosse possibili?

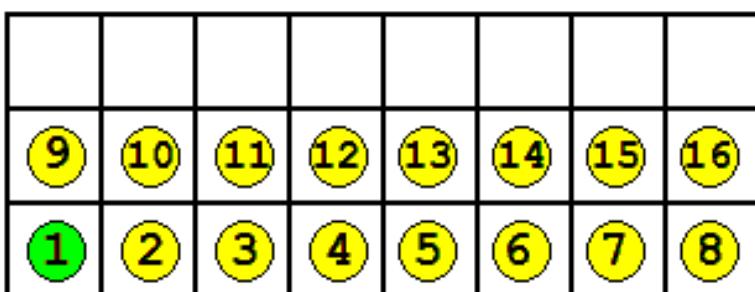
Si noti infine che le singole mosse *non* si alternano fra i due colori: e nulla, infatti, era stato detto al proposito!

### **Un solitario con le pedine proposto da Dudeney.**

Come si può facilmente immaginare, *Frogs and Toads* si è prestato a moltissime variazioni sul tema. Ne propose qualcuna anche uno dei maggiori inventori di enigmi e giochi matematici di tutti i tempi, l'inglese Henry Ernest Dudeney (1857-

1930), del quale però – per il momento – ci interessa un altro solitario con le pedine, che si trova nei suoi *Amusements in Mathematics*, pubblicati a Londra, nel 1917, presso la casa editrice Thomas Nelson and Sons (si possono reperire alla pagina web <http://www.gutenberg.org/ebooks/16713>).

L'autore lo ritiene piuttosto semplice, ma non tanto da essere privo d'interesse; ne dà una soluzione, ma non dice nulla su quante ve ne siano.



Ad ogni mossa, una pedina ne salta un'altra che viene mangiata, in verticale o in orizzontale, ma non in diagonale come invece accade nella dama. Dopo quindici prese, deve rimanere sul tavoliere soltanto la pedina numero 1, cioè quella che inizialmente si trova nell'angolo in basso a sinistra.

### Che cosa cambia nell'algoritmo di esaurimento?

I due puzzle visti nei due paragrafi precedenti hanno in comune una caratteristica: fatta una mossa, non è possibile trovarsi in uno stato già visitato. Nel caso di rane e rospi è stata formulata una regola per ricavare l'unica sequenza di mosse risolutiva (a meno della simmetria); tuttavia, nei casi in cui ciò non sia possibile o non si sappia se e quante soluzioni ammetta il problema, si può sempre procedere in modo analogo a quanto illustrato nel precedente capitolo.

Vediamo dunque come si fa, in generale, a realizzare un programma al computer che proceda per tentativi, utile quando non si conosca o non si abbia voglia di pensare a una strategia che raggiunga l'obiettivo senza mai commettere errori, cioè – come si dice – “in modo deterministico”, strategia che potrebbe non esistere nemmeno.

Abbiamo già analizzato nel capitolo precedente un algoritmo che procede per tentativi. Innanzi tutto, opera sul tavoliere di gioco nella sua configurazione attuale – all'inizio, ovviamente, sarà quella iniziale! – e la prima azione che compie è controllare se questa corrisponde alla configurazione finale: se sì, allora ha trovato una soluzione; altrimenti, guarda quali sono le mosse possibili a partire da questa configurazione. Se ve ne sono, prova a fare la prima, arriva in una nuova configurazione, e da lì il procedimento si ripete, *ricorre*; se non giunge alla soluzione, prova a fare la seconda mossa eccetera. Quando non sia più possibile tentare una nuova mossa, perché non ce ne sono o perché si sono provate tutte senza arrivare a una soluzione, torna indietro di un passo (operazione che, tecnicamente, si chiama *backtracking*, come già sappiamo) rimuovendo l'ultima mossa fatta, e così via.

In questo modo, l'algoritmo esplora tutto l'albero di gioco, fino a trovare una soluzione, se esiste, o a provare “per esaurimento” che non ve ne sono.

Se vogliamo fermarci all'eventuale prima soluzione trovata, stampando in uscita la sequenza di stati o di mosse per giungervi, è sufficiente modificare un poco il metodo `search_one`, togliendo un'assegnazione e spostando l'istruzione di stampa:

```
bool puzzle :: search_one () const {
    if (success()) return true; // trovata una soluzione!
    for (int i = 1; i <= N; i++)
        if (is_legal(i)) {
            puzzle c = *this;
            // è intervenuto il costruttore di copia di puzzle
            c.make_move(i);
            if (c.search_one()) {
                c.print(); // oppure: std::cout << i << " ";
                // se basta il numero della mossa
                return true;
            }
        }
    return false;
}
```

Ricordiamoci allora di definire in maniera opportuna, se necessario, il costruttore di copia per la classe `puzzle` (altrimenti sarà invocato quello di *default*, che copierà i valori campo per campo).

Adesso l'unico effetto collaterale del metodo `search_one` (che non modifica lo stato dell'oggetto a cui è applicato) consiste nelle stampe prodotte, le quali vanno lette in ordine inverso: le eventuali informazioni sulla sequenza di mosse effettuate devono essere lette “alla rovescia” in quanto sono stampate a partire dalla configurazione finale di successo trovata, procedendo a ritroso fino allo stato raggiunto con la *prima* mossa!

Infatti, se un'esecuzione di `search_one` arriva a uno stato finale di successo restituisce `true`; l'esecuzione ricorsiva chiamante stampa lo stato finale stesso (oppure il numero d'ordine della mossa che ha portato a quello stato finale), e a sua volta restituisce `true` a chi l'aveva attivata...

Le cose si complicano un po' quando vogliamo trovare *tutte* le soluzioni.

In effetti, il metodo `search_all` dovrà invocare un metodo ausiliario (privato) che abbia un parametro esplicito di tipo “pila” (passato per riferimento a fini di efficienza) in cui memorizzare la sequenza di stati intermedi o di mosse che hanno portato allo stato corrente, in modo tale che – ogniqualvolta lo stato corrente si riveli di successo – l'*intera* sequenza che determina la soluzione trovata sia presente nella “pila” e possa essere stampata!

La pila (in inglese *stack*) è un tipo di dato (astratto) che prevede l'inserimento (*push*) e il successivo prelevamento (*pop*) di oggetti (di uno stesso tipo) sempre dalla stessa

parte (la cima della pila): così, quando si preleva un oggetto da una pila non vuota, si tratta sempre dell'*ultimo* oggetto che era stato inserito.

La definizione di una classe “pila” *parametrica*, chiamata `STACK<ITEM>` e corredata di tutte le operazioni a noi utili, è data alla fine del capitolo. Qui, per ora, è usata (privatamente) dalla classe `puzzle` e serve per tener traccia del percorso, lungo un ramo dell’albero di gioco, dallo stato iniziale a quello corrente.

Se è sufficiente tenere memoria dei numeri d’ordine delle mosse fatte, allora servirà una pila di interi, altrimenti – quando si devono ricordare le configurazioni intermedie – si userà una pila di istanze della classe `puzzle`.

Cominciamo dalla prima ipotesi:

```
void puzzle :: search_all () const {
    STACK<int> L;
    // il costruttore inizializza L come "pila vuota" ...
    aux_search_all(L);
    // ... e alla fine della ricerca resterà vuota!
}
```

Il metodo `aux_search_all` (per chiarezza abbiamo cambiato nome, sebbene `aux_` non sia necessario data la presenza del parametro) sarà dichiarato nella parte *privata* della classe `puzzle`. Anche a motivo di efficienza, abbiamo preferito una procedura, con un parametro per riferimento, a una funzione che restituisca esplicitamente una pila. Tale procedura può essere così definita:

```
void puzzle :: aux_search_all (STACK<int> & L) const {
    // L = elenco delle mosse che portano dallo stato iniziale a
    //      quello corrente (l'ultima inserita è l'ultima fatta)
    if (success()) L.print_cout();
    // è stampato l'elenco delle mosse fatte
    else
        for (int i = 1; i <= N; i++)
            if (is_legal(i)) {
                puzzle c = *this;
                c.make_move(i);
                L.push(i); // inserisce i in cima a L
                c.aux_search_all(L);
                L.pop(); // backtracking: preleva i (qui perduto)
                // dalla cima di L, e quindi L ritorna come
                // era prima dell'applicazione ricorsiva
            }
}
```

Per tener conto degli stati intermedi, bisogna usare invece uno `STACK<puzzle>` `L`: allora l’applicazione del metodo `push` alla pila `L` riceverà come argomento `c` (anziché `i`) e avrà bisogno del costruttore *senza parametri* della classe `puzzle`, che dunque dovrà essere necessariamente definito (anche nel caso in cui non compia alcuna azione, ovvero abbia il corpo vuoto).

L'unico altro punto da cambiare, e al quale occorre prestare una certa attenzione, è la stampa della pila in caso di successo: in generale, si dovrà prevedere l'applicazione alla pila `L` di un metodo `print` che a sua volta applichi il metodo `print` definito per la classe `puzzle`.

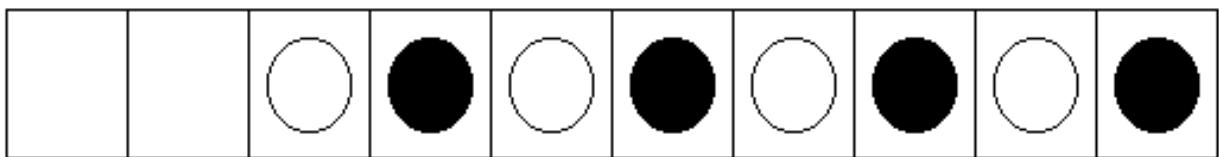
La realizzazione più elegante, comunque, non dovrebbe stampare alcuna soluzione, bensì raccoglierle tutte in un'adeguata struttura di dati, via via che sono trovate. Giacché disponiamo della pila parametrica, la scelta più conveniente cade su una “pila di soluzioni”, ossia uno `STACK<STACK<int>>` oppure uno `STACK<STACK<puzzle>>` (ricordarsi di lasciare uno spazio tra i due caratteri `>`, perché `>>` è simbolo di altre operazioni in C++). Ciò è subordinato al fatto che le pile possano essere trattate come *first-class objects*, senza che sorgano problemi di *aliasing*.

Se qualcuno dei lettori non trovasse del tutto chiari gli ultimi punti del discorso testé fatto, non si preoccupi: sarà sufficiente dare una scorsa alla definizione della classe parametrica `STACK<ITEM>`, riportata alla fine del capitolo, ma soprattutto prestare attenzione all'uso che ne faremo nei prossimi due paragrafi.

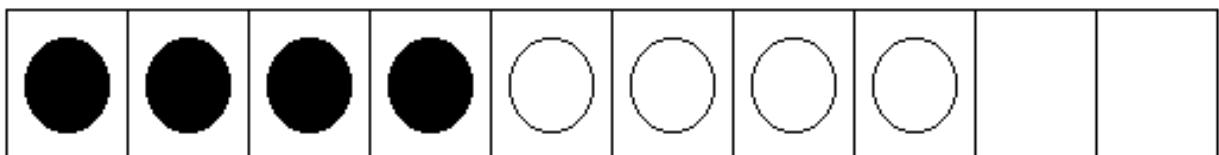
### **Un solitario con le pedine proposto da Tait.**

Illustriamo ora un puzzle che, forse già conosciuto in Giappone da oltre un secolo, incuriosì Peter Guthrie Tait della Edinburgh Mathematical Society, il quale – nel gennaio del 1884 – raccontò di averlo visto proporre su un treno, poche settimane prima, con delle monete di due diversi tipi.

Fu poi studiato e generalizzato da Henri Auguste Delannoy della Société Mathématique de France, come ricorda Édouard Lucas in una delle sue *Récréations mathématiques* raccolte in volume nel novembre del 1892, un anno dopo la sua morte. Lucas ne aveva già parlato in un paio di articoli sui giochi con le pedine, pubblicati sul settimanale *La Nature* nel giugno del 1887.



Ad ogni mossa si devono spostare nelle due caselle libere due pedine qualsiasi, purché contigue, lasciandole nello stesso ordine. Qual è la sequenza di mosse più breve che permette di arrivare alla configurazione di seguito riportata?



Qui si corre il pericolo di ritornare in uno stato già visitato (lungo il corrente ramo dell’albero): banalmente, fatta una mossa, esiste subito la possibilità di tornare nello stesso stato con la mossa successiva! In termini tecnici: il “grafo del gioco” è *ciclico*, e quindi il suo *unfolding* (cioè lo sviluppo come albero che abbia in radice lo stato iniziale) è un albero che presenta rami di lunghezza infinita. In questo caso, è dunque più problematico persino trovare una qualunque soluzione, se c’è.

Per evitare di cadere in *loop*, il provvedimento più immediato e semplice da adottare è porre un limite a priori (per mezzo di una costante `max_moves`) al numero di mosse da fare, ossia alla profondità massima da raggiungere nell’esplorazione di tale albero. L’ovvio aspetto negativo consiste nell’eventualità di fissare un limite troppo basso, insufficiente a trovare una (più lunga) sequenza di mosse risolutiva, fattore che si aggiunge all’aggravio di tempo impiegato per esplorare inutili percorsi ciclici. Pensare di aumentare progressivamente la profondità massima da raggiungere, oltre all’inefficienza, presenta un serio inconveniente: provate a immaginare che cosa accade quando manchi la garanzia dell’esistenza di una soluzione! (Se davvero non ci fossero soluzioni, semplicemente non si arriverebbe mai a sapere che non ci sono soluzioni; se però gli stati del gioco sono in numero finito, si può assumere questo numero, meno uno, come `max_moves`: comunque proibitivo!)

In alternativa, si può cambiare strategia, visitando l’albero *per livelli* (ossia in ampiezza, *breadth first*) a partire dalla radice. L’onere aggiuntivo comprende l’uso di una *coda* di stati in cui immettere gli stati del livello successivo, via via che vengono generati a partire dallo stato corrente... (A differenza della pila, la coda prevede l’inserimento da una parte e il prelevamento dall’altra: così, quando si preleva un oggetto da una coda non vuota, si tratta sempre di quello da più tempo inserito.) È pur vero che, se c’è una soluzione, questo metodo la trova, nel senso che arriva prima o poi a visitare uno stato finale; ma, ammesso di trovare uno stato finale, come si fa a ricordare le mosse fatte per giungervi? La questione si può risolvere mantenendo l’intero albero esplorato, ma ciò di solito richiede una quantità di memoria esorbitante... E poi, se non si ha la certezza dell’esistenza di una soluzione, si ricade nel problema di terminare correttamente la ricerca.

Mantenere in un esplicito parametro la sequenza di stati del gioco sul ramo corrente è dunque praticamente indispensabile.

Per semplicità, procediamo con la classica ricerca *depth first* e memorizziamo gli stati visitati in uno `STACK<puzzle> L`, a patto che siano permesse due operazioni:

- il confronto fra due stati, per decidere se sono uguali oppure no;
- la ricerca di uno stato nella pila `L`, per stabilire se vi sia già oppure no.

Abbiamo quindi arricchito la classe `puzzle` con l’*overloading* dell’operatore `==` il cui prototipo è:

```
bool operator == (const puzzle &) const;
```

e che dovrà essere definito in modo appropriato a seconda del gioco; ad esempio, per quello in esame, basterà controllare se le due file (*array*) di caselle, in cui ciascuna

casella ha un valore *ternario*, sono identiche.

Inoltre, per la classe parametrica STACK<ITEM> abbiamo previsto un metodo a valori booleani *is\_in* che, dato un ITEM, controlli se è uguale a qualcuno di quelli “impilati”.

A questo punto siamo pronti per riscrivere – un’ultima volta! – il metodo search\_one, affinché possa trovare una soluzione non soltanto al solitario proposto da Tait, ma anche agli altri in cui sono possibili cicli, ferma restando l’ipotesi (implicitamente assunta) che gli stati raggiungibili durante il gioco siano in numero finito:

```
bool puzzle :: search_one (STACK<puzzle> & L) const {
    if (success()) return true;
    for (int i = 1; i <= N; i++)
        if (is_legal(i)) {
            puzzle c = *this;
            c.make_move(i);
            if (!L.is_in(c)) {
                // lo stato raggiunto dal corrente con la mossa numero i
                // non è stato ancora visitato (lungo il ramo corrente)
                L.push(c); // mette una copia di c sulla pila L
                if (c.search_one(L)) return true;
                L.pop(); // l'istanza di puzzle restituita non serve più
            }
        }
    return false;
}
```

In ogni momento, la pila L contiene la sequenza di stati sul ramo correntemente esplorato, da quello iniziale (che sta in fondo alla pila) a quello corrente (che sta in cima alla pila).

Ogniqualvolta sia raggiunto uno stato già visitato lungo il ramo corrente (dunque uno stato che già occorre nella pila L) è inibito l’avanzamento, che porterebbe l’algoritmo a cadere in *loop*.

Un programma di prova può essere il seguente:

```
int main () {
    puzzle p;
    // oppure p(...); p rappresenta lo stato iniziale del gioco
    STACK<puzzle> L; // la "pila di stati" L è creata vuota
    L.push(p); // lo stato iniziale è copiato su L
    if (p.search_one(L))
        // una soluzione è stata trovata e mantenuta in L
        // (passato per riferimento)
        L.print();
    else
        std::cout << "Non esiste soluzione." << std::endl;
        // ... e in L è rimasto soltanto lo stato iniziale p
    return 0;
}
```

L'applicazione di `print` a `L` dovrà stampare – e stamperà! – tutta la sequenza di stati risolutiva, che porta da quello iniziale, che sta in fondo alla pila `L`, a quello corrente, che è finale e sta in cima alla pila `L`.

Un'osservazione importante, che chiarisce un punto esposto nel paragrafo precedente: la classe parametrica `STACK<ITEM>` dovrà avere *due* metodi per la stampa del contenuto della pila: uno, chiamato `print_cout`, potrà essere applicato quando il tipo degli oggetti impilati permetta l'uso dell'operatore `<<` (impiegato dalla libreria `iostream` per l'*output*), come nel caso degli interi; l'altro, chiamato `print`, potrà essere applicato quando la classe degli oggetti impilati abbia a sua volta un metodo `print`, come accade nel caso della classe `puzzle`.

Naturalmente, in generale, un'applicazione di `search_one` potrebbe rivelarsi oltremodo costosa: nei casi peggiori, infatti, la lunghezza dei rami può addirittura uguagliare il numero degli stati del gioco...

## Troviamo le soluzioni più brevi!

Il solitario proposto da Tait può essere generalizzato in modo ovvio al caso di  $n$  coppie di pedine, una bianca e una nera, disposte in una fila di  $2 \cdot (n + 1)$  caselle, a partire dalla terza; le prime due a sinistra sono lasciate vuote. Lo stato finale presenta prima tutte le pedine nere, poi le bianche e infine le due caselle libere.

Quando si affrontano simili problemi, non ha molto senso pensare di “trovare tutte le soluzioni”, bensì è ragionevole porsi l'obiettivo di “trovare la soluzione o le soluzioni che richiedano il minimo numero di mosse”.

Tuttavia, a priori, si potrebbe non sapere se il puzzle avrà soluzioni e, anche in caso affermativo, quante mosse saranno necessarie per risolverlo: si pensi, ad esempio, a un gioco come quelli del quindici o dell'otto, di cui parleremo più avanti. Allora, sempre per semplicità, possiamo attenerci alle indicazioni appresso descritte.

Innanzi tutto, arricchiamo la classe `STACK<ITEM>` con un metodo a valori interi `height` che calcola e restituisce il numero di `ITEM` attualmente impilati. Quindi creiamo una “pila di soluzioni” (cioè un'istanza di `STACK<STACK<puzzle>>`) `S`, inizialmente vuota.

Se e quando troviamo una soluzione, ne mettiamo una copia sulla pila `S` (con l'operazione `push`) e ricordiamo il numero di mosse di cui è costituita (calcolato con l'operazione `height`), che al momento è il minimo richiesto.

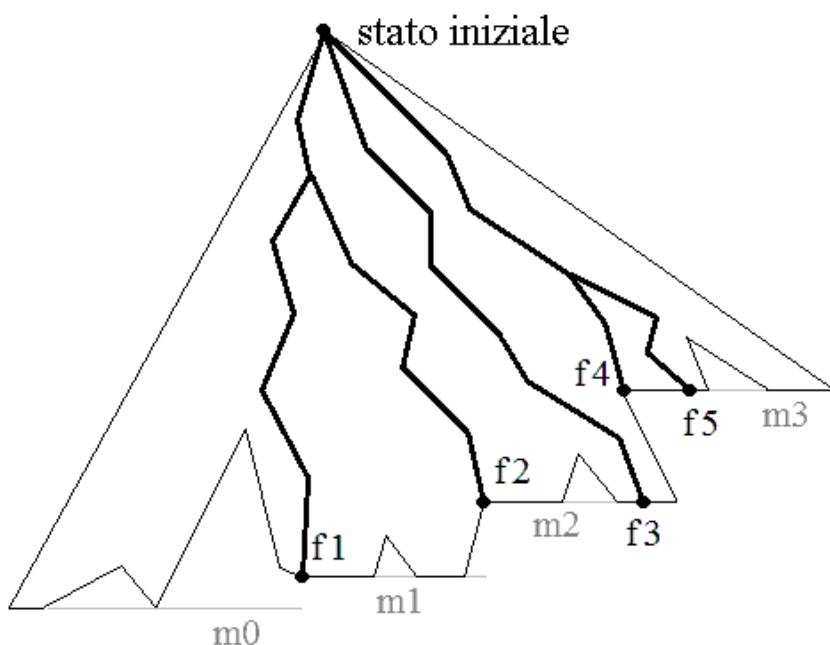
In seguito, ogni volta che troveremo una nuova soluzione con un numero *minore o uguale* di mosse rispetto al minimo corrente, la conserveremo: nel caso “minore”, svuoteremo la pila `S` (contenente soluzioni più lunghe) e aggiorneremo il minimo numero di mosse; in ogni caso, aggiungeremo alla pila `S` una copia di questa nuova soluzione.

Alla fine in `S` ritroveremo *tutte* le soluzioni col minimo numero di mosse.

Come si può procedere all'atto pratico? Si dichiara un *campo di classe* `min_moves` (che è allocato soltanto una volta) e lo si inizializza con un valore intero,

sufficientemente grande, mediante un *metodo di classe* `set_min_moves` (un metodo di classe opera esclusivamente su campi di classe), ponendo così un limite a priori al numero di mosse (anche per efficienza, si dovrebbe cercare una ragionevole maggiorazione della lunghezza di una soluzione).

```
class puzzle {
    private:
        static int min_moves;
        // un campo di classe: serve per ricordare il numero
        // di mosse della più breve soluzione finora trovata
        ...
    public:
        static void set_min_moves (int mm) {
            // metodo di classe: assegna il (nuovo) valore mm
            // al campo di classe min_moves
            min_moves = mm;
        }
        ...
};
```



La figura illustra un esempio. Il campo `min_moves` è posto inizialmente a valore  $m_0$ ; quando è trovato lo stato finale  $f_1$ , è aggiornato con  $m_1$ : la corrispondente soluzione è memorizzata nella pila e da quel momento l'albero sarà potato alla profondità  $m_1$ . Quando poi è trovato lo stato finale  $f_2$  (e dunque una soluzione più breve, con  $m_2$  mosse), `min_moves` è aggiornato con  $m_2$  (e la soluzione che aveva portato a  $f_1$  può essere dimenticata); quando è trovato lo stato finale  $f_3$  (e cioè una soluzione con lo stesso numero di mosse  $m_2$ ), `min_moves` rimane invariato (ma è aggiunta sulla pila la nuova soluzione individuata). Alla fine saranno conservate nella pila soltanto le due soluzioni che hanno portato agli stati finali  $f_4$  e  $f_5$ , in  $m_3$  ( $< m_2$ ) mosse.

Un programma di prova potrà essere scritto, ad esempio, così:

```
int puzzle :: min_moves;
// purtroppo, in C++, ciascun campo di classe deve
// essere allocato esplicitamente fuori da tutto!

int main () {
    puzzle p;
    // oppure p(...); p rappresenta lo stato iniziale del gioco
    puzzle :: set_min_moves(10);
    // non saranno considerate eventuali soluzioni
    // con un numero di mosse > 10
    STACK<STACK<puzzle>> S;
    p.search_shortest(S); // S è passato per riferimento
    // tutte le soluzioni più brevi sono state copiate in S
    int ns = S.height();
    std::cout << "Numero di soluzioni minime trovate = ";
    std::cout << ns << std::endl;
    for (; ns > 0; ns--) {
        S.pop().print();
        std::cout << "Fine soluzione n. " << ns << std::endl;
    }
    // Come si può ricordare e stampare qui il numero
    // di mosse di ciascuna soluzione minima trovata?
    return 0;
}
```

Il ciclo **for** finale (sostituibile, in maniera assai più sbrigativa, con la semplice istruzione `S.print();`) provvede alla stampa della pila di soluzioni minime trovate (in ordine inverso): tale operazione è fatta prelevando e stampando una soluzione alla volta, affinché queste siano separate come conviene per una lettura più agevole. Il metodo (pubblico) `search_shortest` della classe `puzzle` si servirà di un metodo ausiliario (privato) con un ulteriore parametro. Entrambi sono definiti alla pagina successiva.

Ricordiamo che qualsiasi metodo (anche quando è applicato a una particolare istanza della classe) può accedere ai campi di classe; un metodo di classe può accedere *esclusivamente* a campi di classe.

Completere il codice per il solitario di Tait è ora un compito piuttosto facile. Si può provare così che, nel caso illustrato con  $n = 4$ , la soluzione col minor numero di mosse è unica e ne prevede soltanto 4. Se numeriamo le caselle da 0 a 9, iniziando da sinistra, possiamo codificare ciascuna mossa con un numero: la mossa  $i$  consiste nel prendere le due pedine che occupano le caselle  $i$  e  $i + 1$  e spostarle nei due posti liberi. Allora la soluzione più breve è data da: [7, 4, 1, 8]. Per inciso, ci sono poi 7 soluzioni in 5 mosse; in generale, di quali modifiche necessita il programma per trovare (e contare) *tutte* le soluzioni che comportano una mossa in più rispetto alle più brevi?

```

void puzzle :: search_shortest (STACK<STACK<puzzle> > & S) const {
    STACK<puzzle> L; // la "pila di stati" L è creata vuota
    L.push(*this); // lo stato iniziale è copiato su L ...
    aux_search_shortest(S, L);
    // ... e alla fine in L resterà la copia del solo stato iniziale
}

void puzzle :: aux_search_shortest
(STACK<STACK<puzzle> > & S, STACK<puzzle> & L) const {
    // In cima alla pila L c'è lo stato corrente (l'ultimo
    // inserito), in fondo quello iniziale (il primo inserito).
    int moves = L.height() - 1; // il numero di mosse finora fatte
    if (moves <= min_moves)
        if (success()) {
            if (moves < min_moves) {
                S.make_empty(); // dimentica le soluzioni più lunghe
                min_moves = moves; // anche senza invocare set_min_moves
            }
            S.push(L); // la soluzione trovata è copiata su S
        } else { // lo stato corrente non è finale
            for (int i = 1; i <= N; i++)
                if (is_legal(i)) {
                    puzzle c = *this;
                    c.make_move(i);
                    if (! L.is_in(c)) {
                        L.push(c);
                        c.aux_search_shortest(S, L);
                        L.pop();
                    }
                }
        }
    }
    // else ... non fa nulla: il numero di mosse min_moves è stato
    // superato, quindi abbandona la ricerca dallo stato corrente!
}

```

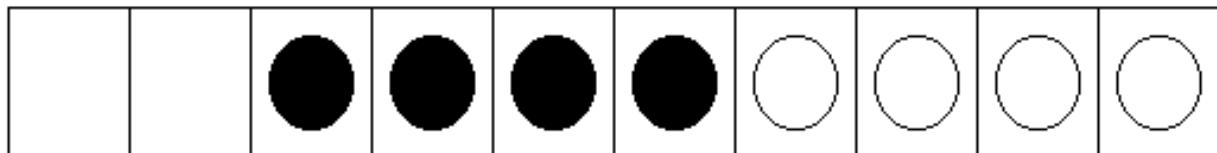
Un fatto interessante: immaginando le caselle disposte in modo *circolare*, cosicché facendo un passo a sinistra della 0 si arriva alla 9, per risolvere il problema basta spostare ad ogni mossa le due pedine che occupano il secondo e il terzo posto alla sinistra delle due caselle libere... Questa idea funzionerà anche per  $n > 4$ ? E vi sarà sempre una sola soluzione di lunghezza minima?

La risposta ad entrambe le domande è negativa. Si può provare però che, per  $n \geq 4$ , occorrono e bastano  $n$  mosse. Invece, per  $n = 3$ , ci sono due soluzioni in *quattro* mosse: [2, 5, 0, 6] e [5, 2, 0, 6], mentre le soluzioni in 5 mosse sono 6 (si veda la tabella in alto alla pagina seguente).

Se cambiamo la configurazione finale invertendovi i colori, cioè ponendovi – da sinistra – prima le pedine bianche, poi le nere e infine le due caselle vuote, allora l'unicità della soluzione più breve è comunque perduta; anzi, il numero di soluzioni di lunghezza minima (che richiedono una mossa in più rispetto al caso illustrato sopra) aumenta piuttosto rapidamente con  $n$ .

<i>n</i>	soluzioni in <i>n</i> mosse	
4	1	[ 7, 4, 1, 8]
5	1	[ 9, 4, 7, 1, 10]
6	1	[11, 8, 3, 7, 1, 12]
7	2	[13, 6, 9, 4, 10, 1, 14] [ 9, 6, 13, 4, 10, 1, 14]
8	16	[ 5, 12, 15, 8, 1, 11, 4, 16] [ 5, 10, 15, 12, 1, 9, 4, 16] più altre sette permutazioni di entrambe le sequenze, dove la 4 <sup>a</sup> e ovviamente la 8 <sup>a</sup> mossa rimangono fisse

Un'altra variante consiste nel lasciare vuote le prime due caselle nello stato finale, anziché le ultime due, com'è mostrato in figura per  $n = 4$ : questo caso fu proposto nella gara *Kangourou dell'Informatica* di marzo 2010.



Qui abbiamo ancora un'unica soluzione di lunghezza minima che però consta di *cinque* mosse: [8, 1, 4, 7, 0] (si noti la simmetria rispetto alla versione originale), mentre le soluzioni in 6 mosse sono ben 27.

Anche per  $n = 3$  adesso c'è una sola soluzione, sempre in *quattro* mosse: [4, 1, 6, 0]; le soluzioni in 5 mosse sono di nuovo 6.

<i>n</i>	soluzioni in ( <i>n</i> + 1) mosse	
4	1	[ 8, 1, 4, 7, 0]
5	3	[10, 1, 6, 3, 9, 0] [ 6, 1, 10, 3, 9, 0] [ 9, 4, 10, 1, 6, 0]
6	4	[ 4, 1, 12, 9, 5, 11, 0] + tre permutazioni
7	22	[13, 4, 11, 5, 14, 1, 8, 0] [ 5, 14, 11, 4, 13, 1, 8, 0] [ 4, 1, 8, 13, 10, 5, 14, 0] + nove permutazioni [ 4, 1, 8, 11, 14, 5, 12, 0] + nove permutazioni

Ovviamente l'ultima mossa è sempre 0: infatti, per lasciare vuote le prime due caselle, bisogna spostare le prime due pedine!

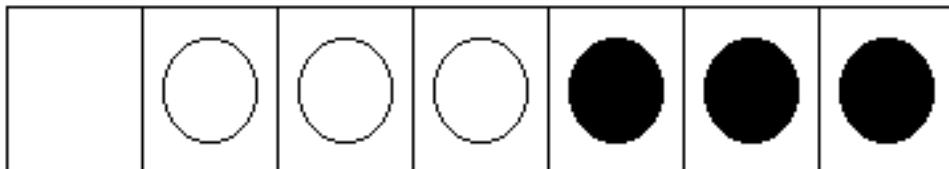
Se cambiamo ancora questo stato finale invertendovi i colori, allora l'unicità della soluzione più breve si mantiene soltanto per il caso “minimo”:  $n = 3$ , in 4 mosse; poi, per  $n \geq 4$ , il numero di soluzioni di lunghezza minima, sempre in  $(n + 1)$  mosse, aumenta assai rapidamente.

In tutte le varianti che abbiamo considerato, l'unica cosa che cambia è la configurazione finale; per realizzarle basterà ridefinire soltanto il corpo del metodo success, la cui funzione è appunto quella di riconoscere lo stato finale del gioco.

Lucas propone ulteriori varianti, imponendo la condizione di *invertire* ad ogni mossa l'ordine delle due pedine della coppia spostata. Che cosa succederà?

Nello stesso volume delle *Récréations mathématiques*, che costituisce il terzo tomo della serie postuma, si trovano diversi solitari da non trascurare, come *taquin* e “la presa della Bastiglia”. (E non dimentichiamo che a Lucas spetta pure il merito di aver descritto per primo la procedura di visita in profondità di un grafo!)

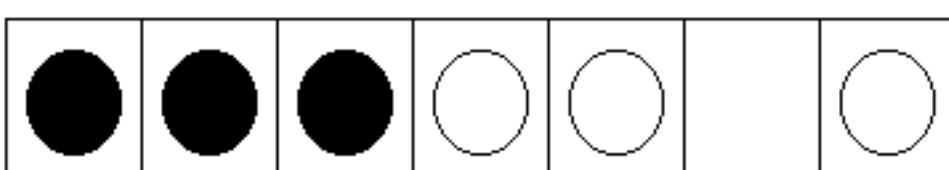
Restando invece nell'ambito degli innumerevoli giochi che si possono inventare con pedine di due colori disposte in fila, ricordiamo ancora quello proposto da Wayne A. Wickelgren in *How to Solve Problems* (Freeman, 1974) e successivamente attribuito a Dudeney.



L'obiettivo è avere tutte le pedine bianche a destra di tutte le nere; non importa quale sia la casella che alla fine rimane vuota: qui c'è un grado di libertà sullo stato finale!

Le mosse consentite sono:

- lo spostamento di una pedina nella casella libera ad essa adiacente;
  - il salto di una pedina sopra a una o a due altre pedine, per finire nella casella libera.
- Con 9 mosse si può ottenere la configurazione qui sotto riportata: come?



In *Principles of Artificial Intelligence* (Tioga Publishing Company, 1980) Nils J. Nilsson lascia come esercizio la ricerca della soluzione con minor costo, assegnando costo 1 sia allo spostamento nella casella libera adiacente, sia al salto sopra a una pedina, e costo 2 al salto sopra a due pedine. Sarà di nuovo quella con le stesse 9 mosse la soluzione ottima?

Come si generalizza questo gioco?

Concludiamo il paragrafo con una doverosa considerazione. In generale, uno strumento come quello che abbiamo progettato, che esplora a tappeto tutte le possibilità senza alcun criterio, può rivelarsi utile finché la dimensione del problema rimane limitata: di solito, infatti, la complessità rispetto al tempo è di tipo *esponenziale* in qualche parametro legato a tale dimensione. Sicché spesso accade che il tempo necessario diventi presto proibitivo... Vale la pena allora di applicare un po' di ingegno nel tentativo di capire quale sia la forma generale di una soluzione, ammesso naturalmente che ciò sia possibile!

### **Le sei rane e altri giochi di cambio di posizione.**

Per una ventina d'anni il già citato Dudeney scrisse e illustrò una rubrica di intrattenimento, *Perplexities*, per la popolarissima rivista *The Strand Magazine*, la stessa in cui apparvero, a iniziare dal 1891, i racconti di Arthur Conan Doyle con Sherlock Holmes come protagonista e con le famose illustrazioni di Sidney Paget.

Dudeney, autodidatta, risolse pure alcune questioni di matematica laddove gli specialisti non erano ancora riusciti, ma la sua notorietà è legata soprattutto ai rompicapi, la maggior parte dei quali traevano comunque origine da problemi matematici. Tra i tanti puzzle da lui ideati – troppo attraenti e pertinenti per non essere ricordati in questo capitolo! – ne ho scelto ancora qualcuno.

Le sei rane – rappresentate dalle pedine numerate, disposte come in figura – sono state ammaestrate per cambiare il loro ordine: alla fine dei movimenti, dopo la casella vuota a sinistra, dovranno comparire i numeri da 6 a 1, in ordine discendente.



Sono permessi lo spostamento nella casella libera adiacente e il salto sopra a una sola rana per finire nella casella libera. Come faranno a raggiungere l'assetto finale nel minor numero di mosse?

Indicando ciascuna mossa col numero della rana che si sposta o salta (si noti che non v'è ambiguità, in quanto c'è un solo posto libero) la soluzione più breve è formata da 21 mosse: dapprima si ripete tre volte la sequenza 2, 4, 6, 5, 3, 1, e infine si muovono ancora una volta 2, 4 e 6.

Dudeney suggerisce il modo (naturale) di generalizzare il gioco, aggiungendo caselle e rane a destra, fino al numero  $n$ , e ne dà per primo la soluzione completa (pubblicata nel 1917).

Se  $n$  è pari, la soluzione si ottiene scrivendo  $n/2$  volte la sequenza costituita dai numeri pari a salire (da 2 a  $n$ ) seguiti dai numeri dispari a scendere (da  $n-1$  a 1), e aggiungendo infine un'ultima volta i numeri pari a salire (da 2 a  $n$ ). In totale, dunque, le mosse sono  $n \cdot (n+1)/2$ , di cui  $n$  sono spostamenti e le altre salti.

Se  $n$  è dispari, la soluzione si ottiene scrivendo  $(n - 1)/2$  volte la sequenza costituita dai numeri pari a salire (da 2 a  $n - 1$ ) seguiti dai numeri dispari a scendere (da  $n$  a 1), proseguendo poi con i numeri pari a salire da 2 a  $n - 3$ , i numeri dispari a scendere da  $n$  a 3, e aggiungendo infine tutti i numeri (sia pari sia dispari) a salire da 2 a  $n - 1$ . In totale, dunque, le mosse sono  $n \cdot (n + 3)/2 - 4$ , di cui  $2 \cdot (n - 2)$  sono spostamenti e le altre salti.

Ad esempio, per  $n = 7$ , si ripete tre volte la sequenza 2, 4, 6, 7, 5, 3, 1, dopodiché si muovono i numeri 2, 4, 7, 5, 3, e infine 2, 3, 4, 5, 6: in tutto sono quindi 31 mosse.

In un successivo problema, che Dudeney chiamò “l’enigma della cavalletta”, le caselle sono disposte *circolarmente*: se preferite, lasciate la figura così com’è, ma tenete presente che prima della casella vuota vi è la pedina numero 6, ovvero dopo la pedina numero 6 vi è la casella vuota...

Dudeney lo propone con 12 pedine e chiede soltanto qual è il minor numero di mosse per invertire l’ordine delle pedine; ma sarà possibile stabilirlo senza dare una soluzione e dimostrare che è di lunghezza minima?

In effetti, egli dà una soluzione in 44 mosse e avvisa che, in generale, il problema è assai difficile da risolvere.

A conclusione dei suoi ragionamenti, afferma che i casi  $n = 2, 3$  e  $4$  richiedono rispettivamente 3, 3 e 6 mosse; per  $n$  dispari  $\geq 5$  occorrono  $(n^2 + 6n - 31)/4$  mosse, mentre per  $n$  pari  $\geq 6$  ce ne vogliono  $(n^2 + 4n - 16)/4$ , ma lascia al lettore il non facile compito di scoprire da solo come determinarle!

Nel quesito “l’enigma di Twickenham” espone una variante – sempre “circolare” – con 10 pedine, ciascuna contenente una lettera della parola “Twickenham”, antico nome di un quartiere di Londra: 5 nere che si muovono in senso orario e 5 bianche che si muovono in senso opposto (si veda la figura in alto alla pagina successiva). Una pedina può scavalcarne una sola, purché dell’altro colore.

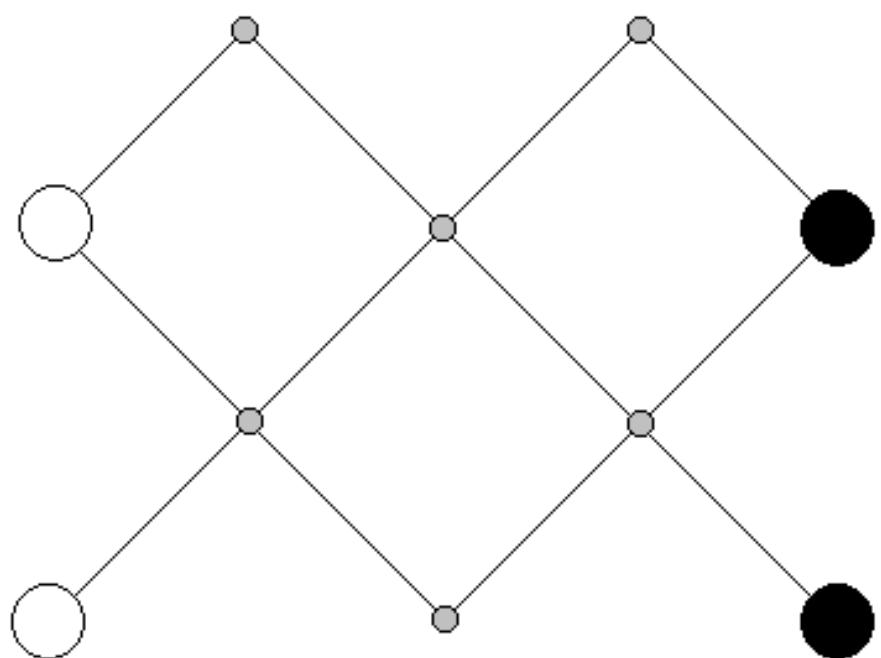
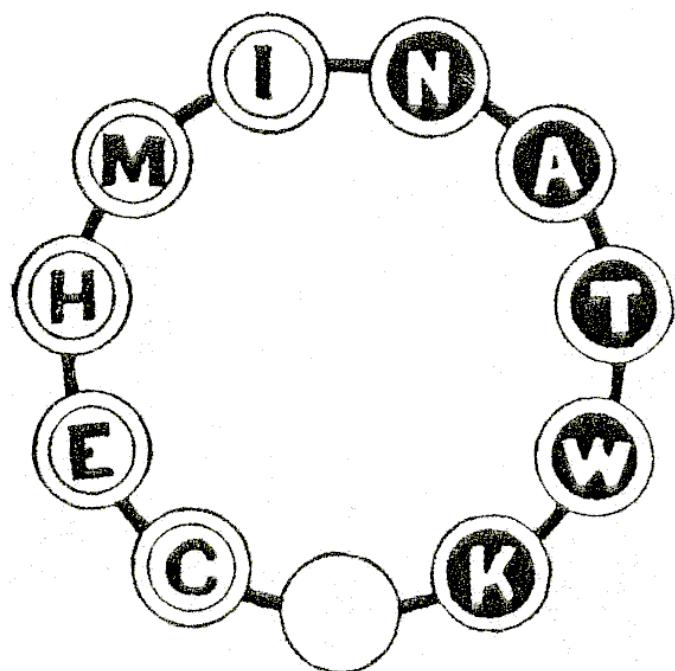
Alla fine si dovrà leggere la parola “Twickenham” in senso orario, rimanendo vuota la stessa casella che già era libera all’inizio del gioco.

Dudeney ne dà una soluzione in 26 mosse (la sequenza non è unica).

Più interessante, e in verità piuttosto complicato, è un piccolo puzzle sempre con le pedine, ma questa volta su una rete bidimensionale (si veda la figura in basso alla pagina successiva).

Le pedine bianche e quelle nere si devono scambiare i posti. Ad ogni mossa, una pedina – senza saltarne altre – può spostarsi di uno, due o tre segmenti, se possibile, purché siano sulla stessa linea retta. L’unico vincolo è che, fatta una mossa, due pedine di diverso colore non si trovino mai su una stessa linea retta.

Dudeney ne dà una soluzione in 18 mosse.



## Il gioco del quindici.

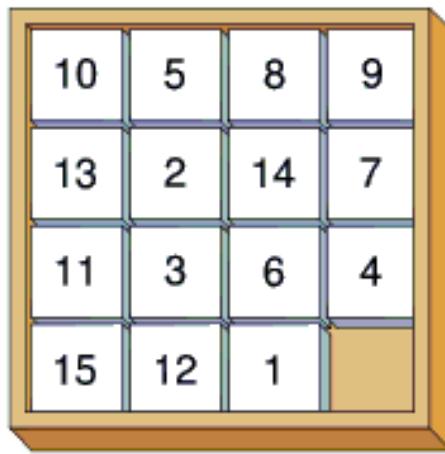
Dall'altra parte dell'oceano, un altrettanto prolifico creatore di rompicapi, lo statunitense Samuel (Sam) Loyd (1841-1911), era giunto al grande pubblico attraverso i giornali e le riviste che riportavano i suoi enigmi – e la popolarità di alcuni di essi resistette al trascorrere del tempo. Dopo la sua morte, il figlio Sam Jr. ne raccolse oltre 5000 in un unico volume, la *Cyclopedia of Puzzles*, pubblicata a New York da The Lamb Publishing Company nel 1914: si tratta certamente di una delle più ricche e avvincenti raccolte di questo genere, che comprende quesiti di vario tipo – assai carini quelli che coinvolgono le abilità visive! Si può trovarne la riproduzione in rete, all'indirizzo <http://www.mathpuzzle.com/loyd/>.

Loyd fu senza dubbio un brillante inventore di puzzle, un vero maestro nell'arte di far emergere gli aspetti più sorprendenti dalle questioni talvolta un po' astruse o bizzarre, che divertivano tanto i suoi lettori, ma – come riteneva Martin Gardner – nei problemi di natura squisitamente matematica Dudeney lo superò largamente. Bisogna ricordare però che Loyd fu pure un eccellente compositore di problemi di scacchi e un appassionato cultore di altri giochi, tra cui il *tangram*.

Il gioco che presentiamo in questo paragrafo *non* è stato inventato da Loyd, sebbene egli abbia contribuito in una certa misura a ravvivarne l'ampia popolarità che risaliva a sedici anni prima; l'inventore oggi riconosciuto è Noyes P. Chapman, all'epoca direttore dell'ufficio postale di Canastota, New York.

Alcuni di voi forse ricorderanno questo puzzle: ancora anni fa ne erano venduti degli esemplari, sebbene di solito non di legno, come in origine, ma di plastica.

All'interno di un quadrato possono scorrere 15 tessere, a loro volta quadrate e numerate da 1 a 15, sì da formare quattro file lasciando libero un posto; un esempio di configurazione (iniziale) è mostrato in figura.



Il gioco consiste nel far scorrere le tessere, con l'obiettivo di raggiungere lo stato (finale) in cui esse dovranno trovarsi ordinate da 1 a 15, per righe da sinistra a destra e dall'alto al basso, col posto vuoto nell'angolo in basso a destra.

Nel caso illustrato dalla precedente figura, la prima mossa consisterà nello spostare la tessera 1 verso destra o altrimenti la tessera 4 verso il basso.

Anzitutto ci si deve chiedere: sarà possibile? Supponiamo di poter preparare uno stato iniziale disponendo in maniera arbitraria le 15 tessere – quanti sono dunque i possibili stati iniziali? Con una sequenza di scorrimenti, portiamo il posto vuoto nell’angolo in basso a destra, se già non c’è, e immaginiamo ora di poter scambiare il posto di due tessere a piacere, ripetutamente. Contiamo il numero degli scambi che occorrono per ottenere lo stato finale: se questo numero è dispari, allora lo stato da cui eravamo partiti era “impossibile da risolvere”; altrimenti, cioè se tale numero è pari, allora lo stato iniziale era “risolvibile”.

Questi due fatti furono dimostrati in due contributi apparsi sull’*American Journal of Mathematics*, il primo a firma di Wm. Woolsey Johnson, il secondo – compito più impegnativo! – di William E. Story. Il numero della rivista è datato dicembre 1879, ma la sua pubblicazione avvenne nella primavera successiva, nel pieno del “boom” di questo fortunato puzzle; e nello stesso periodo anche un matematico tedesco, Hermann Schubert, ottenne lo stesso risultato di Johnson...

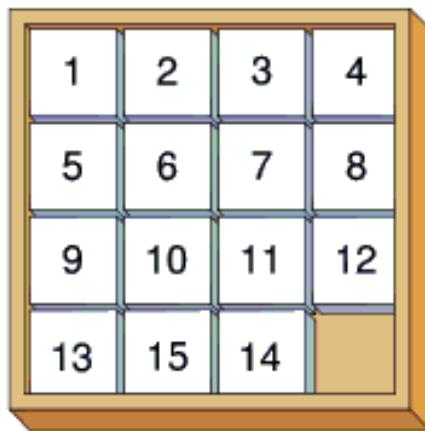
Ne consegue che la metà degli stati di partenza è impossibile da risolvere. Nel caso poc’anzi illustrato, andando in ordine, gli scambi devono essere fatti tutti e 14: si scambiano 10 con 1, 5 con 2 eccetera; 14 è pari, e ciò garantisce la raggiungibilità dello stato finale. E quante mosse occorreranno per arrivarci? Difficile stabilirlo; comunque, oggi si sa che le configurazioni più “lunghe” da risolvere ne richiedono 80 (nel senso che 80 mosse sono necessarie): e di tali configurazioni ce ne sono soltanto 17 (questo è un risultato più recente), due delle quali ad esempio sono:

– 12 9 13	– 12 9 13
15 8 10 14	15 11 10 14
11 7 6 2	7 8 5 6
4 3 5 1	4 3 2 1

Ma torniamo alla storia del puzzle. Dal dicembre del 1879, nel Massachusetts, ne furono prodotti e messi in commercio degli esemplari in legno, dove le tessere non erano costrette a scivolare in scanalature (a differenza delle edizioni più moderne) e le istruzioni erano semplicemente: « Place the blocks in the box irregularly, then move until in regular order ». Venduto col nome di “Gem Puzzle”, nel giro di pochi mesi divenne una vera e propria mania negli Stati Uniti, in Canada e in Europa, ancor più del cubo di Rubik giusto un secolo dopo. Fu oggetto di articoli su riviste matematiche e interi libri, anche piuttosto recenti, come il magnifico *The 15 Puzzle: How it Drove the World Crazy*, di Jerry Slocum e Dic Sonneveld (Slocum Puzzle Foundation, 2006). Ebbe un revival verso la fine degli anni ’40 del Novecento, che durò ancora qualche decennio.

Il 4 gennaio del 1896, su *The Illustrated American*, Sam Loyd propose ai suoi lettori la configurazione riportata alla pagina successiva, dove soltanto il 14 e il 15 sono scambiati rispetto allo stato finale, e offrì 1000 dollari da dividere equamente tra tutti coloro che l’avessero risolta, ben sapendo che il problema non ammette soluzione...

Chiaramente contava sul fatto che al suo grande pubblico non fosse noto il risultato ottenuto da Johnson e Schubert parecchi anni prima!



E in effetti, stando a quanto raccontò lo stesso Loyd, qualcuno addirittura uscì di senno per gli sforzi infruttuosi prodigati nei tentativi: «Il premio di 1000 dollari da me offerto a chi avesse trovato la soluzione esatta non è mai stato reclamato da nessuno, sebbene migliaia di persone affermino di essere riuscite nell’impresa [...] Il mistero di questo gioco sta nel fatto che nessuno sembra ricordare l’esatta sequenza di mosse con le quali è *sicuro* di essere riuscito a risolverlo. » Tuttavia, Loyd dimenticò – o forse non sapeva – che già il 3 marzo del 1880 *The Daily Graphic* aveva pubblicato lo stesso schema!

Non solo: una soluzione era stata data ancor prima, alla fine di gennaio del 1880! «Ma allora non è vero che è impossibile!» esclamerà a questo punto qualcuno dei lettori. Il trucco è presto svelato: se la formulazione del problema prevede soltanto che i numeri alla fine siano disposti «in regular order», allora il posto vuoto può essere lasciato in alto a sinistra; e così i numeri si riescono a disporre per righe, fino al 15 in basso a destra – fate la verifica!

Ma non è ancora finita: a quanto pare, in quello stesso mese di gennaio del 1880, a “boom” da poco iniziato, un dentista di Worcester, Massachusetts, probabilmente per farsi un po’ di pubblicità, aveva lanciato una sfida a risolvere lo stesso problema, ma precisando correttamente che, alla fine, il posto vuoto doveva trovarsi dopo il 15; come premio aveva offerto una somma in denaro (100 dollari, elevati poi giusto a 1000) oltre a un corredo completo di denti finti!

### Ricerca euristica di una delle soluzioni più brevi.

Una partita al (o meglio: un’istanza del) gioco del 15 può essere riguardata come una permutazione delle 15 tessere numerate e del posto vuoto (quindi 16 elementi in tutto) su una griglia 4×4, che deve essere riordinata – ammesso che ciò sia possibile – facendo scivolare ad ogni mossa nello spazio libero una delle tessere ad esso adiacenti.

Per rendere più interessante il gioco, vogliamo trovare una soluzione *minima*, o più breve, ossia una sequenza che conduca allo stato finale nel minor numero di mosse.

Come si può procedere? Gli stati risolvibili sono  $16!/2 \approx 10^{13}$ . Ormai da un po' di anni, non sono poi troppi per pensare di impiegare un metodo di ricerca bruta *depth first*: in termini di tempo, è possibile enumerarli tutti, data la velocità di calcolo degli attuali computer. Ciò che ancor oggi non si può fare è immagazzinare nella memoria principale tutti questi stati (che costituiscono soltanto i nodi del grafo di gioco, in cui ciascun arco collega due nodi separati da una mossa): quanto spazio sarà richiesto?

Può comunque tornare utile un algoritmo euristico che attui, ad esempio, una ricerca *best first* (cioè che dia la precedenza alle mosse "più promettenti", che danno più speranza di giungere vicino all'obiettivo, secondo un'opportuna funzione di valutazione) in cui il "costo" di una mossa sia calcolato come somma del numero di mosse già fatte a partire dallo stato iniziale e del numero *minimo stimato* di mosse che rimangono da fare per giungere allo stato finale. Se la stima non eccede mai il vero numero di mosse *necessarie* per ottenere una soluzione, allora il procedimento trova una delle soluzioni più brevi (se almeno una soluzione esiste) in un tempo che possiamo sperare ragionevole.

Nel nostro gioco, una stima può essere fatta sommando le "distanze Manhattan" (espresso in numero di posti sia in orizzontale sia in verticale) tra ciascuna tessera e la sua posizione finale: così si ottiene certamente un limite inferiore per il numero di mosse rimanenti, poiché ciascuna tessera dovrà muoversi almeno tante volte quanti sono per l'appunto i posti che la separano dalla sua posizione finale, e ogni mossa prevede lo spostamento di un'unica tessera. (Una scelta un po' più grossolana, ma pur sempre ammissibile, prende come stima semplicemente il numero di tessere fuori posto rispetto allo stato finale.)

Tuttavia, per realizzare una simile procedura in modo efficiente anche rispetto allo spazio di memoria richiesto, è necessario l'impiego di particolari tecniche, del tipo di quelle che vedremo tra due capitoli, per i giochi a due contendenti: *iterative deepening* e tagli, con raffinamenti vari... Qui non approfondiamo questi aspetti, e invitiamo i lettori interessati a consultare direttamente gli ormai classici lavori di Judea Pearl, Richard E. Korf, Alexander Reinefeld, Jonathan Schaeffer e altri.

Un'ultima osservazione: siccome conosciamo precisamente lo stato finale da raggiungere, potremmo anche pensare di applicare una ricerca *backward*, cioè che procede a ritroso partendo proprio dall'obiettivo...

## Versioni grandi e piccole.

Versioni più piccole del gioco del 15, in particolare il gioco dell'otto su una griglia  $3\times 3$ , si sono rivelate assai utili in passato come banco di prova per algoritmi euristici nel campo di quella disciplina che da oltre sessant'anni è nota come "intelligenza artificiale".

Il gioco del 24, su una griglia  $5\times 5$ , è ancora "sotto osservazione": ad esempio, si sa che le configurazioni più difficili richiedono, per essere risolte, un numero di mosse non noto, ma di certo non inferiore a 152 e non superiore a 205.

Comunque, dal 1986, si sa pure che il problema di trovare una delle soluzioni più

brevi in un gioco del  $N^2 - 1$ , su una griglia  $N \times N$ , è NP-arduo: come il lettore dovrebbe ricordare, ciò significa che, in generale, è tanto complesso quanto trovare un *tour* ottimo per il commesso viaggiatore.

Il gioco dell'otto è ormai da tempo affrontabile su un personal computer con un algoritmo *depth-first search*, di forza bruta, come quello che abbiamo formulato, in termini generali, a proposito del solitario di Tait. Per avere la certezza di trovare (tutte) le soluzioni più brevi a partire da un certo stato, assegneremo il valore 31 al campo di classe `min_moves`; in effetti, gli stati che richiedono il massimo numero di mosse in una sequenza risolutiva di lunghezza minima sono soltanto due:

8	6	7	6	4	7
2	5	4	8	5	-
3	-	1	3	2	1

ciascuno dei quali richiede 31 mosse e può essere risolto senza ridondanze in 40 modi diversi. Questo risultato è stato verificato dal nostro programma.

Se vogliamo che ciascuna delle  $9! = 362880$  permutazioni delle 8 tessere e del posto libero sia risolvibile, possiamo aggiungere come stato finale quello con i numeri ordinati per colonne:

1	2	3	1	4	7
4	5	6	2	5	8
7	8	-	3	6	-

sicché da qualsiasi configurazione si parta, si potrà arrivare a uno (e uno solo) di questi due stati finali. Come al solito, basterà modificare la definizione del metodo `success`.

Cambiando ancora il corpo di questo metodo, ci siamo divertiti a trovare un quadrato magico a partire dallo stato finale classico, cioè quello a sinistra. Il programma ha stabilito che in 17 mosse si trova

7	2	3
-	4	8
5	6	1

dove, interpretando lo spazio vuoto come lo zero, ciascuna riga o colonna o diagonale maggiore ha somma 12. Come si può formare un quadrato magico di somma 30 nel gioco del 15?

Nel gioco dell'otto, trovare una soluzione (anche non minima) è piuttosto semplice; sicché, per risolvere un'istanza del gioco del 15 si può procedere così: dapprima si sistemano ai loro posti definitivi le tessere coi numeri 1, 2, 3, 4, 5, 9 e 13, che formano i lati in alto e a sinistra; dopodiché il gioco si riduce a un'istanza di quello dell'otto. Ciò velocizza il processo di ricerca di una soluzione (qualunque) in modo piuttosto considerevole.

Una versione “didattica” è limitata a un rettangolo  $2 \times 3$ : è il gioco del 5, con soli 360 stati risolvibili; l’unico stato più lungo da risolvere è quello con le due righe scambiate rispetto allo stato finale, ossia:

$$\begin{array}{ccc} 4 & 5 & - \\ 1 & 2 & 3 \end{array}$$

che richiede 21 mosse. Abbiamo ottenuto questo risultato con un programma che genera tutte le permutazioni risolvibili e a ciascuna applica l’algoritmo *depth first* di cui sopra: non è difficile da progettare!

Dal punto di vista della realizzazione informatica, uno stato può essere rappresentato da una matrice di interi contenente i numeri relativi alle tessere e lo zero al posto della casella libera. Per semplicità, si può pensare che a muoversi sia proprio la “tessera 0”, la quale può spostarsi di una posizione in verticale o in orizzontale, scambiandosi con una delle tessere adiacenti. Quindi, ad ogni mossa, sono al massimo quattro le possibilità che si presentano.

Nell’area privata, oltre alla matrice di  $m$  righe per  $n$  colonne, sono utili due campi che ricordino gli indici di riga e colonna dell’elemento che contiene lo zero; se la matrice è statica, la definizione del costruttore di copia non serve neppure, perché il costruttore di *default* del C++ svolge il compito correttamente.

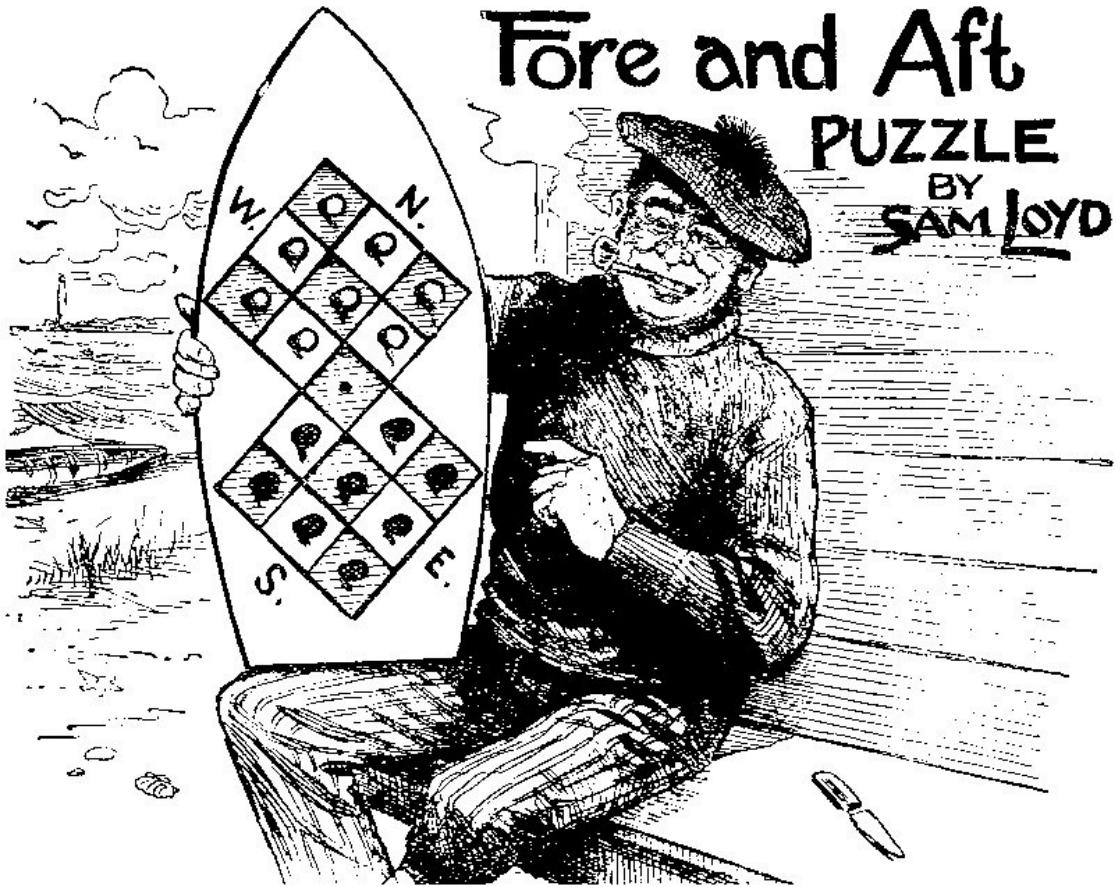
Lo stato iniziale può essere trasmesso al costruttore appropriato per mezzo di una matrice o di un *file*, contenente una permutazione (risolvibile) degli interi da 0 a  $m^*n-1$ ; lo stato finale (di successo) deve contenere lo zero in basso a destra e gli altri numeri ordinati per righe.

### A prua e a poppa, ovvero rane e rospi bidimensionali.

Il gioco con cui ci piace concludere l’argomento “rompicapi” riprende il primo che abbiamo esaminato in questo capitolo e lo espande su una scacchiera di forma inusuale, costituita da due quadrati  $3 \times 3$  che hanno in comune una casa angolare.

Per intenderci meglio, riportiamo alla pagina seguente l’illustrazione originale che accompagnava la presentazione fatta da Sam Loyd ai suoi lettori. Egli – alla pari di Dudeney – aveva l’abitudine di aggiungere una storiella alla spiegazione. Di questo gioco, che chiamò *Fore and Aft* (“A prua e a poppa”, come dire “Avanti e indietro”), scrisse che fu inventato da un marinaio inglese, il quale trascorse gran parte della vita sull’isola di Staten Island, nella baia di New York, dove ai turisti vendeva dei giochi da lui stesso intagliati nel legno. In particolare, costui era orgoglioso proprio di questa sua creazione, che fu portata in Inghilterra e là divenne assai popolare col nome di “gioco del sedici inglese”.

In realtà, questo gioco – da annoverare fra i tanti che ricordano il solitario classico, alla francese o all’inglese – pare sia stato pubblicato per la prima volta dall’inglese Walter William Rouse Ball (1850-1925) nel 1892, e certo è che fu in auge in tarda età vittoriana, per cui in Italia divenne noto come “piccolo solitario vittoriano”.



## Fore and Aft PUZZLE BY SAM LOYD

Le regole ricalcano quelle di "rane e rospi": guardando la figura in modo da leggere correttamente le quattro lettere, le pedine nere possono muoversi soltanto verso nord (N) o verso ovest (W), le bianche soltanto verso sud (S) o verso est (E); non sono permessi movimenti in diagonale. Una pedina può spostarsi di una casa, oppure può scavalcare una pedina di colore opposto, per andare ad occupare la casa libera. Lo scopo è sempre lo scambio di posti tra le pedine dei due colori.

In virtù di queste regole, durante una "partita" può crearsi una situazione di blocco, ma non può ripetersi una stessa posizione: pertanto è inutile controllare se lo stato raggiunto è già stato visitato. Poiché in ogni momento vi è una sola casa libera e poiché non sono messe mosse in diagonale, né a ritroso, ne consegue che, ogni volta che si deve fare una mossa, non più di due pedine di ciascun colore hanno la possibilità di muoversi.

Tuttavia, a differenza di "rane e rospi", qui ci sono parecchie soluzioni – ma quante saranno? – ovviamente simmetriche a gruppi di quattro, a seconda che inizi il bianco o il nero, ciascun colore con l'una o con l'altra delle pedine che possono iniziare.

Ball, riportandolo nella sua opera encyclopedica *Mathematical Recreations and Essays* del 1892 (pubblicata a Londra da Macmillan), ne dà una soluzione in 51 mosse; nella terza edizione, del 1896, ne dà un'altra di 48 mosse. Ma nella quinta edizione, del 1911, pubblica quella che egli dice trovata da Dudeney nel 1898: essa consta di sole 46 mosse ed è particolarmente elegante per la sua simmetria. Prima di esaminarla, voglio lasciarvi un po' di tempo per pensarci, sebbene il compianto

Giampaolo Dossena abbia scritto che chi riesce a trovare una soluzione di una cinquantina di mosse possa già ritenersi un valente giocatore!

Loyd ne trovò una in 47 mosse, ma nella *Cyclopedie* del 1914 (alla p. 353) non è stampata correttamente: vi mancano cinque mosse, dalla 32-esima alla 36-esima (secondo la notazione dello stesso Loyd: East, South, North, East, North).

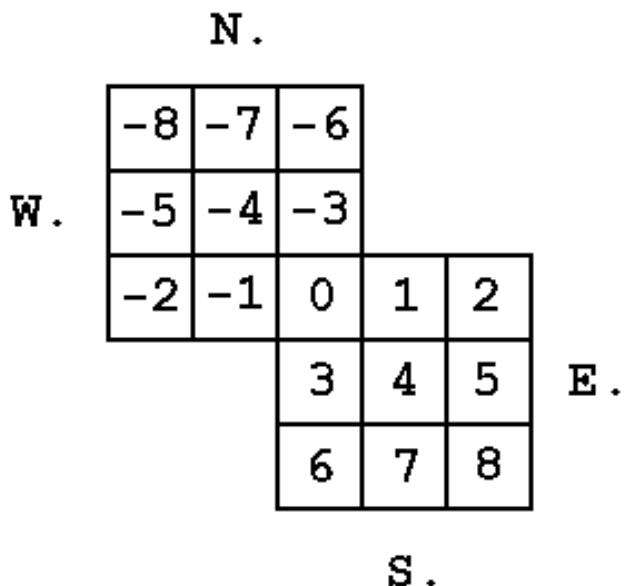
Oltre trent'anni fa è stato provato che 46 mosse sono necessarie e che esistono più soluzioni minime (in 46 mosse). Lasciamo, a chi desidera conoscerle, il compito di predisporre un programma per calcolare quali e quante siano le soluzioni minime – sono tante! Siccome non è possibile ripetere una posizione, ha pure senso chiedersi poi di quante mosse siano costituite le soluzioni *più lunghe*: la risposta è 58.

In rete si trovano due edizioni dello stupendo libro di Ball: una trascrizione della quarta edizione del 1905 e una riproduzione della sesta edizione del 1914; si vedano:

<http://www.gutenberg.org/ebooks/26839>,

<http://www.archive.org/details/mathematicalrecr00ball>.

Per rappresentare in maniera efficace l'eccellente soluzione di Dudeney, numeriamo le case della scacchiera da -8 a 8, come illustrato in figura; le pedine bianche occupano le case negative, quelle nere le case positive, la casa 0 è inizialmente libera.



Senza creare ambiguità, possiamo descrivere ciascuna mossa indicando il numero corrispondente alla casa da cui parte la pedina che si muove. Facciamo iniziare il nero con la pedina che sta nella casa 1 (sebbene non necessario, in neretto sono evidenziati i salti):

1 **-1** -2 0 3 **-3** -6 0 6 7 **1 -1** 0 2 5 3 **-3** -4 -1 **-7** -8 **-2** 0  
2 8 7 1 4 3 **-3** **-5** -2 0 1 **-1** **-7** -6 0 6 3 **-3** 0 2 1 **-1** 0

Su ciascuna linea sono scritte 23 mosse: a metà partita la posizione dei due colori è simmetrica. Se prendiamo le prime 22 mosse, le ribaltiamo e cambiamo i segni,

otteniamo le prime 22 della seconda linea! Anche la distribuzione dei salti sulle due linee è simmetrica. E questa è solo una delle 2476 soluzioni più brevi che iniziano con la mossa 1, alla quale deve necessariamente seguire il salto -1. Invece, le soluzioni più lunghe che iniziano con le stesse due mosse sono soltanto 75. Dovrebbe essere evidente l'analogia con la soluzione di “rane e rospi”.

## **Appendice: una definizione completa della classe parametrica `STACK<ITEM>`.**

Di seguito è riportata la *definizione completa* della classe parametrica, denominata `STACK<ITEM>`, che abbiamo usato nel presente capitolo.

Ciascuna istanza di questa classe è ottenuta specificando un *tipo* (che a sua volta può essere una classe) da sostituire al parametro `ITEM`, e in ogni momento della sua vita rappresenterà una pila (omogenea) di oggetti del tipo specificato.

Si tratta di una classe con *allocazione dinamica* di memoria: lo spazio che serve per creare e “impilare” un nuovo oggetto deve essere trovato nella memoria *heap*, al momento opportuno, dall’operazione `new`.

Per manipolare a piacere istanze della classe `STACK`, trattandole come *first-class objects*, abbiamo definito il *costruttore di copia deep* insieme con l’appropriato *operatore di assegnazione = (overloaded)*, necessari per escludere problemi di *aliasing* (che potrebbero verificarsi qualora pile distinte condividessero uno stesso oggetto).

In generale, il costruttore di copia interviene (automaticamente) in tre occasioni:

- nella “dichiarazione con inizializzazione”, cioè quando una nuova istanza è allocata in memoria ed è esplicitamente indicato di attribuirle come valore iniziale una copia di un’altra istanza (creata in precedenza);
- nel “passaggio per valore”, cioè quando un’istanza è argomento di una chiamata a funzione (laddove sia prevista tale modalità di passaggio), e quindi una sua copia è usata per inizializzare il parametro locale;
- nella “restituzione del risultato”, cioè quando un’istanza è argomento di `return` in una funzione, e quindi una sua copia è trasmessa come risultato esplicito al chiamante.

Poiché in C++ non c’è un *garbage collector* che si attiva automaticamente, abbiamo definito il *distruttore* che liberi la memoria occupata dai relativi *frame*, subito prima che un’istanza della classe `STACK` sia deallocated. È bene ricordare che il distruttore è invocato automaticamente su tutte le istanze locali (compresi i parametri per valore) quando l’esecuzione di una funzione termina; si deve quindi prestare attenzione affinché non accadano “distruzioni” indesiderate (vedi i metodi `pop` e `=`).

I metodi `is_in`, `print_cout`, `print` e `height` realizzano operazioni che *non sono standard* per le pile (infatti, quando si parla di “pila”, di solito si intende una struttura di dati con accesso limitato all’*ultimo* elemento inserito). Abbiamo definito anche la classica funzione `is_empty`, che sarebbe necessaria in mancanza della funzione `height`.

Riassumiamo quali requisiti deve avere la classe parametro ITEM:

- il costruttore senza parametri (vedi il metodo push) e il costruttore di copia,
- gli operatori = e == (*overloaded*),
- l'operatore << (impiegato dalla libreria iostream per l'*output*) oppure un metodo print (per stampare la pila vi si applicherà, rispettivamente, print\_cout o print).

Le istanze di ITEM devono dunque essere *first-class objects*.

Avendo definito anche l'operatore == per la classe STACK (due pile sono uguali quando sono uguali, nell'ordine, i rispettivi oggetti ivi contenuti), possiamo creare e usare senza alcun problema pile di pile di *first-class objects*, pile di pile di pile di *first-class objects* eccetera.

```
#include <iostream>

template <class ITEM> class STACK; // forward

template <class ITEM> struct frame {
    // struttura parametrica che rappresenta una pila non vuota:
    // contiene infatti un oggetto di classe ITEM (la testa) e
    // il "resto" della pila (la coda), che è un'altra pila.
    ITEM head;
    STACK<ITEM> tail;
};

template <class ITEM> class STACK {

private:
    typedef frame<ITEM>* link;
    link sp;
    // unico campo privato; sp (= stack pointer) punta al frame che
    // sta in cima alla pila, e ha valore nullo se la pila è vuota
    void aux_assign (STACK<ITEM> & s1, const STACK<ITEM> & s2) {
        // metodo ausiliario (indipendente da *this) per l'assegnazione:
        // nella pila s1 (ora vuota) è costruita una copia della pila s2
        if (s2.sp != NULL) {
            s1.sp = new frame<ITEM>;
            s1.sp->head = s2.sp->head; // assegnazione tra ITEM
            // s1.sp->tail.sp = NULL; // già fatto dal costruttore!
            aux_assign(s1.sp->tail, s2.sp->tail);
        }
    }

public:
    STACK<ITEM> () {
        // costruttore senza parametri: inizializza l'istanza a pila vuota
        sp = NULL;
    }
    bool is_empty () const {
        // decide se la pila è vuota
        return sp == NULL;
    }
}
```

```

void push (ITEM x) {
    // inserisce l'ITEM x in un nuovo frame in cima alla pila
    link a = new frame<ITEM>;
    // è stato usato il costruttore senza parametri di ITEM
    a->head = x; // qui è fatta un'assegnazione tra ITEM
    a->tail.sp = sp;
    sp = a;
}

ITEM pop () {
    // operazione non totale; se la pila non è vuota,
    // estraе (e restituisce) l'ITEM sulla cima della pila
    if (sp != NULL) {
        ITEM x = sp->head;
        // è stato invocato il costruttore di copia di ITEM
        link a = sp;
        sp = sp->tail.sp;
        a->tail.sp = NULL;
        // si deve scollegare il frame, altrimenti il distruttore ...
        delete a;
        return x;
    }
    // else ... dovrebbe sollevare un'eccezione ...
}

bool is_in (ITEM x) const {
    // decide se l'ITEM x è presente nella pila
    if (sp == NULL) return false;
    if (sp->head == x) return true;
    // è stato fatto un confronto tra ITEM
    return sp->tail.is_in(x);
}

void print_cout () const {
    // stampa su cout tutti gli ITEM della pila,
    // andando a capo dopo ciascuno.
    // Nota: per poter usare questo metodo, l'operatore << deve
    //        essere definito per gli oggetti della classe ITEM.
    if (sp != NULL) {
        sp->tail.print_cout();
        std::cout << sp->head << std::endl;
        // la pila è stampata dal primo all'ultimo ITEM inserito,
        // per stamparla dall'ultimo inserito giù fino al primo,
        // basta invertire l'ordine di queste due istruzioni!
    }
}

void print () const {
    // stampa tutti gli ITEM della pila, andando a capo dopo ciascuno.
    // Nota: per poter usare questo metodo, un metodo print()
    //        deve essere definito nella classe ITEM.
    if (sp != NULL) {
        sp->tail.print();
        // qui è applicato ricorsivamente questo metodo ...
        sp->head.print();
        // ... ma qui è applicato il metodo della classe ITEM!
        // La pila è ancora stampata dal primo all'ultimo ITEM inserito.
    }
}

```

```

int height () const {
    // calcola (e restituisce) il numero di ITEM "impilati"
    if (sp == NULL) return 0;
    return 1 + sp->tail.height();
}

bool operator == (const STACK<ITEM> & s) const {
    // overloading dell'operatore di uguaglianza ("deep")
    if (sp == NULL && s.sp == NULL) return true;
    if (sp == NULL || s.sp == NULL) return false;
    return sp->head == s.sp->head && sp->tail == s.sp->tail;
        // qui il primo == è tra ITEM, ma il secondo è tra STACK
        // (è quello che si sta definendo adesso, che ricorre)
}

STACK<ITEM> & operator = (const STACK<ITEM> & s) {
    // overloading dell'operatore di assegnazione, definito
    // in modo che sia possibile comporre più assegnazioni
    STACK<ITEM> aux;          // crea la pila vuota aux ...
    aux_assign(aux, s);       // ... e vi costruisce una copia di s
    make_empty();             // rende vuota questa pila
    sp = aux.sp;              // assegnazione tra puntatori
    aux.sp = NULL;            // per "ingannare" il distruttore!
    return *this;
}

STACK<ITEM> (const STACK<ITEM> & s) {
    // costruttore di copia "deep" (versione che usa l'assegnazione)
    sp = NULL;
    *this = s;   // ora si può assegnare s a questa pila (vuota) !
}

~STACK<ITEM> () {
    // distruttore: recupera la memoria occupata dai frame eliminati
    make_empty();
}

void make_empty () {
    // rende vuota la pila (con recupero della memoria ...)
    if (sp != NULL) {
        sp->tail.make_empty();
        delete sp;
        sp = NULL;
    }
}
};

};
```

Per esercizio, dare una definizione *iterativa* di `is_in`, `make_empty`, `height` e dei metodi di stampa.

Inoltre, definire una classe di eccezioni che possa essere usata dalla funzione `pop`. A parte un opportuno cambiamento di nomi, per realizzare un tipo di dato “coda” (`QUEUE<ITEM>`) basterà sostituire la definizione dell’operazione `push` ...

**Parole chiave:** albero di gioco, *puzzle* con possibili cicli, pila e coda, classe parametrica in C++, strutture circolari, algoritmo di esaurimento, ricorsione, *backtracking*, complessità esponenziale, ricerca euristica.



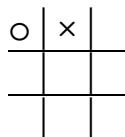
## 9. Una partita a tris

Nella gara *Kangourou dell'Informatica* di marzo 2010 fu proposto ai concorrenti della categoria “Biennio” un quesito su un famoso giochino, divenuto paradigmatico nella nostra disciplina per spiegare come si trattano, più in generale, i giochi tra due avversari che abbiano certe caratteristiche: di questi ci occuperemo qui e nei prossimi capitoli, illustrando come possano essere “risolti”, almeno in linea di principio, e quali schemi si possono applicare per realizzare un programma al computer che sia in grado di giocare in modo decente – o addirittura infallibile – una partita contro un avversario, umano o software che sia. Naturalmente, le parti più tecniche a quest’ultimo riguardo potranno essere saltate dal lettore non specialista, senza che la comprensione del discorso ne risulti inficiata.

Per cominciare, riproponiamo quel quesito in una forma leggermente diversa, con una risposta (sbagliata) in più; poi passeremo agli approfondimenti…

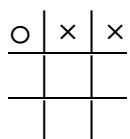
Sir Cross e Lady Circle stanno giocando una partita al gioco del tris (o “filetto”), che certamente tutti conoscete: vince chi per primo fa tris, in orizzontale o verticale o diagonale, e se nessuno fa tris la partita finisce in parità.

Ha iniziato Sir Cross, mettendo una croce in mezzo a uno dei lati; Lady Circle ha risposto disegnando un cerchio in uno degli angoli a fianco della croce, come mostrato nella figura qui sotto.

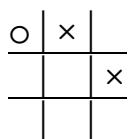


Quale tra le seguenti mosse deve fare adesso Sir Cross per assicurarsi almeno il pareggio?

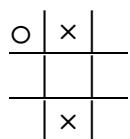
A



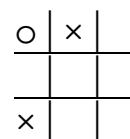
B



C

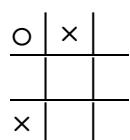


D



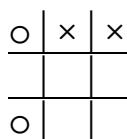
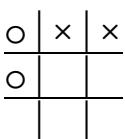
E – Nessuna delle precedenti: Sir Cross ha ormai perduto la partita!

**Soluzione.** Tra quelle proposte, l'unica mossa giusta per Sir Cross è la D, cioè questa:

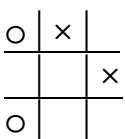


Infatti, a questo punto, Lady Circle non ha alcuna mossa vincente, e pertanto Sir Cross si assicura almeno il pareggio – se giocherà bene, s'intende!

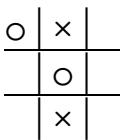
Gli altri tre casi vanno esclusi perché vi è sempre almeno una mossa che consentirebbe a Lady Circle di vincere. Nel caso A, la mossa vincente di Lady Circle è indifferentemente una di queste due:



Nel caso B, Lady Circle ha un'unica mossa vincente:

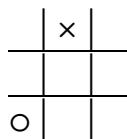
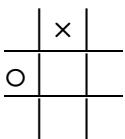


e così pure nel caso C, in cui l'unica possibilità di vittoria per Lady Circle è:



Per inciso, oltre alle quattro mostrate, Sir Cross aveva altre tre possibili mosse (in questa situazione non v'è alcuna simmetria!) e tutte e tre gli avrebbero assicurato (almeno) il pareggio.

Senza alcun dubbio il filetto è universalmente conosciuto e ancora praticato da tanti ragazzi, sebbene sia altrettanto noto che si tratta di un gioco “alla pari”: nessuno dei due giocatori riesce a vincere se l'avversario non commette errori! Occorre soltanto fare un po' di attenzione proprio nella fase iniziale. Ad esempio, se alla sua prima mossa Lady Circle avesse risposto in uno di questi due modi:



ovvero in uno dei due simmetrici, allora Sir Cross avrebbe facilmente vinto...

### **Le prime nozioni: stati e albero di gioco.**

Il gioco del tris – che forse risale all'Antico Egitto – è stato oggetto non soltanto di analisi e pubblicazioni, ma anche del primo *video game* a computer, realizzato nel 1952 per l'EDSAC (*Electronic Delay Storage Automatic Calculator*) presso la Università di Cambridge; l'utente sceglieva la casella ruotando un disco telefonico, il tavoliere completo era mostrato sul tubo a raggi catodici di un oscilloscopio.

In verità, già un secolo avanti, Charles Babbage – menzionato in più occasioni nel nostro *excursus*, ma principalmente come colui che per primo concepì l’idea di calcolatore *programmabile universale* – aveva progettato sulla carta un automa capace di giocare a “*tit-tat-to*”: questo il nome da lui usato per indicare il gioco; tra gli altri nomi tuttora diffusi: *tic-tac-toe* o *ticktacktoe* o, con riferimento ai simboli grafici, *noughts and crosses*, che il grande poeta romantico William Wordsworth (1770-1850) chiama *crosses* e *cyphers* in un suo preludio sui ricordi di scuola...

Contando come una sola tutte le posizioni che si possono ottenere l’una dall’altra per qualche simmetria (rotazioni e/o ribaltamenti), quelle possibili sono 765 e le partite 26830: ben poca cosa per un computer, decisamente più arduo compito una elaborazione manuale.

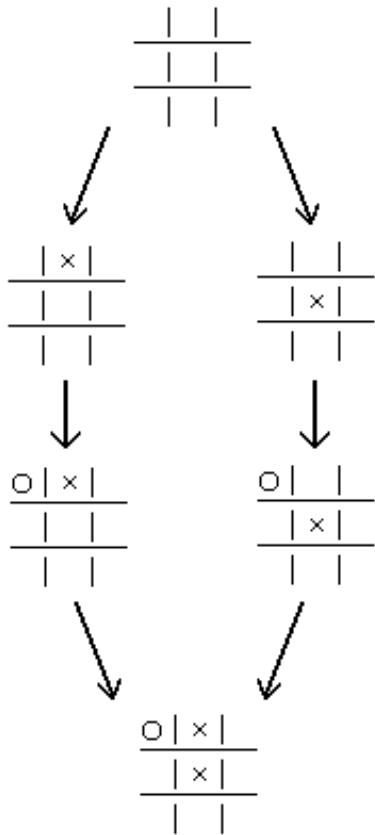
Come si dovrebbe procedere? Per costruire l’*albero di gioco* completo partiamo dall’alto, mettendo in *radice* lo *stato iniziale* (lo schema vuoto, ove inizierà chi mette la croce), e poi procediamo verso il basso, facendo in modo che ciascun nodo abbia come *figli* tutti gli stati che si possono originare con una mossa del giocatore al quale tocca muovere nello stato rappresentato da quel nodo. Ad esempio, i figli della radice saranno tre, poiché la croce può essere messa in sole tre posizioni essenzialmente differenti (al centro, o in uno degli angoli, o in mezzo a uno dei lati). Gli stati *finali* (in cui uno dei due giocatori ha fatto tris o – inclusivo! – tutte e nove le mosse sono state fatte, e la partita è comunque terminata) saranno le *foglie* dell’albero. E proprio dalle foglie dovremo partire, via via risalendo, per analizzare completamente (o, come si dice, *risolvere*) il gioco, come tra poco vedremo.

Prima, però, una precisazione è doverosa. Abbiamo parlato di stati o posizioni, ossia in generale configurazioni del tavoliere di gioco che si possono presentare durante una partita (in cui va compresa l’informazione binaria che dice “a chi tocca muovere”, se non è già implicita). Se pensiamo a tutti questi stati collegati da archi, così che vi sia un arco dallo stato  $x$  allo stato  $y$  se e soltanto se con una mossa lecita si può passare da  $x$  a  $y$ , allora in realtà non stiamo pensando a un albero ma più in generale a un *grafo* (orientato). Se questo grafo avesse dei cicli, allora il gioco potrebbe continuare all’infinito... Ciò non accade nel filetto, il cui grafo è *aciclico*: ogni mossa aggiunge un nuovo segno, e la partita termina sempre.

A titolo d’esempio, nella figura alla pagina successiva è riportato un *sottografo* che mostra come a uno stato si possa arrivare con sequenze di mosse diverse.

Tuttavia – in accordo all’uso solito nella letteratura – è più semplice e conveniente pensare a un albero, in cui più nodi distinti possono rappresentare la stessa posizione, se a questa si può arrivare mediante altrettante diverse sequenze di mosse a partire dallo stato iniziale.

Abbiamo affermato che, nel gioco del tris, gli stati raggiungibili (compreso quello iniziale) sono 765: essi costituiscono i nodi del *grafo* di gioco – quelli dell’albero sono ben di più! Se ci riferiamo allora al numero di mosse per giungervi a partire dallo stato iniziale, tali stati sono distribuiti come di seguito è riportato.



Dopo 0 mosse: 1 stato (quello iniziale, senza alcun segno);

dopo 1 mossa: 3 stati;

dopo 2 mosse: 12 stati;

dopo 3 mosse: 38 stati;

dopo 4 mosse: 108 stati;

dopo 5 mosse: 174 stati (21 dei quali sono finali, a favore del primo giocatore, quello che mette la croce);

dopo 6 mosse: 204 stati (21 dei quali sono finali, a favore del secondo giocatore, che mette il cerchio);

dopo 7 mosse: 153 stati (58 dei quali sono finali, a favore del primo giocatore);

dopo 8 mosse: 57 stati (23 dei quali sono finali, a favore del secondo giocatore);

dopo 9 mosse: 15 stati (tutti ovviamente finali: 12 a favore del primo giocatore e 3 in cui nessuno fa tris).

In effetti, soltanto tre sono i possibili stati finali di una partita patta:

○	×	○
×	×	○
×	○	×

○	×	○
×	○	×
×	○	×

○	×	×
×	○	○
×	○	×

Diciamo ancora che dopo 9 mosse, col tavoliere riempito di segni (5 croci e 4 cerchi), in 6 casi sui 12 favorevoli al primo giocatore, questi realizza ben *due* tris apponendo l'ultima croce; ad esempio, ciò accade quando lo stato precedente è:

o	x		o
o	x		o
x			x

Questo non deve sorprendere: tra tutte, infatti, ci sono anche quelle partite in cui un giocatore commette qualche errore – e magari l'avversario non ne approfitta, o quantomeno non subito! Naturalmente, invece, tra gli stati raggiungibili non ve ne può essere alcuno con un tris di croci *e* un tris di cerchi.

Nello schema che segue sono riportate le mosse tra le quali il secondo giocatore deve scegliere la sua prima mossa, a seconda di quella fatta dal primo giocatore, per non lasciare subito a quest'ultimo la certezza di vincere:

o		o
	x	
o		o

		x
	o	

o	x	o
	o	
	o	

## L'analisi, in generale.

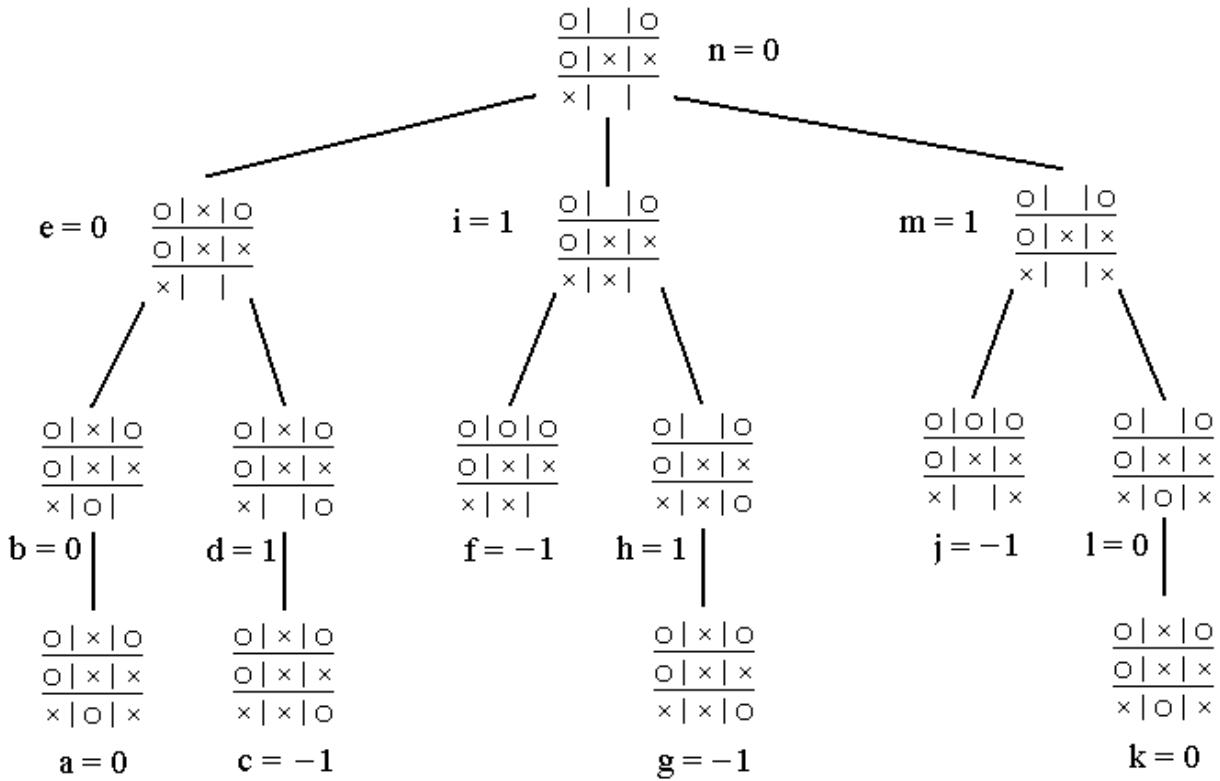
L'analisi del gioco si basa su questo ragionamento: se in un certo stato tocca a me muovere, cercherò la mossa che mi farà guadagnare il massimo possibile (ovvero perdere il minimo possibile); se invece tocca al mio avversario, cercherò quella che farà guadagnare a lui il massimo possibile: suppongo infatti che pure lui giochi, come me, al meglio delle proprie possibilità.

Parto dunque dalle foglie dell'albero, attribuisco a ciascuna di esse un *punteggio di tipo ternario*:  $-1$  se perdo,  $0$  se la partita è patta,  $1$  se vinco. Poi risalgo via via fino alla radice, attribuendo a ciascun nodo (di cui ho già valutato tutti i figli) il *massimo* punteggio dei figli se in quel nodo tocca a me muovere, il *minimo* se tocca al mio avversario. Quando arrivo a valutare la radice, il gioco è risolto: se il suo punteggio è  $1$  io ho una strategia vincente, se è  $-1$  ce l'ha il mio avversario, se è  $0$  la partita non potrà che finire in parità, qualora nessuno di noi due commetta errori.

Una versione del tutto equivalente, di più semplice codifica in un linguaggio di programmazione come C++ o Java, nota come *negamax*, consiste nel valutare ciascuna foglia  $0$  se la partita è patta,  $-1$  se *chi dovrebbe muovere* perde; risalendo l'albero, basterà sempre attribuire a ciascun nodo (di cui sono già stati valutati tutti i figli) il *minimo* punteggio dei figli *cambiato di segno*. In questo caso, il punteggio attribuito a ciascun nodo (dunque pure alla radice) è riferito al giocatore di turno nel nodo. Teniamo infatti presente che

$$\max(x_1, x_2, \dots, x_n) = -\min(-x_1, -x_2, \dots, -x_n).$$

Per esemplificare questo procedimento, valutiamo un particolare stato del gioco nel quale è possibile giungere durante una partita (si veda l'albero illustrato alla pagina successiva).



Nel nodo radice di questo *sottoalbero* (dell'albero di gioco completo) abbiamo messo lo stato da valutare, in cui deve muovere chi mette la croce.

Per attuare la valutazione *negamax* delineata sopra, possiamo procedere con una visita in *post-order* (prima di valutare un nodo si valutano tutti i suoi figli, ricorsivamente): quindi i nodi saranno valutati nell'ordine indicato dalle lettere, con i punteggi accanto riportati... In ultimo, la radice: il minimo punteggio dei figli ("e", "i", "m") è 0, che cambiato di segno è ancora 0, e dunque l'unica mossa giusta che dovrà fare il giocatore di turno (che mette la croce) per non perdere è quella che porta al nodo "e".

A questo punto, i più attenti tra i lettori potrebbero aver intuito che la valutazione dei nodi "g", "h", "k", "l" è inutile... ma questo aspetto sarà approfondito nel prossimo capitolo, dedicato al gioco dell'Aware!

Piuttosto un'altra cosa notiamo, a conferma di quanto già esposto: nell'albero di gioco diversi nodi possono contenere (o, come si dice in termini tecnici, essere etichettati con) lo stesso stato; in figura, i quattro nodi sull'ultimo *livello* contengono soltanto due stati differenti (quello di "a" e "k", che è uno degli stati di parità, e quello di "c" e "g", che è uno degli stati in cui vince chi ha appena messo la croce, e quindi chi dovrebbe muovere perde).

Se fossimo partiti dallo stato iniziale del gioco e avessimo visitato l'intero albero, saremmo arrivati alla stessa valutazione del nodo radice: 0, cioè il gioco è alla pari.

Ovviamente, in ogni specifica circostanza, a meno che non si parta da una foglia, bisognerà ricordare almeno una delle mosse che portano a uno dei figli della radice che hanno determinato il punteggio ad essa attribuito, vale a dire una delle mosse *migliori* (nell'esempio visto sopra ce n'è una sola: quella che porta al nodo "e").

## Idee per la realizzazione di un programma.

Diamo ora qualche consiglio a chi intende scrivere un programma che giochi (in modo infallibile) a filetto.

Continuando a usare un linguaggio a oggetti, come il C++, definiremo una *classe* board, un’istanza della quale sia in grado di rappresentare uno stato (legale) del gioco.

Avremo bisogno di un *campo* per il tavoliere: per semplicità, possiamo pensare a un *array* di 9 interi (anche con un solo indice), in cui 0 significhi “casella vuota”, 1 il segno “croce” messo dal giocatore che inizia, 2 il segno “cerchio” messo dal secondo giocatore. Inoltre, per comodità, possiamo prevedere altri due campi: uno, *booleano*, per tener pronta l’informazione binaria che dice “a chi spetta muovere” e l’altro, intero, che tenga conto del numero di mosse finora fatte, evitando così di ricavare queste informazioni contando croci e cerchi già presenti sul tavoliere.

La più semplice versione del *metodo* di analisi, ricorsiva all’interno di un ciclo **for**, rispecchia il procedimento che abbiamo appena illustrato (magari, per maggiore semplicità, senza tener conto delle simmetrie!) e – *a prescindere dal gioco specifico* – può essere così codificata:

```
int board :: minimax (int & move) const { // in versione negamax
    board C;
    int i, value, new_value, new_move;
    if (is_a_leaf()) { move = 0; return payoff(); }
    value = -2;
    // basta che value sia < -1, affinché la prima mossa lecita
    // venga considerata la migliore finora trovata!
    for (i = 1; i <= MAX_MOVES; i++)
        // a una a una, il ciclo considera tutte le mosse possibili
        if (is_legal(i)) {
            // copia il board corrente nel board C
            // e poi esegue su C la mossa numero i:
            C = *this; // assegnazione tra board (overloaded)
            C.make_move(i);
            // ricorre a partire dal nuovo stato di gioco:
            new_value = - C.minimax(new_move);
            // ... ma qui new_move è dimenticata!
            // se è stata trovata una mossa migliore rispetto
            // alla migliore finora trovata, ne tiene memoria:
            if (new_value > value) { value = new_value; move = i; }
        }
    return value;
}
```

Per il gioco del tris, un’applicazione di questo metodo a un *board* che rappresenta il tavoliere iniziale (vuoto) restituirà valore esplicito 0 e, quale effetto collaterale, nella variabile intera trasmessa come argomento (per riferimento) memorizzerà il numero 1, poiché tutte le mosse iniziali assicurano ugualmente il pareggio, e quindi dopo la prima non ne troverà di migliori.

Il *board* al quale il metodo `minimax` è applicato non viene comunque modificato: tale metodo è infatti dichiarato **const**, e nel medesimo modo dovranno comportarsi anche i metodi `is_a_leaf()`, `payoff()` e `is_legal(int)`, implicitamente applicati al *board* stesso.

Il metodo a valori *booleani* `is_a_leaf()` restituisce **true** se e soltanto se nello stato rappresentato non ci sono mosse lecite da poter tentare: la partita è finita. Nel nostro caso, basta controllare se sul tavoliere c'è un tris oppure se sono state fatte 9 mosse (cioè tutte le caselle sono state segnate).

Il metodo a valori interi `payoff()` restituisce il punteggio relativo al giocatore che *dovrebbe* muovere in quello stato, e pertanto qui le possibilità sono soltanto due: -1 se perde (cioè se il giocatore che ha appena mosso ha fatto tris), 0 altrimenti (cioè se nessuno ha fatto tris e quindi la partita finisce in parità).

Il metodo a valori *booleani* `is_legal(int i)` restituisce **true** se e soltanto se la mossa con numero d'ordine *i* è lecita: ciò significa che può essere eseguita in quello stato. Nel nostro caso, banalmente, basta controllare se la casella *i* ha valore 0 (cioè non è stata ancora segnata).

Il metodo `make_move(int i)` esegue la mossa con numero d'ordine *i*, e pertanto modifica di conseguenza lo stato dell'oggetto al quale è applicato (che è una *copia deep* di quello a cui è stato applicato correntemente il metodo `minimax`). Nel nostro caso, basta assegnare il numero del giocatore che deve muovere alla casella *i*, cambiare il numero del giocatore a cui spetta muovere (al turno successivo toccherà infatti all'avversario) e incrementare di un'unità il numero di mosse eseguite.

La costante `MAX_MOVES` è uguale al massimo numero di mosse tra le quali può essere scelta quella da effettuare; nel nostro caso 9, se rinunciamo a sfruttare le simmetrie (ciò evita comunque complicazioni nella stesura del programma, e qui è fattibile poiché non compromette sensibilmente l'efficienza). Ovviamente, il codice scritto sopra presuppone una qualche numerazione delle mosse, stabilita in precedenza: nel nostro caso ciò è estremamente semplice – come abbiamo già capito – poiché basta fissare una corrispondenza biunivoca tra i numeri da 1 a 9 e le caselle del tavoliere, ad esempio numerandole per righe oppure per colonne, partendo dall'alto o dal basso, da sinistra o da destra...

Le mosse lecite eseguibili nello stato corrente sono generate via via nel *board* ausiliario *C* e *ricorsivamente* valutate: ogni volta, lo stato corrente è copiato in *C*, su quest'ultimo sono riportati gli effetti prodotti dalla mossa considerata e subito dopo è invocato lo *stesso* metodo `minimax`. Ciascuna applicazione del metodo `minimax` restituisce come risultato esplicito il valore (-1, 0 o 1) che rappresenta *il più grande guadagno minimo garantito* al giocatore al quale spetta muovere in *C*, e ha l'effetto collaterale di assegnare alla variabile intera trasmessa come argomento (per riferimento) il numero d'ordine di una mossa che assicura questo guadagno (zero significa nessuna mossa). Infine, si controlla se tale risultato, cambiato di segno, migliora la valutazione finora fatta dello stato corrente: in caso affermativo (e di certo la prima volta), la valutazione è aggiornata e la mossa considerata è ricordata.

## Rendiamo migliore il metodo!

E così il metodo `minimax` che abbiamo scritto ricorda (al più) *una* mossa da fare: sarebbe meglio ricordare *tutte* le mosse che assicurano il punteggio calcolato e, quando sono almeno due, avere poi la possibilità di scegliere (casualmente) quella da fare. Allora il parametro `move` non dovrà essere semplicemente una variabile di tipo intero, bensì un oggetto adatto a rappresentare un *insieme* di interi, che possiamo supporre istanza di una classe (parametrica) `SET<int>`.

Sebbene non abbia la stessa rilevanza, introduciamo anche un'altra modifica: non ci serviremo più di un *board* ausiliario, ma eseguiremo ciascuna mossa lecita sul *board* corrente (quindi il metodo non sarà più `const`) e, dopo la sua valutazione, la ritireremo; nel nostro caso questo compito risulterà assai facile.

```
int board :: minimax (SET<int> & moves) {
    int i, value, new_value;
    SET<int> new_moves; // new_moves = insieme vuoto
    moves.make_empty(); // moves = insieme vuoto
    if (is_a_leaf()) return payoff();
    value = -2;
    for (i = 1; i <= MAX_MOVES; i++)
        if (is_legal(i)) {
            // esegue la mossa numero i sul board corrente:
            make_move(i);
            // ricorre a partire dal nuovo stato di gioco:
            new_value = - minimax(new_moves);
            // ... ma, di nuovo, new_moves non sarà usato!
            // se è stata trovata una mossa migliore o equivalente
            // rispetto alle migliori finora trovate, ne tiene memoria:
            if (new_value > value) { // mossa migliore
                value = new_value;
                moves.make_empty(); // dimentica le (eventuali) precedenti
                moves.insert(i); // e ricorda (per ora) soltanto questa
            } else if (new_value == value) { // mossa equivalente
                moves.insert(i); // è aggiunta alle (eventuali) precedenti
            }
            // ripristina lo stato precedente alla mossa numero i:
            retract_move(i);
        }
    return value;
}
```

Assumiamo che il costruttore senza parametri della classe `SET<int>` assegni all'oggetto creato il valore iniziale “insieme (di interi) vuoto”. Tale classe dovrà avere i due metodi (pubblici): `make_empty()`, che rende vuoto l'insieme (già allocato in memoria), e `insert(int i)`, che aggiunge all'insieme l'intero `i`. Bisognerà poi disporre di metodi opportuni per controllare se l'insieme è vuoto e, ammesso che sia non vuoto, per estrarne gli elementi (o magari per stamparli);

ad esempio, potrà essere utile un metodo a valori interi che restituisca (senza rimuoverlo) un elemento scelto a caso tra quelli dell'insieme (purché non vuoto).

Il metodo `retract_move(int i)` (privato!) della classe `board` deve semplicemente ritirare la mossa con numero d'ordine `i`. Nel nostro caso, basta assegnare valore 0 alla casella `i`, cambiare nuovamente il numero del giocatore a cui spetta muovere e decrementare di un'unità il numero di mosse eseguite.

Riassumendo, e prescindendo dal gioco in esame, questa versione di `minimax` può essere impiegata per valutare *tutto* il (sotto)albero di gioco che ha come radice lo stato rappresentato dal `board` al quale il metodo è applicato, e per ricordare tutte le mosse (equivalenti) più convenienti per il giocatore di turno in tale stato:

- se restituisce 1, significa che il giocatore di turno ha una strategia vincente, e dovrà fare una delle mosse dell'insieme `moves`;
- se restituisce 0, significa che, se entrambi i giocatori agiranno in modo perfetto, senza commettere errori, da questa situazione di gioco la partita finirà in parità, e di nuovo il giocatore a cui tocca muovere dovrà fare una delle mosse dell'insieme `moves`;
- se restituisce -1, significa che questa volta è l'avversario del giocatore di turno ad avere una strategia vincente, e allora nell'insieme `moves` ci saranno necessariamente tutte le mosse (ugualmente perdenti!) che può fare il giocatore a cui tocca muovere.

Se l'insieme di mosse rappresentato da `moves` (che è un parametro di *puro output*) è vuoto, significa che il giocatore di turno non ha mosse da fare perché la partita è terminata, ed egli vince, pareggia o perde a seconda del valore esplicitamente restituito dal metodo (1, 0 o -1, rispettivamente). Tuttavia, nel filetto, la prima di queste tre eventualità non può accadere poiché, come abbiamo visto, alle foglie dell'albero sono assegnati soltanto i valori 0 o -1; in altri giochi, però, potrebbe succedere che il giocatore che ha fatto l'ultima mossa perda la partita, oppure che non si possa avere un pareggio...

Dovrebbe a questo punto apparire compito non troppo impegnativo la stesura di un programma che, a scelta dell'utente, sia in grado di giocare una partita:

- contro l'utente, al quale spetti decidere chi muove per primo, oppure
- contro sé stesso,

in ogni caso mostrando sullo schermo la situazione del gioco dopo ciascuna mossa, fino a quando sarà premuto un tasto o comunque per un tempo adeguato. Ogni volta che saranno registrate più mosse equivalenti (ugualmente convenienti), la scelta sarà fatta in maniera casuale. Quando non ci sono mosse possibili, il gioco è finito e deve essere stampato l'esito della partita. Qualora, a un certo punto della partita, il risultato di `minimax` sia stato 1, per onestà dovrebbe essere stampato un messaggio di avviso per l'avversario, che andrà a perdere! Potrà infatti succedere che, dopo un errore dell'utente (umano), il programma sia sicuro di vincere.

## Giochi strettamente determinati.

Si intuisce un’ipotesi, qui verificata ma non ancora esplicitata nel nostro discorso: che il gioco sia “*a somma zero*”, vale a dire che uno svantaggio di uno dei due giocatori equivalga a un vantaggio di pari entità dell’altro giocatore (e se i giocatori sono più di due, come nel poker, la somma delle vincite uguaglia quella delle perdite).<sup>26</sup>

Due sono le altre caratteristiche salienti dei giochi di questa categoria, che a noi qui interessa (e nella quale rientrano anche l’Awari, la dama e gli scacchi):

- il gioco è *finito*: ossia, l’albero di gioco è finito, sia in ampiezza (in ogni stato il numero di mosse lecite è limitato), sia in altezza (prima o poi la partita termina);
- il gioco è *a informazione perfetta*: ossia, in ogni momento, in particolare prima della scelta della propria mossa, ciascun giocatore conosce lo stato completo del gioco, oltre alle regole e alle funzioni di *payoff*, e nulla è tenuto nascosto o lasciato al caso.

Per inciso, in queste due ipotesi, il gioco ammette un *equilibrio*, unico se il *payoff* è lo stesso. Siccome poi i giocatori sono due, i quali muovono a turno, *alternandosi*, e il gioco è *a somma zero*, si tratta di un gioco *strettamente determinato*: o uno dei due giocatori ha almeno una strategia *pura* (non probabilistica, ma precisa) che gli assicuri la vittoria (cioè: ad ogni suo turno ha almeno una mossa che può portarlo infine a vincere, qualunque sia la strategia adottata dal suo avversario), o entrambi hanno almeno una strategia pura che assicuri loro il pareggio.<sup>27</sup>

In effetti, il procedimento di *induzione a ritroso* che abbiamo delineato nei paragrafi precedenti permette di decidere quale si verifichi dei tre casi possibili già previsti nel 1912 dal matematico tedesco Ernst Zermelo (1871-1953), fondatore della teoria assiomatica degli

---

<sup>26</sup> I giochi a somma zero furono i primi ad essere studiati, in quanto più facili da analizzare; sono detti anche *puramente competitivi*, o in totale conflitto di interessi, poiché gli interessi dei giocatori sono completamente contrapposti. La moderna *teoria dei giochi* si occupa altresì (e soprattutto) di giochi *a interessi misti* (con parziale coincidenza di interessi) e *di coordinamento* (o di *collaborazione pura*, con totale coincidenza di interessi), casi assai più frequenti in ambito economico o sociale.

Anticipiamo un’osservazione: se l’esplorazione dell’albero di gioco è spinta soltanto fino a una certa profondità, a partire dallo stato attuale, come assumeremo nel prossimo capitolo, allora l’ipotesi “a somma zero” si verifica *a ciascuna singola mossa*: il giocatore di turno cercherà sì di *minimizzare il massimo payoff* (guadagno) possibile del suo avversario (da cui il nome “*minimax*”), ma alla successiva mossa di quest’ultimo lo scenario cambierà, a parità di profondità, poiché l’orizzonte si sposta “in avanti”…

<sup>27</sup> Tra il 1926 e il 1928, John von Neumann provò che nei giochi a somma zero, a due giocatori, ognuno dei quali può scegliere fra un numero finito di mosse, esiste un *equilibrio* (caso particolare di quello che sarà poi noto come “*equilibrio di Nash*”) in cui ciascuno dei due minimizza il massimo *payoff* dell’avversario; in virtù dell’ipotesi “a somma zero”, minimizza pure la propria massima perdita. Se il gioco non fosse a somma zero, allora, allo scopo di evitare il peggio, egli dovrebbe invece applicare il metodo *maximin*: rendere massimo il proprio *payoff* minimo garantito, che è una scelta estremamente pessimistica; ma nell’ipotesi di gioco “a somma zero” le due regole di decisione, *minimax* e *maximin*, sono equivalenti. Se l’informazione è *incompleta* (di solito, quando si ignora qualcosa dell’altrui situazione), ciascuno può trovare una strategia *mista* che costituisce un “*equilibrio*”: ossia il miglior compromesso per entrambi. Insieme con l’economista austriaco Oskar Morgenstern, von Neumann estese poi il risultato al caso di più giocatori (*Theory of Games and Economic Behavior*, 1944).

Tra il 1950 e il 1951, John F. Nash (1928-2015) provò che ogni gioco (finito), anche a somma *variabile*, ha almeno un equilibrio, in strategie pure e/o miste; ciò si verifica, ad esempio, anche nella morra cinese, purché i due giocatori muovano sì simultaneamente (quindi con informazione incompleta), ma scegliendo a caso tra le mosse lecite con pari probabilità. Dunque, se esiste un unico equilibrio, esso è la soluzione del gioco (ma non è detto che sia una “buona” soluzione: si veda il famoso “dilemma del prigioniero”). In un equilibrio di Nash, la strategia di ciascun giocatore è la risposta ottimale a quella degli altri: anche se ciascun giocatore potesse cambiare *unilateralmente* la propria scelta, dopo aver saputo quelle degli altri, nessuno ne trarrebbe vantaggio. Questi notevoli contributi alla teoria dei giochi valsero a Nash il premio Nobel per l’economia nel 1994.

insiemi: salvo errori, è *sempre* in grado di vincere chi inizia, oppure chi risponde, oppure è patta teorica. Vale a dire: due giocatori *razionali* sanno ancor prima di giocare qual è l'esito di *ogni* partita, sempre lo stesso!

Nel filetto, e come vedremo nell'Awari, entrambi i giocatori hanno almeno una strategia pura che assicura loro il pareggio. Allora potremmo ritenere che sia facile risolvere un gioco: basta idearne un'adeguata rappresentazione o modello informatico, e poi applicare il metodo *minimax* allo stato iniziale.

C'è un "però", come facilmente si intuisce da quanto accennato: l'albero di gioco può risultare gigantesco, tanto da non essere immaginabile! Si pensi agli scacchi o allo Shōgi (gli scacchi giapponesi) o addirittura al grande Go giocato sul quadrato  $19 \times 19$  (si veda l'illustrazione in copertina), nei quali il numero di posizioni diverse è stimato di 47 o 71 o 171 cifre decimali, rispettivamente: comunque, enormemente più grande sia di 765, quanti gli stati del filetto, sia di 900 miliardi, quanti all'incirca quelli (sempre essenzialmente differenti) dell'Awari – figuriamoci il numero di partite che possono essere giocate! Si tratta di numeri che sfuggono all'umana comprensione: il numero di atomi in tutto l'universo conosciuto sembra avere appena un'ottantina di cifre decimali...

Come può agire allora un programma che abbia la pretesa di combattere contro un avversario? Anziché arrivare alle foglie, esplora l'albero dalla posizione attuale fino a una certa profondità, cercando di attribuire in modo efficace (e questo è un aspetto critico) un punteggio agli stati raggiunti, senza proseguire oltre (salvo che in casi particolari). In più, esso tenta di "potare" l'albero, evitando di scendere lungo rami che non possono migliorare la valutazione finora fatta della posizione attuale.

Rimane un problema: come assegnare un punteggio a uno stato che non sia finale, rinunciando ad indagare oltre? In taluni casi ciò può essere relativamente semplice, quando si riesca a quantificare bene il vantaggio di un giocatore in base al materiale o alla posizione; si tratta, in genere, di una valutazione ispirata da criteri euristici, strettamente collegati al gioco specifico e magari suggeriti da un esperto. In altri casi, invece, il problema può apparire insormontabile: si pensi proprio al filetto... ma si tenga comunque presente che, in mancanza di idee migliori, è sempre possibile seguire la falsariga dell'algoritmo *negamax* introdotto a pagina 249, attribuendo valore  $-1$  a ciascuno degli stati in cui il giocatore a cui spetterebbe muovere perde, e valore  $0$  a ciascuno degli altri stati nei quali l'analisi si ferma).<sup>28</sup>

Volendo essere più raffinati, e far le cose al meglio, al posto dell'intero  $-1$  si può considerare il razionale  $-1/n$ , dove  $n$  è il numero di mosse che ha portato dallo stato a cui va assegnato il punteggio allo stato di sconfitta del giocatore a cui spetterebbe muovere: così facendo, il programma sarà in grado sia di anticipare la propria eventuale vittoria certa, sia di ritardare la propria eventuale sconfitta "certa" (purché l'avversario giochi bene), nella speranza che l'utente (umano) commetta un errore...

Approfondiremo alcuni aspetti della questione nel prossimo capitolo, dove proprio a questo scopo ci serviremo dell'Awari, un altro gioco – decisamente più complesso del filetto – che ha suscitato l'interesse sia dei matematici sia degli informatici.

---

<sup>28</sup> In generale, l'"erroneità" di una mossa dipende dalla profondità di analisi: infatti, per stabilire con certezza che una mossa è "sbagliata" in modo irrimediabile (ammesso che l'avversario giochi bene), occorre spingere l'analisi del gioco sino alle estreme conseguenze, appurando che, una volta fatta tale mossa, l'avversario abbia una strategia vincente, o comunque migliore dell'attuale.

Un’ultima curiosità: la dama giocata sulla tradizionale scacchiera  $8 \times 8$  è stata risolta (in senso debole) a favore della parità, nella primavera del 2007, dopo 18 anni di calcoli da parte di una cinquantina di computer. Si tratta del più grande gioco per cui sia stato trovato un risultato di questo tipo, sebbene il numero di posizioni abbia “soltanto” 21 cifre decimali... e quindi sia parecchi milioni di miliardi di miliardi di volte più piccolo di quello stimato per gli scacchi! Per individuare una strategia non perdente, a partire dalla posizione iniziale, è bastata l’analisi di circa quarantamila miliardi di posizioni. Tuttavia – e per fortuna – ciò non toglie proprio nulla all’emozione di una partita a dama, almeno tra umani!

Nel prossimo capitolo diremo qualcosa sui programmi “tradizionali” che giocano a scacchi, e vedremo in particolare gli algoritmi di ricerca in profondità con “potature”. In essi hanno un ruolo importante le funzioni di valutazione delle posizioni, studiate “a mano” e rifinite da esperti nel corso di parecchi decenni. In tempi recenti, tuttavia, ha fatto notizia la realizzazione di giocatori artificiali in un paradigma radicalmente diverso da quello dei motori tradizionali: impiegano infatti reti neurali, con autoapprendimento “profondo”, e partendo soltanto dalle semplici regole del gioco, senza alcun’altra conoscenza preliminare, si sono mostrati di gran lunga superiori a tutti gli altri giocatori, non solo umani. Le ultime novità sono rappresentate da AlphaZero, che gioca sia a scacchi, sia a Shōgi, sia a Go, costruito da DeepMind (David Silver e altri), e dall’*open-source* Leela Chess Zero (Gary Linscott e altri): ad essi, e ai loro predecessori, ho accennato nella prefazione a questa seconda edizione.

### **Un gioco isomorfo al filetto.**

Un gioco in apparenza diverso dal filetto è il seguente: sul tavolo, senza seguire alcun ordine particolare, si dispongono nove carte, con valori da 1 a 9; il giocatore di turno prende una carta; vince chi riesce a totalizzare 15 sommando i valori di tre delle carte da lui prelevate.

In realtà si tratta di un gioco *isomorfo* al filetto: vi è una corrispondenza biunivoca tra le possibili partite a questo gioco e quelle a filetto! Per rendersene conto, basta disporre le nove carte secondo il *quadrato magico* di ordine 3, come mostrato a fianco.

4	9	2
3	5	7
8	1	6

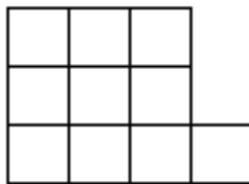
A parte le solite simmetrie (rotazioni e/o ribaltamenti), questo è l’unico quadrato magico di ordine 3: la somma dei numeri su ciascuna riga, su ciascuna colonna e su entrambe le diagonali maggiori è 15. Nell’antica Cina, dove fu ideato e chiamato *Lo-Shu*, questo quadrato era considerato un talismano, tanto che lo si trova inciso su amuleti portati ancor oggi nell’Estremo Oriente e in India.

L’aspetto che qui interessa è che esso rappresenta tutti i possibili modi per ottenere 15 sommando tre numeri diversi compresi tra 1 e 9: quindi prendere una carta equivale ad apporre il proprio segno (croce o cerchio) sulla corrispondente casella di questo quadrato, e riuscire a fare tris sarà equivalente a prendere tre carte la cui somma sia 15, prima che vi riesca l’avversario.

## Due varianti del filetto.

Nella variante *misère* chi fa tris perde. Come si risolve il gioco? E come finirà una partita senza errori?

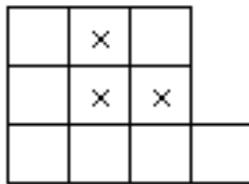
Tra le molte altre varianti del filetto, anche in più dimensioni, mi piace infine ricordarne una semplice – che rimane in due dimensioni – ideata da Martin Gardner (1914-2010), il grande e prolifico divulgatore che, dal 1957 al 1980 e oltre, curò la rubrica di giochi matematici della rivista mensile *Scientific American* (*Le Scienze*, in Italia): si aggiunge una casella alla riga orizzontale in basso, sulla destra, e per vincere su questa riga più lunga bisognerà occupare tutte e quattro le caselle, mentre altrove basta il tris, anche sull’ulteriore diagonale di tre caselle che viene a crearsi.



Chi vince? Tra quali mosse deve scegliere chi inizia?

**Risposte.** Per risolvere il gioco nella variante *misère*, basta cambiare segno al valore restituito da `payoff()`. L’unica possibilità di pareggiare per il primo giocatore consiste nel segnare la casella centrale e successivamente muovere sempre in modo simmetrico alla mossa dell’avversario rispetto al centro. Provate a numerare per righe o per colonne le caselle del tavoliere e poi guardate qual è la casella numerata col complemento a 10 di una a vostra scelta...

Invece, nella variante proposta da Gardner, vince il primo giocatore con una delle tre mosse in figura:



mentre negli altri casi è patta teorica.

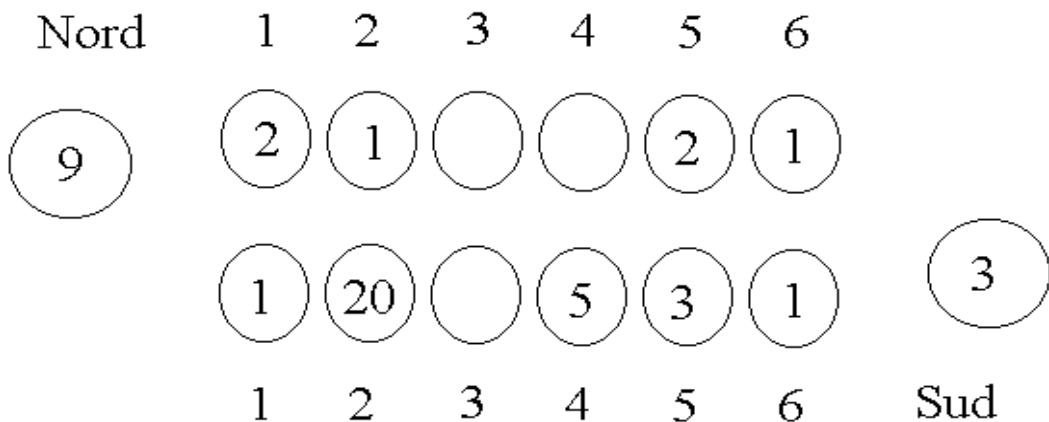
Se si giocasse a *misère* su questo tavoliere con 10 caselle, fermo restando che sulla riga in basso per perdere sia necessario occupare tutte e quattro le caselle, allora sarebbe patta teorica; in particolare, chi inizia *non* deve segnare la casella al centro, altrimenti finirà col perdere!

**Parole chiave:** teoria dei giochi, gioco strettamente determinato, grafo e albero di gioco, visita in *post-order* (dell’albero di gioco), algoritmo *minimax* (per risolvere il gioco).

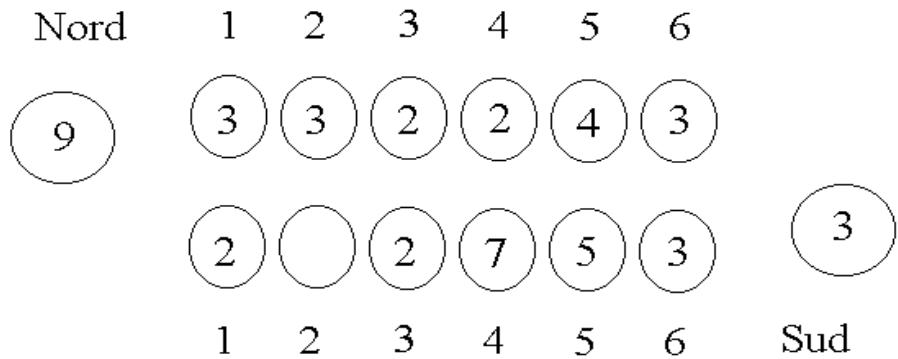
## 10. Awari... ma con un occhio agli scacchi!

All'inizio degli anni '90, allo scopo di analizzarla con l'ausilio del computer, un gruppo di ricercatori olandesi (L. V. Allis, M. van der Meulen e H. J. van den Herik) scelse una delle tante varianti di un antichissimo gioco di origine africana – ormai ben nota col nome *Awari* anche nella comunità informatica – che fu poi “risolta” nel 2002 da alcuni loro colleghi.

Le regole sono semplici, il gioco non altrettanto. Si gioca su un tavoliere che ha, per ciascuna delle due parti, una fila di 6 buche (contenenti all'inizio 4 semi ciascuna) e un “granaio” (che conterrà i semi catturati). Il giocatore che deve muovere sceglie una delle proprie buche, non vuota; preleva tutti i semi che vi sono contenuti e, procedendo in senso antiorario, li semina a uno a uno nelle buche successive, anche nel campo opposto, saltando poi la buca originaria se i semi da questa prelevati sono più di 11: fatta questa operazione, tale buca rimane quindi sicuramente vuota. Se l'ultimo seme cade in una buca avversaria che contiene o 2 o 3 semi (compreso quello appena seminato) questi sono catturati, e così di seguito, a ritroso, finché si incontrano buche nel campo avversario con 2 o 3 semi.



Se, per ipotesi, la situazione (o *stato del gioco*) fosse come nella figura sopra con tratto a Sud, questi può scegliere tra cinque buche, rimanendo esclusa la buca 3, l'unica vuota. Se Sud gioca 6 (mossa che indicheremo scrivendo S6), semina l'unico seme nella buca 6 del campo avverso, vi cattura i due semi e li aggiunge ai tre del proprio granaio. Se gioca S5, arriva a seminare l'ultimo seme nella buca 5 del campo avverso e cattura 5 semi (dalle buche 5 e 6 del campo avverso); se gioca S4 oppure S1, non cattura nulla; ma se gioca S2, giunge dopo la semina (che termina nella buca 2 del campo avverso) alla situazione illustrata nella figura seguente:



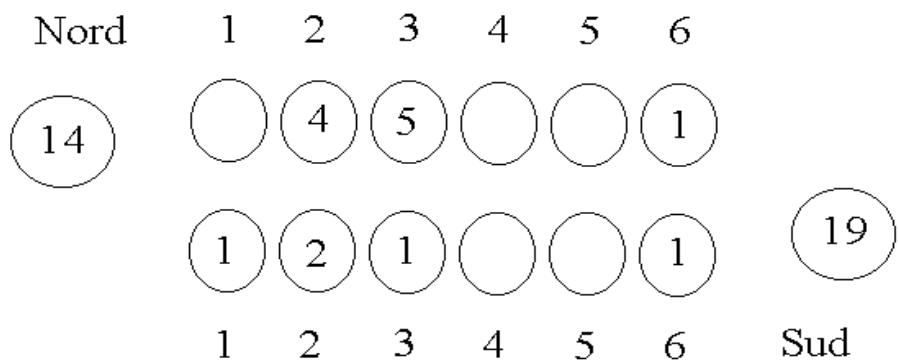
e quindi cattura 7 semi (dalle buche 2, 3 e 4 del campo avverso), raggiungendo quota 10 nel granaio. A questo punto, Nord può giocare N1, catturando 3 semi (dalla buca 3 del campo di Sud)...

Ovviamente, non è detto che la mossa che porta al massimo risultato immediato sia la migliore: spesso la tattica *greedy* (golosa) conduce alla sconfitta, se l'avversario gioca bene!

Se tutte le buche nel campo del giocatore di turno sono vuote, egli non può muovere; allora l'avversario trasferisce nel proprio granaio i semi che sono eventualmente rimasti nel proprio campo, e la partita finisce. Il punteggio è determinato dalla differenza tra i contenuti dei rispettivi granai.

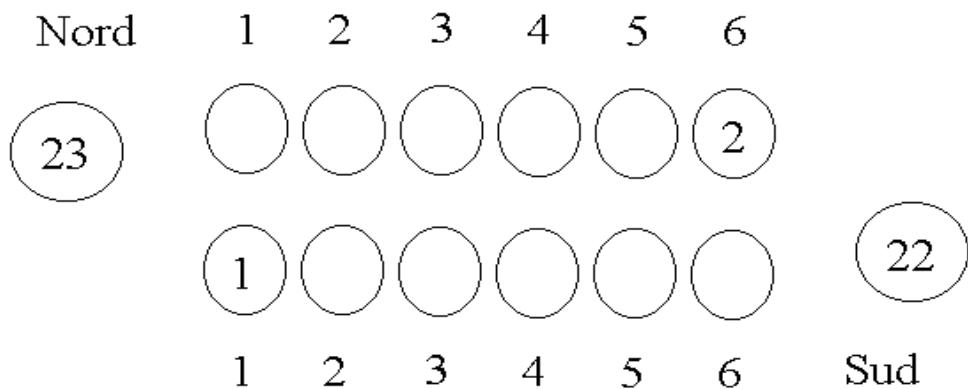
Questa sarebbe la regola più semplice per stabilire la fine di una partita; tuttavia, per prolungare il gioco, nell'Awari qui considerato vige una regola *fair* (leale, onesta): non sono consentite mosse che lasciano l'avversario senza semi, eccetto nel caso in cui *tutte* portino inevitabilmente a questa situazione. (Non è permesso comunque saltare un turno, ciò che invece è previsto in altre varianti.)

Ad esempio, ipotizzando lo stato di figura con tratto a Sud:



se questi muove dalla buca 6 (S6) allora deve mangiare e guadagna 2 semi, ma poi Nord vince 27 a 21 perché entrambe le mosse che può fare lasciano Sud senza semi. Se invece la buca 6 di Nord fosse vuota, a seguito della stessa mossa di Sud, adesso Nord sarebbe obbligato a muovere lo stesso unico seme dalla propria buca 6 nella 5: infatti, ora le altre due mosse sono entrambe illecite poiché lasciano Sud senza semi.

Infine, se una stessa posizione si ripete per tre volte, la partita termina: secondo la nostra variante, i semi rimasti sono divisi *esattamente* a metà tra i due giocatori, e aggiunti ai rispettivi granai. Ad esempio, ipotizzando il tavoliere di figura:



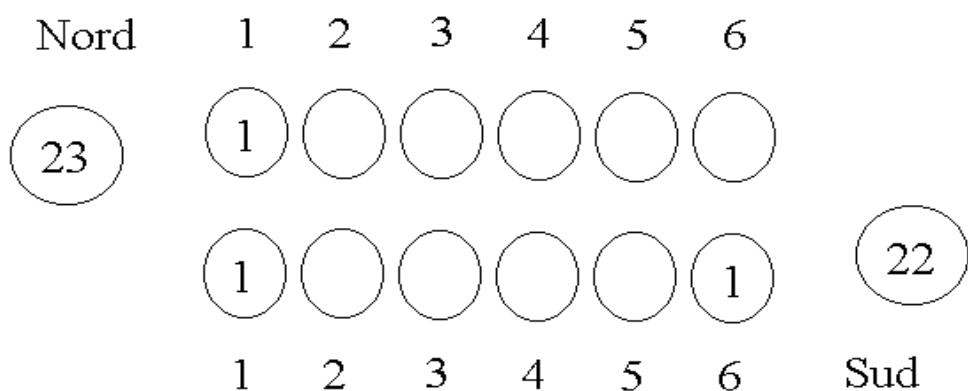
se tocca a Sud, allora Nord riesce a catturare tutti e tre i semi: S1, N6, S2, N4, S3, N3, S4, N5, S5, N4, S6, e la partita termina 26 a 22 per Nord.

Se invece tocca a Nord, questi riesce a forzare la ripetizione di posizione: N6, S1, N4, S2, N3, S3, N5, S4, N4, S5, N2, S6, N1 (obbligata), S1, N3, S2, N6, S3, N5, S4, N4, S5, N2, S6, N1 (obbligata), S1, N3, ... e la partita termina 24.5 a 23.5 per Nord.

Tuttavia, se Nord non sta attento, può perdere tutti e tre i semi! Riprendiamo dall'inizio: N6, S1, N4, S2, N3, S3, N5, S4, N4, S5, N3 (ecco l'errore di Nord!), S6, N2 (obbligata), S1, N6, S2, N5, S3, N4, S4, N3, S5, N2, S6, N1, S2, N6, S3, N5, S4, N4, S1, N3, S2, N2, S3, N1, e la partita finisce 25 a 23 per Sud.

Se, nella posizione illustrata, inizia Nord, ce la può fare a catturare tutti e tre i semi rimasti sui campi? [Mi pare proprio di no!]

Vediamo un altro caso.

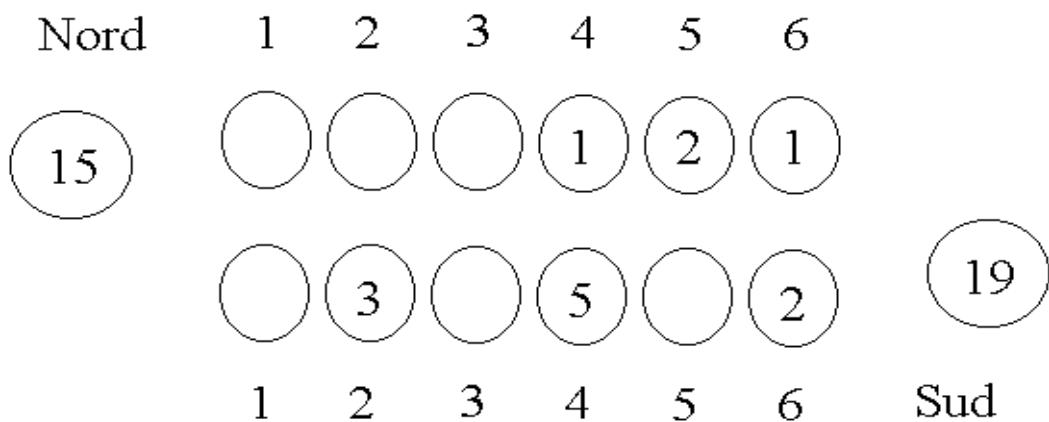


Se tocca a Nord, questi cattura due semi con l'unica mossa che può fare, e dopo S6 Nord guadagna anche l'ultimo seme rimasto; risultato: 26 a 22 per Nord.

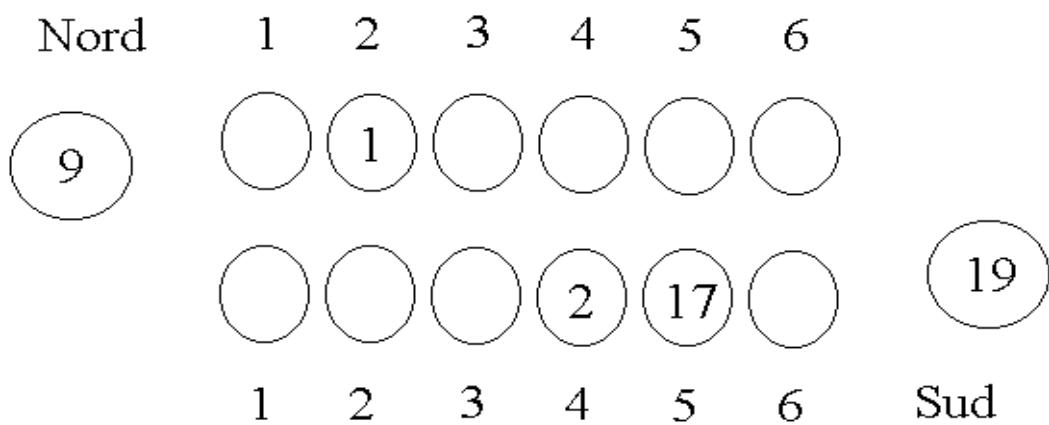
Se invece tocca a Sud, questi deve giocare S1, dopodiché: N1, S6 (obbligata), N6, S2, N5, S3, N4, S4, N3, S1, N2, S2, N1, e la partita finisce 25 a 23 per Sud.

Se la buca 1 di Sud fosse vuota, e il granaio di Sud contenesse 23 semi, la ripetizione di posizione sarebbe forzata, chiunque muovesse, e la partita finirebbe in parità, 24 a 24. (Secondo altre varianti, invece, i due semi sarebbero catturati dal giocatore che per primo li trova entrambi nel proprio campo: in questo caso, colui al quale *non* tocca muovere.)

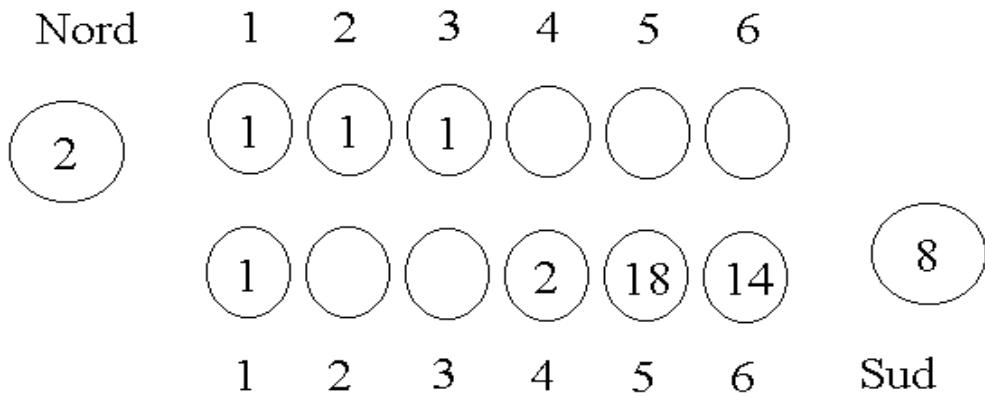
Passiamo infine a qualche situazione che può presentarsi nel corso di una partita. La prima, riportata in un articolo dei ricercatori olandesi, è la seguente, con tratto a Sud:



Sud non può giocare S4 per non lasciare vuoto il campo di Nord. (In altre varianti, questa mossa sarebbe lecita e determinerebbe la fine della partita, o altrimenti potrebbe essere fatta ma senza catturare i semi dal campo avverso.) La mossa giusta per Sud è S6, con la quale guadagna 5 semi. Il miglior punteggio finale per Sud (garantito, se entrambi giocano al meglio) è 27 a 21: come proseguirà la partita? La seconda situazione che proponiamo, sempre con tratto a Sud, è questa:



Per la regola *fair*, Sud è obbligato a giocare S4 e, dopo N2, deve muovere S6; a questo punto N6 sarebbe perdente per Nord. Come potrebbe proseguire la partita? La terza e ultima proposta, che lasciamo commentare ai nostri lettori, è riportata alla pagina successiva; il tratto è ancora a Sud.



Come abbiamo accennato tra le righe, le principali varianti di questo gioco sono legate all'impossibilità di muovere o al finale di partita, e poi allo svolgimento di un *match* (o sfida) costituito da più partite. Ad esempio: non appena uno dei due giocatori arriva ad accumulare almeno 25 semi nel proprio granaio, vince la partita.

### Un po' di storia...

Sono tantissimi i giochi appartenenti a questa famiglia, detta degli Awélé (o Mankalah, secondo la denominazione araba, oggi predominante nella letteratura anglosassone; Mancala, nella trascrizione in italiano, con accento a piacere): variano il numero delle file di buche, il numero delle buche in ciascuna fila, il numero dei semi, la presenza o meno dei granaio, e conseguentemente le regole.

Considerando tutti i luoghi in cui sono diffusi – nell'Africa pressoché intera, isole Comore comprese, in Medio Oriente e nei paesi arabi, in India, Sri Lanka e Maldive, nelle Filippine e in Indonesia (a Giava, una bella e complessa variante, Congklak o Congkak, è apprezzata, in particolare, dalle giovani donne), e persino nelle Americhe: in Brasile, nelle Guyane e nelle Antille, in Louisiana... – si contano oltre duecento nomi diversi e almeno altrettante regole differenti!

Sebbene da noi non possa affatto dirsi popolare, in alcuni paesi europei, come Francia e Inghilterra, si tengono dei campionati di Awari, e una variante è pure contemplata nelle gare delle Olimpiadi degli Sport della Mente!

Gli Awélé sono annoverati tra i giochi più antichi: certamente vi giocavano gli Egizi 35 secoli fa, e a due secoli ancor prima risale il più antico tavoliere che si conosca, fatto di terracotta e ritrovato nell'isola di Creta; quindi la diffusione riguardò anzitutto il continente africano.

Un bel libro che ne tratta è *Giochi africani*, di Carlo Zampolini (Sansoni Editore, Firenze 1984, pp. 23-78). Assai interessanti, oltre a quelle matematiche, le valenze rituali o persino magiche: non si giocava dopo il tramonto, ma si lasciava la tavola pronta affinché durante la notte potessero giocare le entità ultraterrene. Quasi ogni villaggio aveva le proprie regole, il cui denominatore comune era il ciclo *antiorario*, forse legato a quello celeste (a nord dell'equatore) o a quello terreno: le pedine (in

mancanza di meglio, semi, nòccioli o sassolini) rappresentano le stelle, la tavola (alla peggio, due file di buche scavate in terra) rappresenta il cielo; allo stesso tempo, esse rinviano, simbolicamente, alla semina e al susseguirsi dei mesi o delle stagioni. In un torneo notturno, all'uopo organizzato, si cimentavano gli aspiranti capo-tribù, senza manifestare alcun sentimento ostile né aggressività reciproca, in ciò aiutati dalla pacifica atmosfera che avvolge il gioco, dal movimento circolare di pedine uguali, non divise in eserciti contrapposti – come invece accade negli scacchi – ma appartenenti ora all'uno ora all'altro dei due contendenti: «il senso di proprietà è appena latente, le conquiste di materiale e di territorio avvengono in tranquillità. La tavola è una specie di altare, la raffigurazione del cosmo mitologico tribale: un cambio di proprietà non incide sulla ricchezza collettiva del villaggio.» (*op. cit.*, p. 28)

Tra gli altri numerosi spunti di riflessione offerti dall'opera citata, vogliamo menzionare gli usi “didattici” degli Awélé. «In molti paesi africani, [...] i processi educativi attraverso il gioco sono da secoli istituzionalizzati secondo principî di libero approccio ludico al materiale strutturato. I bambini, a seconda dell'età, vengono esentati da qualsiasi forma di rispetto sacrale perfino nei confronti di un gioco dalle forti connotazioni religiose come l'Awele. [...] L'uso di non giocare su tavole strutturate, ma nelle solite buche scavate per terra, fa sì che i bambini per imitazione si costruiscono giochi diversissimi l'uno dall'altro, con più o meno buche a seconda del grado di percezione o dell'estro del momento. Variano anche il numero di sassi, le regole di gioco e le condizioni di vincita. Non importa tanto il gioco quanto il fatto che i bambini si appropriino della struttura e apprendano gradatamente i concetti di insieme, insieme vuoto (buca vuota), numero successivo; imparano spontaneamente a contare attraverso il gioco. A mano a mano che crescono, si avvicinano al gioco strategico degli adulti, che praticheranno a tempo e luogo. [...] La veicolazione di valori educativi attraverso il gioco avviene [...] in modo naturale [...]; con buche e sassolini imitano fin da piccoli i gesti degli adulti, ma nessuno impone loro le rigide e complesse regole dell'Awele.» (*op. cit.*, pp. 56-61)

A chi avrà la possibilità e la fortuna di consultare questo prezioso libretto, raccomando di leggere... tutte le sue pagine, ma in particolare le note finali sugli Awélé (pp. 76-78). L'autore ha condiviso la certezza che i giochi a base logico-matematica provochino negli studenti la richiesta di istruzione matematica: sicché, con l'occhio sul nostro obiettivo, accingiamoci pure ad affrontare alcuni aspetti di tipo “informatico”, legati alla realizzazione di un programma in grado di competere con un avversario, umano o automa che sia; tali aspetti costituiscono un approfondimento di quanto già abbiamo visto a proposito del gioco del tris.

## L'Aware è stato risolto!

Anzitutto, come già anticipato in apertura di capitolo, nell'estate del 2002 John W. Romein e Henri E. Bal, della Vrije Universiteit di Amsterdam, hanno annunciato di

aver *risolto* (in senso forte) l’Awari, determinando la mossa o le mosse migliori (cioè quelle “più giuste”, tra cui scegliere quella da fare) e il risultato finale per quasi 900 miliardi di posizioni (essenzialmente differenti), che ne costituiscono lo *spazio degli stati legali*, cioè raggiungibili (da uno dei due giocatori) in una partita regolare. Con “risultato finale” s’intende il punteggio che si avrà alla fine della partita (il numero di semi nei rispettivi granai) assumendo che, a partire dallo stato considerato, entrambi i contendenti giochino al meglio delle proprie possibilità, ossia che il gioco si svolga in modo *perfetto* (ciascun giocatore facendo, ad ogni suo turno, una mossa che gli garantisca il massimo punteggio finale ottenibile).

La conclusione a cui sono giunti gli autori citati è che, in questa ipotesi, a partire dallo stato iniziale, la partita termina in parità – così come accade per il filetto o per la dama sulla scacchiera 8×8.

I risultati di questa analisi furono condensati – impiegando il minimo numero di bit! – in un enorme database, che complessivamente occupava circa 778 GB (gigabyte), e resi pubblici in internet, insieme con la possibilità di ottenere varie statistiche e di competere con un programma “invincibile” ovvero tarabile su livelli “non perfetti” (vale a dire più “umani”, che lasciavano all’avversario una qualche possibilità di vittoria). Sfortunatamente, la macchina che ospitava database e Awari Oracle adesso non è più disponibile…

Come hanno proceduto i ricercatori olandesi? Per sommi capi, servendosi di un computer dotato di 144 processori in parallelo e una memoria principale di 72 GB, hanno effettuato un’*analisi retrograda*, partendo dall’unico stato con tutte le buche vuote per arrivare ai quasi 204 miliardi di stati che presentano ancora tutti e 48 i semi in campo (peraltro, osserviamo che l’unico di questi stati che non abbia almeno una buca vuota è quello iniziale).

In tale analisi non hanno considerato la distribuzione dei semi già catturati tra i due granai, poiché essa non influisce sull’andamento ottimale del resto della partita. (Altrimenti, quanti sarebbero gli stati con tutte le buche vuote? Ricordiamo che, quando la partita termina per ripetizione di posizione, un seme può essere diviso a metà.)

Notiamo che all’inizio il tavoliere è perfettamente simmetrico, per cui non importa a chi tocchi la prima mossa: possiamo convenire che inizi Sud. (Per inciso, le sole aperture “non perdenti” sono S6-N1, S6-N2 e S6-N5, secondo la nostra notazione, simile a quella usata da Zampolini ma diversa da quella adottata dagli olandesi.) In virtù di questa convenzione, gli analisti olandesi hanno allora preso in esame tutte e sole le posizioni legali in cui Sud ha il tratto; quando deve muovere Nord, è sufficiente ruotare di 180 gradi ciascuna posizione considerata.

Infine, basta ricordare, per ogni stato legale, il numero di semi presenti nel granaio di Sud al termine di una partita che, da quello stato in poi, si svolga in modo perfetto: il complemento a 48 darà il numero di semi alla fine presenti nel granaio di Nord, e sarà dunque sufficiente esaminare i risultati finali per ciascuno degli stati raggiungibili con una mossa lecita da quello considerato, per conoscere la mossa o le mosse migliori da fare!

## Come realizzare un programma che giochi “piuttosto bene”.

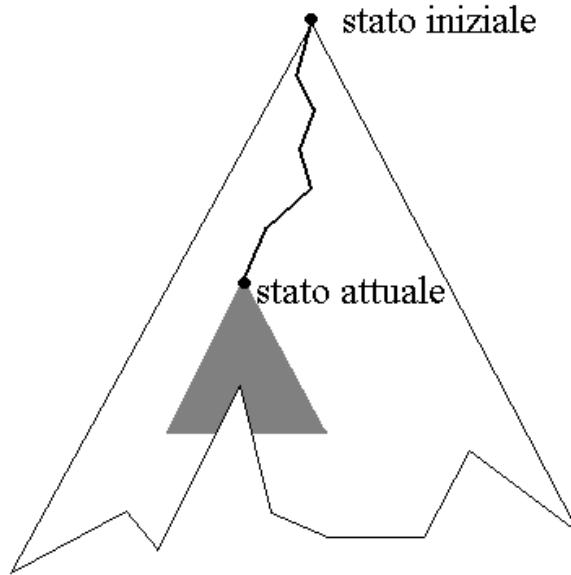
Senza ovviamente pretendere di stabilire l’ottimo, vediamo come si può modificare l’algoritmo *minimax* già usato per l’analisi completa del filetto. L’aspetto cruciale è che ora, per evitare tempi di attesa inaccettabili, non possiamo assolutamente permetterci di visitare tutto l’albero di gioco: sicché ci limiteremo ad esplorarlo fino a una certa profondità soltanto, a partire dallo stato attuale.

L’Awari rientra nella categoria di giochi da noi considerata: in particolare, l’albero di gioco è finito, sia in ampiezza (in ogni stato il numero di mosse lecite è, al più, 6), sia in altezza (prima o poi la partita termina, al limite per ripetizione di posizione), e alla fine della partita sarà determinante la differenza tra le quantità di semi accumulati nei rispettivi granai; così se, ad esempio, Sud vince 27 a 21 si può affermare che guadagna 3 semi, mentre Nord ne perde 3, rispetto alla situazione ideale di parità.

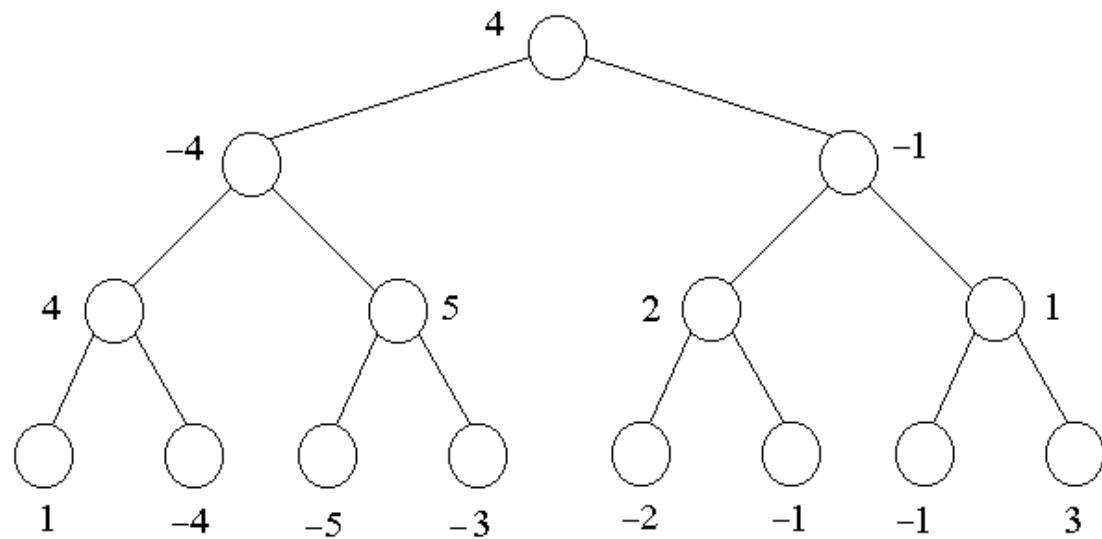
Come sappiamo, il fatto che il gioco sia *a somma zero* (sia globalmente, sia rispetto a ogni singola mossa: si veda la nota 24 a pagina 255) comporta che uno svantaggio di uno dei due contendenti equivalga a un vantaggio di pari entità dell’altro; e, di nuovo, ciascuno dei due cercherà di *minimizzare* la propria *massima perdita* possibile. Conseguentemente, il nostro programma dovrà partire dallo *stato attuale* (ossia lo stato da cui parte la ricerca, *radice del sottoalbero* che sarà almeno in parte esplorato, in cui spetta muovere al programma stesso), innescare una visita in *post-order* (prima di valutare un nodo si valutano tutti i suoi “figli”) che arrivi, al più, fino a una data profondità, e attribuire un punteggio (*payoff*) agli stati in cui l’esplorazione non proseguirà, perché stati finali o perché si trovano alla profondità massima stabilita... Ma attenzione: adesso non si tratta più di un punteggio di tipo ternario (sconfitta o pareggio o vittoria), bensì di una quantità che possa esprimere un “valore di merito” o “grado di bontà” per il giocatore che avrebbe il tratto nello stato da valutare – e che comunque, poiché il gioco è a somma zero, è sempre l’*opposto* di quanto spetterebbe all’avversario: basta quindi fare attenzione al segno. Nel nostro gioco la scelta è immediata: il punteggio è la *differenza* in semi tra il granaio del giocatore *al quale spetterebbe muovere nello stato raggiunto* e il granaio dell’avversario, che è un numero intero  $\geq -48$  e  $\leq 48$  (infatti tale differenza è in ogni caso intera, anche quando un seme sia stato spezzato a metà).

Nella figura in alto alla pagina successiva è schematizzato l’albero di gioco; il percorso dallo stato iniziale all’attuale (in cui il programma ha il tratto) rappresenta la parte di partita già giocata, mentre la zona in grigio rappresenta la parte di albero che sarà esplorata (la cui altezza è limitata dalla profondità massima prestabilita).

Per realizzare dunque questo primo punto essenziale basterà un parametro in più, la profondità da raggiungere relativamente al nodo corrente, che sarà decrementata ad ogni chiamata ricorsiva e, quando essa scenderà a zero, il nodo corrente sarà considerato terminale. Al solito, una volta valutati tutti i nodi “figli”, il valore del nodo “padre” è il minimo dei valori dei figli cambiato di segno.



Ad esempio, consideriamo il caso – del tutto ipotetico – rappresentato nella figura sottostante, dove la profondità stabilità è 3 e in ciascuno degli stati considerati il giocatore di turno ha due mosse possibili.



Nel nodo più a sinistra, il primo in cui l'esplorazione si ferma, la funzione di *payoff* – ossia la funzione di valutazione (dello stato) – assegna punteggio 1: ciò significa che il giocatore a cui spetterebbe muovere (in questo esempio, per inciso, è l'avversario, se in radice il programma ha il tratto) è in vantaggio di un seme; mentre  $-4$ , nel secondo nodo, significa che egli è in svantaggio di 4 semi. A questo punto, essendo  $\text{min}(1, -4) = 4$ , al nodo padre è attribuito punteggio 4; il motivo è presto detto: al giocatore che qui deve muovere conviene la seconda mossa, che lo porterà in vantaggio di 4 semi. Al “fratello” di questo nodo, dopo che i suoi figli sono stati valutati  $-5$  e  $-3$ , è attribuito punteggio 5; quindi al loro padre sarà attribuito punteggio  $\text{min}(4, 5) = -4$ , poiché chi deve muovere non rischierà di incorrere in uno svantaggio di 5 semi...

Diamo ora alcune indicazioni più tecniche, a beneficio di coloro che vorranno cimentarsi nella costruzione di un programma simile a quello realizzato nel 2009 da un mio brillante allievo. Per inciso, questo programma era dotato della possibilità di giocare contro sé stesso, per poter confrontare le prestazioni di diversi algoritmi (alcuni dei quali saranno illustrati nel seguito di questo capitolo), magari spinti a diverse profondità di analisi.

Usando sempre il linguaggio C++, partiamo col definire una *classe* board, un'istanza della quale sia in grado di rappresentare uno stato del gioco: questa volta è il tavoliere con la disposizione dei semi, più l'informazione binaria che dice “a chi spetta muovere”. Quindi i *campi* saranno di tipo *array* di interi per le file di buche, *float* per i granai e *booleano* per il giocatore di turno.

La più semplice versione del *metodo* di analisi, ricorsiva all'interno di un ciclo **for**, ricalca quella già vista nel capitolo precedente (con l'effetto collaterale di ricordare una mossa) e può essere codificata come segue, a prescindere dal gioco specifico.

```
int board :: minimax (unsigned int depth, int & move) const {
    board C;
    int i, value, new_move, new_value;
    if (depth == 0 || is_a_leaf()) { move = 0; return payoff(); }
    value = -INFINITY;
    for (i = 1; i <= MAX_MOVES; i++)
        if (is_legal(i)) {
            C = *this;           // assegnazione tra board (overloaded)
            C.make_move(i);      // esegue la mossa che svuota la buca i
                                  // sul board C (copia dell'attuale)
            new_value = - C.minimax(depth - 1, new_move);
            if (new_value > value) { value = new_value; move = i; }
        }
    return value;
}
```

Conformemente a quanto abbiamo detto, l'esplorazione lungo un ramo dell'albero si ferma quando il parametro *depth* ha valore zero oppure quando il *board* corrente al quale il metodo è applicato (**\*this**) è una *foglia*, ossia uno stato di fine del gioco. Ricordiamo che, comunque, il punteggio restituito da *payoff()* deve essere relativo al giocatore che dovrebbe muovere nello stato raggiunto; inoltre, in generale, la progettazione della funzione di *payoff* è piuttosto delicata, poiché si assume implicitamente che ciascuno dei due contendenti giochi al meglio delle proprie possibilità proprio in base a questa particolare funzione di valutazione.

La costante *INFINITY* deve essere maggiore del massimo punteggio attribuibile a un *board* (perciò  $> 48$ ). La costante *MAX\_MOVES* è pari al massimo numero di mosse tra le quali può essere scelta quella da effettuare (nel nostro caso 6). Le mosse lecite eseguibili nello stato corrente sono generate via via nel *board* ausiliario *C*; e, come al solito, una volta valutati tutti gli stati figli, il valore dello stato padre sarà proprio il minimo dei valori dei figli cambiato di segno.

Un esempio di applicazione (esterna) del metodo illustrato è contenuto nella seguente istruzione:

```
value = B.minimax(10, move);
```

dove il *board* *B* rappresenta l'attuale tavoliere di gioco, col tratto al programma.

Il primo dei due argomenti esplicativi (nell'esempio, 10) stabilisce la profondità massima da raggiungere nell'albero a partire dal nodo attuale, vale a dire il numero di mosse di cui avanzare (nell'esempio, cinque del programma alternate a cinque dell'avversario), a meno che non siano trovate foglie (né intervengano *potature*, come vedremo nel prossimo paragrafo) più in alto.

Il secondo argomento (nell'esempio, *move*) è una variabile intera trasmessa per riferimento: si tratta di un parametro di puro *output*, il cui valore finale indicherà il numero d'ordine della buca da svuotare in *B* (zero significa nessuna buca).

Il metodo *minimax* ha dunque un effetto collaterale, che si aggiunge al calcolo del punteggio associabile al *board* *B*, valore che è restituito invece come risultato esplicito (e, nell'esempio, memorizzato nella variabile intera *value*): tale punteggio è il più grande guadagno minimo garantito per il programma (e si ottiene con la mossa *move*) nei limiti dell'orizzonte esplorato (nell'esempio, dieci mosse).

A questo proposito, qualche precisazione è doverosa. Una complicazione è costituita dalla regola della ripetizione di posizione, che implica la memorizzazione, in una opportuna *lista*, di tutte le posizioni attraversate per arrivare allo stato attuale (ossia la parte di partita già giocata) e da lì allo stato corrente (l'ultimo al quale è giunta la nostra analisi, nella "zona in grigio" – si veda la figura in alto a pagina 267 – oggetto di indagine): con tutte queste posizioni dovrà essere confrontata quella che si otterrà con la mossa successiva, e qualora essa si trovi già due volte nella lista, allora dovrà essere interpretata come foglia. L'idea è simile a quella già applicata nel puzzle dei quindici.

Teniamo poi presente che di solito, nelle applicazioni concrete (si pensi anche ad altri giochi, come gli scacchi), la ricerca procede oltre la profondità massima stabilita qualora lo stato raggiunto non sia *quiescente*: ad esempio, nel gioco degli scacchi, uno stato può dirsi quiescente se non vi sono possibili catture, né minacce al Re, né si è appena verificata la presa di un pezzo (ché potrebbe trattarsi di un "sacrificio"); nell'Aware, se non vi sono buche con semi esposti a cattura. E spesso, in tanti giochi, la valutazione di uno stato, pur quiescente, non è affatto semplice ed è compiuta seguendo criteri euristici. In generale, inoltre, quando vi è una sola mossa sensata, è bene spingere l'analisi a una maggiore profondità – ma se nello stato attuale una sola mossa è lecita, allora l'esplorazione diviene inutile!

Infine, occorre ricordare che, procedendo nella partita e riapplicando *minimax*, può facilmente accadere che il nuovo scenario esplorato non garantisca più il guadagno calcolato in precedenza.

## Ricordiamo le varianti principali!

Con la funzione delineata nel precedente paragrafo, abbiamo ovviamente ricordato *una mossa migliore*, almeno quella o una di quelle (nel codice riportato è la prima, ma nell’Awari pare più opportuno valutare le mosse lecite in senso orario, da 1 a 6 per Nord, ma da 6 a 1 per Sud) che il programma dovrà eseguire nello stato attuale per assicurarsi – entro l’orizzonte di mosse fissato – almeno il punteggio ad esso attribuito al termine dell’esplorazione. Tuttavia, a parità di punteggio tra due o più nodi figli, sarebbe meglio scegliere quel nodo che ha a sua volta il figlio con punteggio massimo (per chi deve muovere); se poi persiste una situazione di parità, fare una scelta casuale tra i nodi valutati ugualmente “buoni”.

Per procedere con un occhio attento all’efficienza, bisognerebbe in realtà tenere memoria – collezionandole in una lista o in un’altra struttura adeguata – delle mosse migliori calcolate, a turno, per il programma e per l’avversario, fino all’ultimo stato al quale sia giunta l’analisi del gioco, ossia ricordare il “ramo migliore” (la continuazione più conveniente per entrambi i contendenti, la cosiddetta *variante principale*) della parte di albero esplorata, o addirittura *i rami* (le varianti) più promettenti, magari in ordine discendente di importanza – nel qual caso una lista di mosse non è sufficiente, bensì occorre una struttura di dati più raffinata...

Si osservi che questa idea è utile indipendentemente dalla profondità a cui si spinge l’analisi del gioco.

Qui ci limiteremo a dare qualche indicazione su come ricordare *le varianti principali*: più rami della parte di albero esplorata potrebbero infatti garantire il punteggio calcolato per lo stato attuale. Bisognerà allora raccogliere l’informazione convogliata da questi rami in un... *albero di mosse*!

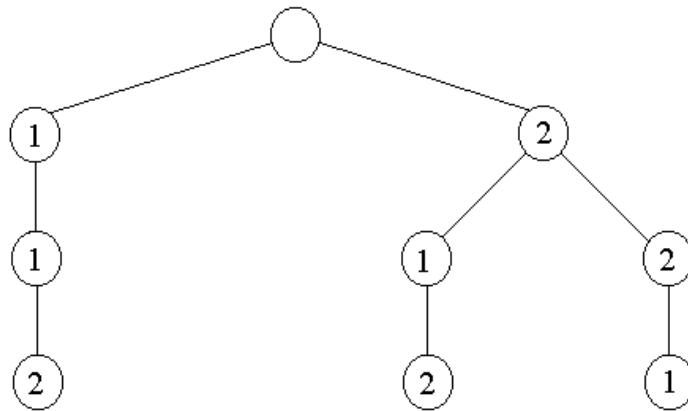
L’idea è semplice, e per illustrarla possiamo riprendere il piccolo albero (binario) del paragrafo precedente, assumendo comunque che la mossa numero 1 porti al nodo di sinistra, la mossa numero 2 a quello di destra.

Qui la variante principale è una sola: il giocatore di turno muove 1; all’avversario – per non rischiare di perdere 5 (semi) – conviene rispondere 1, a cui segue 2.

Ma se supponessimo per un momento che i due nodi appartenenti all’ultimo livello considerato e valutati -1 (in figura a pagina 267) fossero invece valutati -4, allora le varianti principali sarebbero tre e potrebbero essere rappresentate dall’albero (binario) di mosse disegnato nella pagina seguente.

È importante notare che in tale albero la radice è *fittizia*; per assicurarsi un vantaggio di 4 (semi) entro tre mosse, il giocatore di turno può scegliere indifferentemente la mossa 1 o la mossa 2: se sceglie la mossa 1, allora c’è un’unica mossa migliore per l’avversario, e cioè la 1; altrimenti le due mosse sono equivalenti per l’avversario (se questi giocasse contro uno sprovveduto, farebbe tuttavia meglio a muovere 2 per conservare la speranza di un guadagno)...

Il metodo *minimax* dovrà sempre ricevere come secondo argomento esplicito (per riferimento) un *albero di interi* pvs contenente la sola radice (con etichetta non assegnata o convenzionalmente posta a valore zero).



Dopo l'applicazione ricorsiva, si etichetta con l'intero *i* (che è il numero d'ordine della mossa appena valutata) la radice dell'albero da essa preparato; questa operazione di etichettatura può anche essere anticipata. Indichiamo con *new\_pvs* l'albero così ottenuto.

Se *new\_value* > *value* (una mossa migliore è stata trovata), allora si devono rimuovere tutti gli eventuali sottoalberi della radice di *pvs* (corrispondenti alle mosse migliori precedentemente trovate), rimpiazzandoli con *new\_pvs*; altrimenti, se *new\_value* == *value* (si tratta di una mossa equivalente alle migliori finora trovate) si aggiunge *new\_pvs* ai sottoalberi della radice di *pvs*.

Pertanto il metodo *minimax* potrebbe essere così delineato:

```

int board :: minimax (unsigned int depth, TREE<int> & pvs) const {
    board C;
    int i, value, new_value;
    TREE<int> new_pvs;
    if (depth == 0 || is_a_leaf()) return payoff();
    value = -INFINITY;
    for (i = 1; i <= MAX_MOVES; i++)
        if (is_legal(i)) {
            C = *this;
            C.make_move(i);
            rendere vuoto l'albero new_pvs;
            aggiungere un nodo (radice) con etichetta i
            all'albero new_pvs;
            new_value = - C.minimax(depth - 1, new_pvs);
            if (new_value > value) {           // mossa migliore
                value = new_value;
                rimuovere tutti i nodi dall'albero pvs tranne la radice;
                aggiungere una copia di new_pvs come sottoalbero
                della radice di pvs;
            } else if (new_value == value) { // mossa equivalente
                aggiungere una copia di new_pvs come sottoalbero
                della radice di pvs;
            }
        }
    return value;
}

```

Naturalmente, un'applicazione (esterna) di questo metodo dovrà essere preparata nel seguente modo:

```
rendere vuoto l'albero pvs;  
aggiungere un nodo (radice) all'albero pvs  
    (eventualmente con etichetta 0);  
value = B.minimax(10, pvs);
```

dove, come al solito, il *board* B rappresenta l'attuale tavoliere di gioco, con il tratto al programma. Alla fine, in value sarà memorizzato il più grande guadagno minimo garantito per il programma nei limiti dell'orizzonte esplorato (nell'esempio, dieci mosse), mentre in pvs saranno ricordate le varianti principali per ottenere tale guadagno, secondo le modalità sopra illustrate.

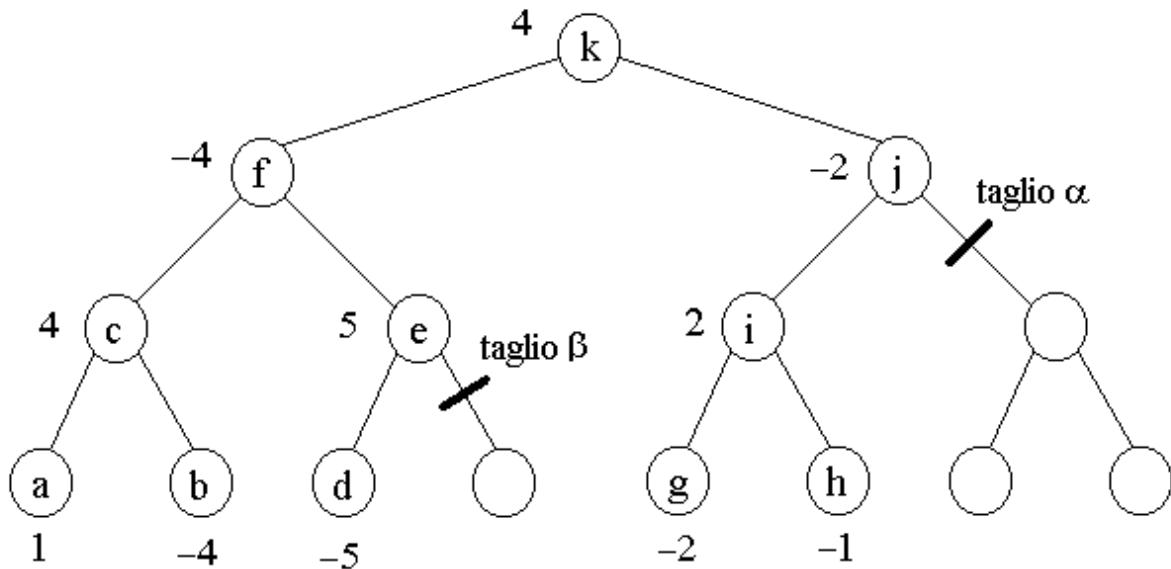
### Potature dell'albero di gioco.

Possiamo evitare, se non è necessaria, l'esplorazione completa della parte di albero pur limitata dalla profondità massima prestabilita? Sì, possiamo sperare di aumentare l'efficienza “potando”, ov’è possibile, l’albero di gioco: se ci accorgiamo che un ramo non è buono, non perdiamo tempo per capire quanto sia cattivo! In sintesi, possiamo dire che una *potatura* evita di scendere lungo rami che non possono affinare (cioè migliorare, rendere più accurata) la valutazione dello stato attuale finora fatta.

Il primo procedimento di potatura, conosciuto come  $\alpha$ - $\beta$ , fu presentato da John McCarthy nell'estate del 1956, durante il primo seminario sull'intelligenza artificiale, il *Dartmouth Summer Research Project on Artificial Intelligence*, organizzato da McCarthy stesso insieme con Marvin Minsky, Nathaniel (Nat) Rochester e Claude Shannon; tra le altre cose, proprio in quella circostanza McCarthy suggerì le idee in base alle quali poi progettò il linguaggio LISP.

Un'approssimazione *branch-and-bound* dell'algoritmo di potatura  $\alpha$ - $\beta$  fu codificata e utilizzata per la prima volta in un programma per gli scacchi, finito nel 1958, da Allen Newell, Herbert A. Simon (futuro premio Nobel per l'economia nel 1979) e J. C. (Cliff) Shaw, all'epoca ricercatori presso il Carnegie Institute of Technology (CIT) di Pittsburgh, Pennsylvania. Questo programma fu chiamato NSS dalle iniziali dei loro cognomi e fu sviluppato su un elaboratore IBM 704 nel linguaggio IPL-II (IPL è l'acronimo di *Information Processing Language*), il primo linguaggio funzionale per computer – da loro stessi presentato in occasione del seminario sopra menzionato – che diede corpo alle pionieristiche idee sulla ricorsione e sull'elaborazione di liste con allocazione dinamica della memoria. Purtroppo questo linguaggio, messo a punto da Shaw per il computer JOHNNIAC (*John von Neumann Numerical Integrator and Automatic Computer*) della RAND Corporation, era ancora a un livello sperimentale, ciò che penalizzò le prestazioni del programma, tanto che impiegava circa un’ora per fare una mossa!

Per illustrare la tecnica di potatura  $\alpha$ - $\beta$ , riprendiamo l'albero (binario) del nostro esempio, così com'era in origine:



Procedendo in *post-order*, una volta valutato il quarto nodo “d” a  $-5$ , sappiamo già che al nodo padre “e” corrisponderà un valore  $\geq 5$ ; ma al suo nodo fratello a sinistra (il terzo valutato) “c” abbiamo attribuito valore  $4$ , che è  $\leq 5$ , sicché la valutazione del nodo “f” non sarà influenzata dal valore “vero” del nodo “e”: quindi può essere evitata la valutazione di tutti i rimanenti figli del nodo “e” (nel nostro caso, uno solo).

Notiamo che in effetti ciò è indipendente dal punteggio del nodo “tagliato”: anche ipotizzando per esso un punteggio  $< -5$ , al nodo “e” sarebbe attribuito un valore  $> 5$ . La stessa cosa succede al livello superiore, dopo la valutazione del nodo “i”: al nodo padre “j” corrisponderà di certo un valore  $\geq -2$ , che non influenzera la valutazione del nodo attuale “k” dato che il valore già attribuito al nodo “f”, ossia  $-4$ , è  $\leq -2$ . Possiamo così evitare la valutazione di tutti i rimanenti figli del nodo “j”.

Scendendo a un livello più tecnico – sempre per chi sia interessato! – il miglioramento che abbiamo illustrato è compiuto dal metodo codificato alla pagina successiva (che ricorda una mossa, sicuramente non migliorabile entro la profondità fissata), basato sulla formulazione dell’algoritmo  $\alpha$ - $\beta$  detta *fail-hard*, data da Donald E. Knuth e Ronald W. Moore in una loro analisi pubblicata nel 1975.

Forniamo, anche in questo caso, un esempio di chiamata esterna:

```
value = B.alphabeta(-INFINITY, +INFINITY, 10, move);
```

L’intervallo  $[a, b]$  è detto *finestra di ricerca*: indicando con  $v$  l’opposto del valore attribuito a un nodo figlio del corrente, se  $v$  supera il corrente valore di  $a$  allora ne diventa il nuovo valore, e se raggiunge o supera anche quello di  $b$  allora è inutile esplorare i rimanenti figli (e i loro discendenti).

```

int board :: alphabeta (int a, int b, unsigned int depth,
                      int & move) const {
    // Precondizione: a < b
    board C;
    int i, value, new_move;
    if (depth == 0 || is_a_leaf()) { move = 0; return payoff(); }
    for (i = 1; i <= MAX_MOVES; i++)
        if (is_legal(i)) {
            C = *this;
            C.make_move(i);
            value = - C.alphabeta(-b, -a, depth - 1, new_move);
            if (value > a) { a = value; move = i; }
            if (a >= b) return a; // potatura
        }
    return a;
}

```

In virtù della precondizione  $a < b$ , l'ultima istruzione **if** (la potatura) può essere spostata come terza istruzione nel corpo dell'**if** precedente.

Servendoci ancora dell'esempio fatto sopra, il taglio  $\beta$  corrisponde a un "fallimento verso l'alto":  $5 \geq 4 = \beta$ , equivalente a  $-5 \leq -4 = \alpha$ ; il taglio  $\alpha$  corrisponde invece a un "fallimento verso il basso":  $2 \leq 4 = \alpha$ , equivalente a  $-2 \geq -4 = \beta$ .

Attenzione: in entrambi i casi, l' $\alpha$  è quello corrente dell'attivazione *superiore* (*chiamante*)!

Vogliamo ancora ricordare, qui di seguito, un paio di successivi raffinamenti che, specialmente all'aumentare della profondità, hanno dato migliori prestazioni (cioè un maggior numero di tagli, e quindi un minor tempo di esecuzione) rispetto all'algoritmo  $\alpha$ - $\beta$ ; entrambi restano comunque di assai semplice realizzazione – basta "ritoccare" un poco il metodo `alphabeta`, di cui conservano l'*interfaccia*: hanno infatti la stessa funzionalità e la stessa modalità d'impiego – rispetto a procedimenti più moderni ma più sofisticati, che usano strutture di dati ausiliarie piuttosto complesse.

Il primo è conosciuto come *Minimal Window Principal Variation Search*, abbreviato in *PVS*, ed è dovuto a T. Anthony Marsland e Murray Campbell; il codice riportato a pagina 275 si basa su una versione migliorata rispetto all'originale del 1982.

L'ultimo metodo di "potatura" che qui citiamo è dovuto ad Alexander Reinefeld, il quale lo propose per la prima volta nel 1982 e poi lo pubblicò, corretto, sette anni più tardi; è conosciuto come *NegaScout* (un miglioramento dell'algoritmo *SCOUT*, ideato da Judea Pearl nel 1980) e può essere realizzato come si vede a pagina 276.

Un fatto è da tener comunque presente: in generale, adottando uno degli algoritmi di potatura qui delineati, non è più valida una scelta casuale tra le mosse che infine risultano ugualmente "buone". Infatti, per come operano questi algoritmi, i nodi non valutati "compiutamente" (fratelli destri del primo scelto tra i migliori) possono, in realtà, rivelarsi meno promettenti!

```

int board :: pvs (int a, int b, unsigned int depth,
                  int & move) const {
    board C;
    int i, value, new_move, aux_a;
    bool first;
    if (depth == 0 || is_a_leaf()) { move = 0; return payoff(); }
    first = true;
    for (i = 1; i <= MAX_MOVES; i++)
        if (is_legal(i))
            if (first) {
                move = i;
                C = *this;
                C.make_move(i);
                aux_a = - C.pvs(-b, -a, depth - 1, new_move);
                first = false;
            } else {
                if (aux_a >= b) return aux_a; // normale potatura
                if (aux_a > a) a = aux_a; // condizione "fail-soft"
                C = *this;
                C.make_move(i);
                // esegue una ricerca con "finestra nulla":
                value = - C.pvs(-a - 1, -a, depth - 1, new_move);
                if (value > aux_a) {
                    move = i;
                    if (value > a && value < b && depth > 2)
                        // meglio di depth > 1, pure corretto; a seguito di
                        // eventuale "fail-high", esegue una nuova ricerca:
                        aux_a = - C.pvs(-b, -value, depth - 1, new_move);
                    else
                        aux_a = value;
                }
            }
    return aux_a;
}

```

```

int board :: negascout (int a, int b, unsigned int depth,
                      int & move) const {
    board C;
    int i, value, new_move, aux_a, aux_b;
    bool first;
    if (depth == 0 || is_a_leaf()) { move = 0; return payoff(); }
    aux_a = a;
    aux_b = b;
    first = true;
    for (i = 1; i <= MAX_MOVES; i++)
        if (is_legal(i)) {
            C = *this;
            C.make_move(i);
            value = - C.negascout(-aux_b, -aux_a, depth - 1, new_move);
            if (value > aux_a && value < b && !first && depth > 1) {
                // qui la condizione depth > 2 non sarebbe corretta!
                move = i;
                // esegue una nuova ricerca:
                aux_a = - C.negascout(-b, -value, depth - 1, new_move);
            }
            if (value > aux_a) { move = i; aux_a = value; }
            if (aux_a >= b) return aux_a; // normale potatura
            aux_b = aux_a + 1; // imposta una nuova "finestra nulla"
            first = false;
        }
    return aux_a;
}

```

Applicati all'albero binario che sopra ci è servito come semplice esempio, sia pvs sia negascout operano gli stessi tagli di alphabeta, quindi evitano la valutazione di quattro nodi. Per apprezzare qualche differenza, si possono provare questi casi separatamente:

- cambiare il punteggio della quinta foglia (partendo da sinistra), facendolo scendere da -2 a -5 o -6;
- scambiare tra loro i punteggi attribuiti alla seconda e alla terza foglia (sempre da sinistra), oppure cambiare il punteggio della prima foglia (a sinistra) da 1 a -6.

Ultime osservazioni importanti: tutti i metodi che abbiamo definito si possono usare, senza alcuna modifica, anche nel progetto di altri giochi tra due avversari della classe di giochi considerata; inoltre, essi sono tutti *funzionalmente equivalenti* a minimax (sugli stessi dati, forniscono gli stessi risultati) e quindi intercambiabili. Infine, notiamo che una qualsiasi tecnica di potatura può essere applicata anche all'analisi completa di un gioco – purché tale analisi sia praticabile! – vale a dire in mancanza di un limite prestabilito alla profondità da raggiungere: basta infatti eliminare dalla codifica il parametro `depth` e le espressioni che lo riguardano...

Nelle prove fatte in laboratorio con l’Awari, abbiamo potuto constatare la *crescita esponenziale* del tempo di calcolo richiesto dal metodo *minimax*: in particolare, quando la profondità di analisi aumenta di un’unità (cioè una mossa), il tempo di attesa della prima mossa del programma si moltiplica per un fattore circa uguale a 5, e ciò corrisponde a quanto ci si poteva aspettare, poiché nella fase iniziale del gioco accade spesso che un giocatore disponga di 5 mosse possibili (una delle sue buche è rimasta vuota al suo turno precedente e, se non è stata “seminata” con l’ultima mossa dell’avversario, è ancora vuota).

Il tempo di risposta degli algoritmi di potatura varia piuttosto sensibilmente durante una partita, spesso riducendosi drasticamente (specie all’inizio e talvolta pure per le ultime mosse) rispetto a quando è applicato il metodo *minimax*. Mediamente, nell’arco di una partita, abbiamo riscontrato, ad esempio, una riduzione intorno al 50% a profondità tra 6 e 8, oltre l’80% a profondità 9 e circa il 90% a profondità 10. Scendendo a profondità maggiori, vediamo poi gli effetti dei miglioramenti apportati all’algoritmo  $\alpha$ - $\beta$  dagli ultimi due metodi: ad esempio, a profondità 14, il risparmio di tempo rispetto ad *alphabeta* ha superato il 15%, sia con *pvs* sia con *negascout*.

Abbiamo pure provato questi algoritmi nell’analisi completa del filetto, fino alle foglie e *senza sfruttare le simmetrie* degli stati di gioco: ovviamente *minimax* esamina fino in fondo tutte le 255168 partite possibili, valutando tutti i 549946 nodi dell’albero, mentre gli altri metodi sopra presentati arrivano, rispettivamente, alla fine di 7330, 6827 e 6831 partite, operando dei tagli ai livelli superiori che permettono di limitare la valutazione a 18297, 17082 e 17092 nodi soltanto, vale a dire a poco più del 3% dell’intero albero – però qui l’analisi è completa!

In generale, allargando il discorso all’intera categoria di giochi da noi considerata, l’incremento di efficienza degli algoritmi di potatura – e soprattutto degli ultimi due – si apprezza specialmente in quei giochi ove le mosse lecite sono almeno una ventina ogni volta (mentre nell’Awari sono, al più, soltanto 6) e quanto più le mosse lecite sono ordinate in una lista *best first* (cioè in ordine decrescente di bontà). Ecco dunque giustificata l’utilità di ricordare le varianti principali: se non altro per tenerne conto quali “primi rami” nelle successive esplorazioni!

In effetti, conviene porre la variante principale (o le varianti principali) più a sinistra, prima di passare all’iterazione seguente, qualora si usi un particolare algoritmo di ricerca chiamato *iterative deepening*: al livello di profondità appena analizzato, le mosse sono ordinate dalla migliore alla peggiore; quindi la profondità massima è incrementata, cioè si scende di un livello e si ripete la ricerca, sicché – trovando le mosse al livello superiore già ordinate – aumenta la probabilità di tagli. Di solito le iterazioni continuano finché il tempo di elaborazione disponibile, stabilito in precedenza, non si esaurisce.

Ulteriori benefici si possono ottenere introducendo degli accorgimenti per evitare di analizzare più volte uno stesso stato: tipicamente ciò viene fatto mediante l’ausilio di un’appropriata tabella, chiamata *transposition table*...

Comunque, proprio per la loro semplicità rispetto a più recenti e più sofisticati algoritmi, che usano per l'appunto ulteriori strutture di dati piuttosto complicate, i metodi che abbiamo passato in rassegna – e in particolare *NegaScout* – sono tuttora largamente impiegati, e costituiscono pur sempre il nucleo dei cosiddetti “motori scacchistici” (*chess engine*).

## A proposito di scacchi...

Alcune notizie sul “nobil giuoco” e sulla sua complessità sono state date in questi ultimi due capitoli. Superfluo ricordare che da secoli più d'uno ha tentato progetti di “macchine” in grado di giocare una partita... Ma il primo vero automa scacchistico fu realizzato nel 1912 da Leonardo Torres y Quevedo, all'epoca presidente della Accademia delle Scienze spagnola.

Si tratta di un dispositivo capace di giocare finali con Re e Torre contro Re solo, fino a dare il matto (sebbene non sempre nel minimo numero di mosse), ed è pure, con tutta probabilità, il primo *automa logico interattivo* che sia stato realizzato con successo nel mondo. Otto anni dopo, con l'aiuto del figlio, Torres approntò una versione più evoluta di tutta l'apparecchiatura (visibile nella fotografia qui sotto), elettromeccanica, dotata di sensori elettrici e barre mobili per lo spostamento dei pezzi, anch'essa attualmente conservata – come “el primer ajedrecista” – nel piccolo museo allestito al Politecnico di Madrid.



Le regole imposte nella costruzione di queste macchine furono dedotte partendo dai principî sui sistemi di commutazione, che egli enunciò nei suoi celebri *Essais sur l'Automatique* (*Saggi sull'Automatica*), pubblicati nel 1914: e fu proprio in tale occasione che il termine “automatica” venne coniato. Questa memoria contiene il progetto completo di uno strumento in grado di calcolare in modo automatico il valore di una funzione algebrica per una data sequenza di valori delle variabili coinvolte; le aree conduttrici dovevano stare sulla superficie di un cilindro rotante, sì da poter realizzare le “diramazioni condizionate”, giustamente ritenute essenziali alla “capacità di discernimento” di cui dev’essere dotato un automa.

Quasi casualmente, nello stesso saggio, Torres avanza la prima proposta di aritmetica in virgola mobile: la rappresentazione in *floating point* dei dati numerici da elaborare, a noi così familiare, fu poi impiegata con successo da Konrad Zuse, il quale tra il 1935 e il 1941 realizzò i primi computer digitali controllati da programma, menzionati nel terzo capitolo.

Il nome di questo geniale tedesco ritorna a proposito: infatti, tra il 1945 e il 1946, egli raccolse in un corposo manoscritto le idee su un linguaggio di programmazione da lui stesso inventato, che chiamò *Plankalkül*. Più precisamente, si tratta di un sofisticato sistema formale per la pianificazione dell’elaborazione, ricco di concetti innovativi e sorprendentemente moderni. Tra i tanti esempi di possibili applicazioni, Zuse si dilettò a scrivere una sessantina di pagine di algoritmi per gli scacchi, ma la stesura del programma rimase sulla carta – con le sue limitazioni: la funzione di valutazione della posizione considera soltanto il bilancio di materiale, non è trattata correttamente la presa *en passant* eccetera.

Nel marzo del 1950 il matematico e ingegnere elettronico statunitense Claude E. Shannon (1916-2001) – noto soprattutto come fondatore della teoria quantitativa dell’informazione, all’epoca era ricercatore presso i Bell Laboratories, ma già nella seconda metà degli anni ’30, al MIT, si era occupato della realizzazione dell’algebra booleana mediante circuiti a relè – scrisse un articolo su *Philosophical Magazine* (Series 7, Vol. 41, No. 314, pp. 256-275) dal titolo *Programming a Computer for Playing Chess*: è la prima pubblicazione sul tema “scacchi al computer”.

Questa la tesi sostenuta da Shannon: in teoria, applicando il *principio del minimax* con l’*induzione a ritroso*, un computer può giocare una partita “perfetta” di qualsiasi gioco strettamente determinato, ma la ricerca *per forza bruta* è impraticabile negli scacchi; è dunque più saggio concentrare gli sforzi in una ricerca *con valutazione euristica*: ci si ferma comunque a una certa profondità, esaminando le sole mosse che soddisfano qualche *criterio di plausibilità*. Tuttavia, stabilire criteri adeguati non è affatto semplice e, col senno di poi, si è constatata l’impossibilità di assicurare la scelta di sequenze di mosse che magari soltanto alla fine si rivelano più vantaggiose. Per fortuna, oggi le macchine sono enormemente più veloci di allora, e inoltre possono contare su strutture di dati assai sofisticate e intere biblioteche, non soltanto di aperture e finali ma anche di posizioni e partite, sicché alla lunga l’approccio “forza bruta” – pur sempre su un orizzonte limitato, ma coadiuvato da funzioni di valutazione della posizione sempre meglio calibrate – si è rivelato preferibile.

Ma torniamo ancora a quegli anni: tra il 1950 e il 1951, Alan M. Turing – che, come abbiamo ricordato, legò il suo nome alla macchina teorica tuttora modello universale di computazione e ai limiti insiti nei processi del calcolo, nonché al test per determinare se un automa riesca ad emulare l’essere umano – giunse a scrivere e a mettere alla prova il primo programma in grado di giocare una partita, sebbene non del tutto autonomamente; la sua funzione di valutazione privilegiava il guadagno di materiale e l’esplorazione dell’albero si fermava a profondità 2!

Abbiamo già accennato al contributo fondamentale dato nella seconda metà degli anni ’50 dal CIT di Pittsburgh, dove nel 1955 Newell e Simon iniziarono la progettazione di un programma “intelligente”, che purtroppo si rivelò inefficiente... Nello stesso periodo, presso il MIT di Cambridge, Massachusetts, che divenne il più importante centro di ricerca sul gioco artificiale, Alex Bernstein e i suoi collaboratori svilupparono su un computer IBM 704 un programma ben più realistico, che in ogni stato prendeva in considerazione le sette mosse giudicate migliori (per materiale, sicurezza del Re, mobilità dei pezzi e controllo dello spazio) e spingeva l’esplorazione fino a profondità 4, impiegando circa 8 minuti per fare una mossa, cioè per analizzare  $7^4$  varianti, corrispondenti a 2801 stati complessivi: si può dire che esso esaminasse circa 6 stati al secondo. Chi è interessato può continuare da solo la ricerca di notizie sulla storia degli scacchi al computer, che in rete abbondano (partendo, ad esempio, da [https://www.chessprogramming.org/Main\\_Page](https://www.chessprogramming.org/Main_Page)).

Ricordiamo ancora che, soprattutto dagli anni ’80 in poi, col progresso delle tecnologie VLSI, si è verificata una crescita esponenziale delle prestazioni dei giocatori artificiali. Nel febbraio del 1996 per la prima volta una macchina, *Deep Blue* della IBM, riuscì a vincere una partita contro il campione del mondo in carica, il russo Garry K. Kasparov, ma perse il *match*. Nel maggio dell’anno successivo, opportunamente potenziata, riuscì invece a vincere un *match* classico di sei partite: era una macchina parallela in parte dedicata, in grado di valutare oltre 200 milioni di posizioni al secondo, mediante una funzione di valutazione che considerava più di 8000 caratteristiche!

Da allora le tecnologie, hardware e software, hanno fatto ulteriori passi avanti: oggi si disputano campionati di altissimo livello tra giocatori artificiali, e per una modica somma, o perfino gratuitamente, ci si può procurare un software pur “tradizionale” (Houdini, Komodo, Stockfish) che batte regolarmente il campione del mondo!

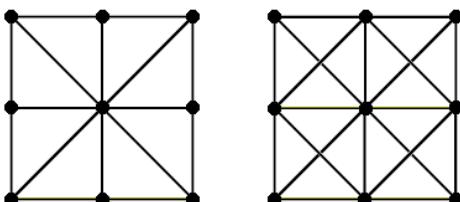
## **Chi vuol cimentarsi nella progettazione di un gioco?**

Abbiamo parlato soprattutto di algoritmi per scegliere una mossa ed eventualmente ricordare le prosecuzioni migliori, trascurando gli altri aspetti fondamentali del problema, che per certi giochi, come gli scacchi, sono tutt’altro che banali: le strutture di dati per la rappresentazione dello stato (la scacchiera, la posizione dei pezzi) e la generazione (e magari l’ordinamento) delle mosse lecite in un arbitrario stato del gioco, nonché la funzione di valutazione. Anche per quanto concerne tali componenti la letteratura in rete proliferata...

Tuttavia, a chi vuol cimentarsi nella realizzazione di un gioco al computer, non consigliamo di iniziare con gli scacchi! Esistono tanti giochi avvincenti – magari dai nomi esotici e oggi poco noti, sebbene di antiche origini – che non presentano eccessive difficoltà di progettazione, in particolare quelli “di cattura” come l’Awari, dove il vantaggio possa essere semplicemente espresso in termini di “pezzi” presi all’avversario. Qui ci limitiamo a suggerirne qualcuno: Alquerque (parimenti millenario gioco di origine egiziana, diffuso poi dagli Arabi, del quale si conoscono numerose varianti elaborate in diverse parti del mondo, ossia: Fanorona, Lau Kati Kata, Kolowis Awithlaknannai, Kharberg, ...), l’indonesiano Surakarta, la dama thailandese di Chiang Mai (*Mhark Nhaeb*), Wali (o Dara, descritto da Zampolini in *Giochi africani*, pp. 7-13), la famiglia di origini scandinave dei Tafl (Hnefatafl, Tablut, Brandubh, ...), l’hawaiano Konane, fino ai più moderni Kuba (o Akiba o Traboulet, più semplice di Abalone), Lines of Action, Flux o – perché no? – la “nostra” tela, o mulino o *telamolino*, che altro non è se non un filetto più complicato. Per inciso, le più antiche tracce di tavolieri di filetto e di tela – insieme con tavolieri di Awélé, di Alquerque e di altri giochi, certuni rimasti indecifrati – sono state trovate negli scavi del tempio di Qurna, edificato a Tebe, nell’Alto Egitto, circa 3400 anni or sono; tuttavia, c’è chi ritiene che queste tracce debbano ascriversi a un’epoca posteriore...

Della tela, che in lingua inglese è detta *Nine Men’s Morris*, si trovano in effetti tante versioni in giro per il mondo, sia più grandi (come il Morabaraba sudafricano, con le diagonali e 12 pedine a testa) sia più piccole, fino a scendere a un tavoliere  $3 \times 3$ , con le diagonali e 4 pedine a testa (come l’Achi del Ghana) o 3 pedine a testa (come il Tapatan delle Filippine), oppure addirittura senza le diagonali e 3 sole pedine a testa (come quello – probabilmente assai popolare tra gli antichi Romani, insieme coi tavolieri più grandi – menzionato dal poeta latino Ovidio nella sua *Ars amatoria*).

Un breve cenno a questi ultimi: l’Achi giocato dagli Ashanti, bellico popolo del golfo di Guinea, ha lo stesso tavoliere del Tapatan (a sinistra nella figura qui sotto).



Lo scopo è comunque quello di fare tris, in orizzontale o verticale o diagonale, muovendo a turno le proprie pedine lungo i segmenti tracciati, dopo averle disposte, sempre a turno, negli incroci evidenziati (fermo restando che una pedina deve occupare un posto libero e non può saltare); in entrambi i casi, con 4 o 3 pedine rispettivamente, ha una strategia vincente il primo giocatore.

Nell’Achi, tuttavia, le cose si possono complicare se il giocatore che fa un tris non vince, ma toglie una delle pedine avversarie e la partita prosegue, dando al giocatore rimasto in svantaggio materiale la facoltà di far saltare le proprie pedine in qualsiasi posto libero. (In tal caso, se il primo giocatore non commette errori e fa tris *dopo* che

il secondo ha messo in tavola la sua quarta e ultima pedina, allora il primo riesce a vincere; se invece fa tris *prima* che il secondo abbia concluso la disposizione delle proprie pedine, ed entrambi giocano bene, nessuno dei due può forzare la vittoria.) Il tavoliere a destra nella figura alla pagina precedente ha quattro segmenti in più: vi si gioca a Picaria, con 3 pedine a testa come nel Tapatan, ma qui per l'appunto la mobilità è maggiore. Si tratta di un gioco praticato dagli Zuni, una popolazione amerindia di agricoltori che vive nell'attuale Nuovo Messico; è conosciuto e giocato anche in altre parti del mondo, ad esempio in alcune regioni della Svezia. Il diverso tavoliere cambia l'esito di una partita ben giocata: recentemente è stato provato che, se nessuno dei due contendenti commette errori, nessuno dei due riesce a fare tris (U. Larsson e I. Rocha, 2016: *Eternal Picaria*, <https://arxiv.org/abs/1607.04236>).

Altri giochi da tener presenti per la loro maneggevolezza, ma di blocco anziché di allineamento, sono il Mu Torere originario della Nuova Zelanda; la Madelinette, che risale con probabilità all'Inghilterra medievale, e l'ancor più piccolo gioco cinese Pong Hau K'i (o Qi): salvo errori, sono tutti “eterni”! Della Madelinette parla pure Édouard Lucas nel già citato terzo tomo delle *Récréations mathématiques* (apparso postumo a cura dei suoi amici della Société Mathématique de France), ricco di ulteriori spunti interessanti: il ferro di cavallo, il gioco militare, tre dame contro una... Attenzione al fatto che, per alcuni giochi fra i tanti sopra nominati, le pedine devono essere disposte dai due contendenti, a turno, nella prima fase del gioco – addirittura certe varianti si limitano esclusivamente a questa fase, senza prevedere successivi spostamenti. E attenzione anche, in certi giochi come Picaria, a non cadere in *loop*! Preventivamente, è opportuna una ricerca per sapere se e come il gioco prescelto sia stato risolto. Ad esempio, tela classica e Fanorona sono teoricamente pari, mentre a Gomoku (o Go-Moku) vince il primo giocatore, se gioca bene, e così pure a Forza 4 e a Hex; per inciso, di Hex esistono delle varianti quadrate (anziché esagonali) più semplici da realizzare, come Quax.

Ulteriori suggerimenti sono reperibili attraverso alcuni ricchi e stimolanti siti, quali <http://superdupergames.org/> e, in italiano, <http://www.pergioco.net/>.

Infine, è pure consigliata un'indagine sulle varianti a due giocatori delle torri di Hanoi. Nel numero di dicembre 1976 della rivista inglese *Games and Puzzles*, Harry Wellerton descrive una di queste, che si gioca su una fila di sette pioli: alle due estremità ci sono due torri di cinque dischi ciascuna, una bianca e una nera; un disco non può mai essere sovrapposto a uno più piccolo, ma il colore non ha importanza; se un disco arriva al piolo di partenza dell'avversario non può più muoversi; vince chi riesce a ricostruire la propria torre sul piolo di partenza dell'avversario.

Un'altra variante con cinque pioli e torri di quattro dischi è stata ideata nel 2001 dall'olandese Corné van Moorsel ed è nota con diversi nomi: 5th Year Cwali, Stapel e Dutch Mountains.

**Parole chiave:** Awari, spazio degli stati legali, visita in *post-order* (dell'albero di gioco, entro una data profondità), varianti principali, potature, scacchi.

## 11. Uno dei due vince sempre!

In quest'ultimo capitolo proponiamo alcuni altri giochi – di una particolare sottocategoria di quelli analizzati nei due capitoli precedenti – al duplice scopo di fare ancora qualche considerazione, che speriamo possa accrescere l'interesse del lettore, e di fornire quesiti utili all'allenamento in vista di future gare di informatica, qualora vi fossero posti problemi sul tema.

Si tratta di esempi di giochi *combinatori imparziali*, che sono sempre strettamente determinati; due sono gli ulteriori requisiti che soddisfano:

- non v’è distinzione tra materiale (pezzi, pedine, gettoni, segni eccetera) dell’uno o dell’altro giocatore: in altre parole, il giocatore di turno ha le *stesse identiche* possibilità di scelta della mossa da fare che avrebbe il suo avversario, se toccasse a lui muovere;
- non sussiste possibilità di pareggio, e pertanto delle due l’una: o ha una strategia vincente chi inizia o ha una strategia vincente chi risponde.

### Babylone.

Questo gioco, grazioso ma soltanto apparentemente più complesso del filetto, fu ideato dal francese Bruno Faidutti nel 2003, sulla scia del celebre *Focus* e di sue semplificazioni o varianti, come *Quarto!*, *Pogo* e *Gobblet*.

All’inizio del gioco, 12 pedine o gettoni colorati (e precisamente 3 verdi, 3 neri, 3 rossi e 3 beige, colori che indicheremo con le lettere V, N, R e B) sono sparsi sul tavolo, a formare 12 “pile” di un solo gettone ciascuna. Il giocatore di turno deve prendere una pila e posarla sopra un’altra che abbia almeno una di queste due caratteristiche in comune con la pila spostata: lo stesso colore del gettone alla sommità oppure lo stesso numero di gettoni (cioè la stessa altezza). Quando il giocatore di turno non può muovere, perde la partita.

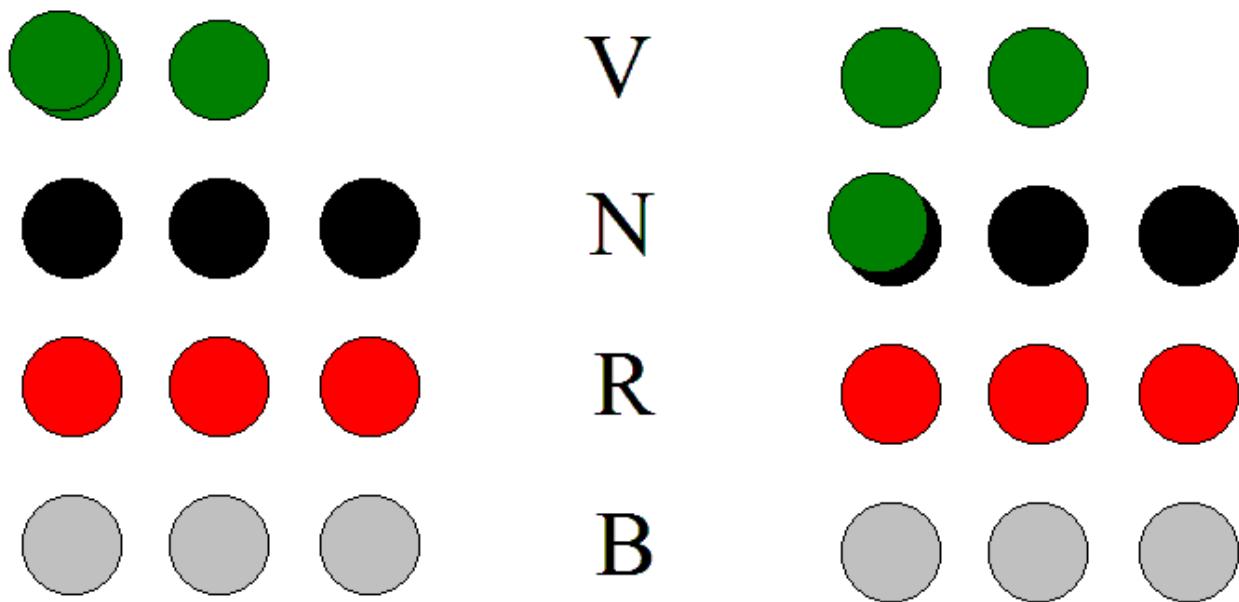
Ci sono soltanto due mosse iniziali essenzialmente differenti, per cui dopo la prima mossa si giunge in uno stato “equivalente” a uno dei due raffigurati alla pagina successiva; indicheremo con:

(V: 2, 1; N: 1, 1, 1; R: 1, 1, 1; B: 1, 1, 1) lo stato a sinistra e

(V: 2, 1, 1; N: 1, 1; R: 1, 1, 1; B: 1, 1, 1) lo stato a destra.

Vale a dire: il primo giocatore sposta un gettone sopra un altro dello stesso colore o altrimenti di colore diverso.

In entrambi i casi, il secondo giocatore può spostare la pila (verde) di altezza 2 sopra un gettone di stesso colore (facendo il viceversa si arriva allo stesso identico stato, poiché di ciascuna pila importano soltanto l’altezza e il colore del gettone alla sommità), oppure – altrimenti – può spostare un singolo gettone sopra un altro, indipendentemente dal colore.



Sicché gli stati essenzialmente differenti che si possono presentare dopo due mosse sono 10, dei quali:

- uno si può ottenere soltanto dalla posizione a sinistra nella figura:

(V: 3; N: 1, 1, 1; R: 1, 1, 1; B: 1, 1, 1)

- 5 si possono ottenere soltanto dalla posizione a destra nella figura:

(V: 3, 1; N: 1, 1; R: 1, 1, 1; B: 1, 1, 1)

(V: 2, 2, 1; N: 1, 1; R: 1, 1; B: 1, 1, 1) \*

(V: 2, 2, 1; N: 1; R: 1, 1, 1; B: 1, 1, 1)

(V: 2, 1, 1; N: 1, 1; R: 2, 1, 1; B: 1, 1)

(V: 2, 1, 1; N: 1; R: 2, 1, 1; B: 1, 1, 1)

- mentre i restanti 4 si possono ottenere da entrambe:

(V: 2, 2; N: 1, 1; R: 1, 1, 1; B: 1, 1, 1) \*

(V: 2, 1; N: 2, 1; R: 1, 1, 1; B: 1, 1, 1)

(V: 2, 1; N: 1, 1; R: 2, 1, 1; B: 1, 1, 1)

(V: 2; N: 2, 1, 1; R: 1, 1, 1; B: 1, 1, 1)  $\equiv$  (V: 2, 1, 1; N: 2; R: 1, 1, 1; B: 1, 1, 1).

L'analisi dello svolgimento di una partita può proseguire su questa falsariga, sebbene compierla a mano sia un compito improbo. Avvalendoci di un programma da noi appositamente realizzato, siamo giunti a risultati piuttosto interessanti:

- tenendo conto delle "equivalenze", gli stati raggiungibili (compreso quello iniziale) sono soltanto 688 (di cui 15 finali), quindi meno di quelli del filetto;

- sulla base del numero di mosse che occorrono per giungervi a partire dallo stato iniziale, tali stati sono distribuiti come segue:

dopo 0 mosse: 1 stato (quello iniziale);

dopo 1 mossa: 2 stati;

dopo 2 mosse: 10 stati;

dopo 3 mosse: 32 stati;

dopo 4 mosse: 76 stati;  
dopo 5 mosse: 129 stati;  
dopo 6 mosse: 164 stati;  
dopo 7 mosse: 142 stati;  
dopo 8 mosse: 88 stati, 2 dei quali sono finali, a favore del secondo giocatore:  
(V: 6; N: 3; R: 2; B: 1)  
(V: 5; N: 4; R: 2; B: 1);  
dopo 9 mosse: 35 stati, 7 dei quali sono finali, a favore del primo giocatore:  
(V: 9; N: 2; R: 1; B: 0)  
(V: 8; N: 3; R: 1; B: 0)  
(V: 7; N: 4; R: 1; B: 0)  
(V: 7; N: 3; R: 2; B: 0)  
(V: 6; N: 5; R: 1; B: 0)  
(V: 6; N: 4; R: 2; B: 0)  
(V: 5; N: 4; R: 3; B: 0);  
dopo 10 mosse: 8 stati, 5 dei quali sono finali, a favore del secondo giocatore:  
(V: 11; N: 1; R: 0; B: 0)  
(V: 10; N: 2; R: 0; B: 0)  
(V: 9; N: 3; R: 0; B: 0)  
(V: 8; N: 4; R: 0; B: 0)  
(V: 7; N: 5; R: 0; B: 0);  
dopo 11 mosse: 1 stato, finale, a favore del primo giocatore:  
(V: 12; N: 0; R: 0; B: 0).

Ciascuna partita dura quindi dalle 8 alle 11 mosse, e il nostro algoritmo *minimax* ha stabilito – fatto peraltro già scoperto nel 2004 – che *ha una strategia vincente il giocatore che muove per secondo*.

In particolare, se il primo giocatore sposta un gettone sopra un altro dello stesso colore  $x$  (stato a sinistra nella precedente figura, dove  $x = \text{verde}$ ) allora il secondo giocatore deve prendere il gettone singolo di quel colore  $x$  e metterlo sopra un altro gettone singolo (ovviamente di diverso colore).

Se invece il primo giocatore sposta un gettone di colore  $x$  sopra un altro di colore diverso  $y$  (stato a destra nella precedente figura, dove  $x = \text{verde}$  e  $y = \text{nero}$ ) allora il secondo giocatore può scegliere tra due alternative (tutte le altre deve escluderle perché portano alla sua sconfitta):

- può prendere uno dei due gettoni singoli di colore  $x$  e metterlo sopra il suo gemello (dopodiché si ottiene la stessa identica situazione del caso precedente!) oppure
- può prendere uno dei due gettoni singoli di colore  $x$  e metterlo sopra un gettone singolo di un terzo colore  $z$ , diverso sia da  $x$  sia da  $y$ .

In breve, il secondo giocatore deve raggiungere con la sua prima mossa uno dei due stati contrassegnati con l'asterisco nell'elenco dei 10 stati raggiungibili dopo due mosse, riportato alla pagina precedente. Se così non farà, metterà l'avversario in condizione di vincere.

Questo gioco può apparire più complesso del filetto perché la presenza di quattro colori, nonché la distribuzione disordinata dei gettoni sul tavolo all'inizio della partita, sembrano moltiplicare le possibili situazioni, ma in realtà non è così!

### Babylone-one.

Tuttavia il gioco si presta bene ad essere generalizzato o variato: anziché 4 colori e 3 gettoni per ciascun colore, consideriamo il caso generico di  $m$  colori, con  $n$  gettoni per colore.

Per riuscire a risolvere il gioco in tempi ragionevolmente pratici all'aumentare di questi due parametri, è necessario che il programma calcoli e memorizzi (in una lista) soltanto le mosse essenzialmente differenti che il giocatore di turno può eseguire nello stato corrente.

Per il momento, abbiamo ricavato la seguente tabella:

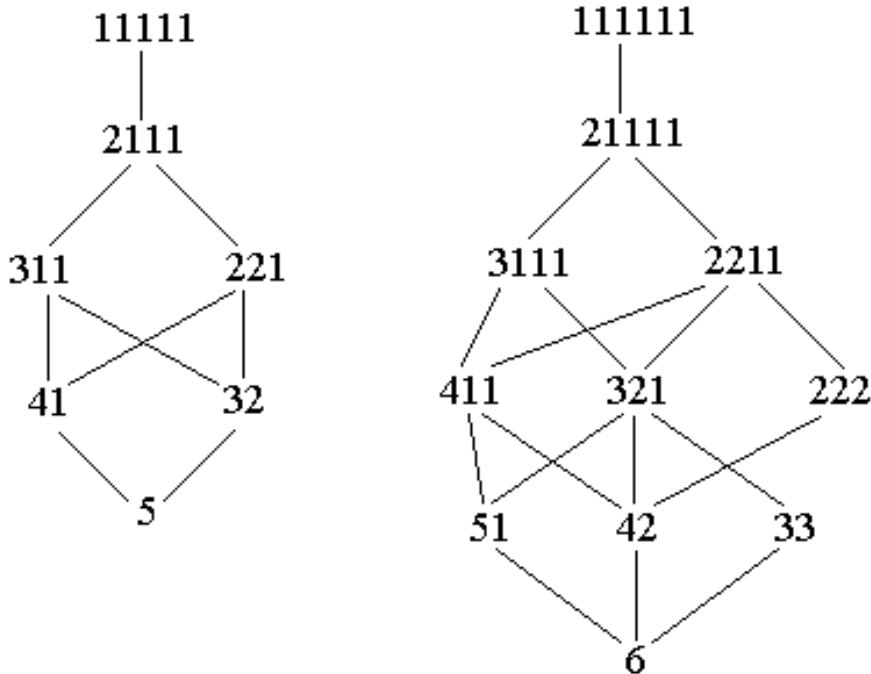
gettoni ( $n$ )	2	3	4	5	6
colori ( $m$ )					
2	1	2	2	2	2
3	2	1	1	2	
4	1	2	1		
5	2	2			
6	2				

dove 1 e 2 indicano il giocatore che dispone della strategia vincente, il primo o il secondo rispettivamente. A titolo d'esempio, con 3 colori, 4 gettoni per colore, vince il primo giocatore, e deve iniziare spostando un gettone sopra un altro dello stesso colore...

Apriamo una breve digressione. Nell'ottica di cercare qualche plausibile condizione di partita patta, abbiamo appurato che, se nel gioco classico (4 colori, 3 gettoni per colore) si introducesse la regola che decreta la parità quando alla fine rimangono sul tavolo *due* pile (ovviamente non sovrapponibili, altrimenti vincerebbe il giocatore di turno), allora il gioco perfetto sarebbe alla pari, e il primo giocatore potrebbe iniziare con una mossa a suo piacere. Considerando invece la versione con i numeri invertiti (3 colori, 4 gettoni per colore), se si assumesse la parità qualora alla fine rimanga *una sola* pila, allora il gioco sarebbe ancora alla pari, ma il primo giocatore avrebbe l'obbligo di iniziare spostando un gettone sopra un altro dello stesso colore, pena lasciare al secondo la possibilità di vittoria...

Torniamo alla nostra tabella, nella quale – come si può notare – mancano la riga 1 e la colonna 1. La riga 1, relativa al caso di un solo colore, è facile da esaminare; le pile possono comunque essere sovrapposte, indipendentemente dalla loro altezza, e l'esito della partita è determinato dalla parità del numero di gettoni  $n$ : infatti, se  $n$  è pari vince il primo giocatore, se  $n$  è dispari (anche banalmente 1) vince il secondo.

Il motivo è presto detto: per arrivare allo stato finale, dove immancabilmente sul tavolo sarà formata una pila di altezza  $n$ , occorrono esattamente  $n - 1$  mosse, a prescindere dall'ordine in cui le pile degli stati intermedi vengono via via composte. In figura sono esemplificati i casi con  $n = 5$  e  $n = 6$ , da cui già emergono l'incompletezza del reticolo e le asimmetrie.



Dunque questo gioco... non è un vero gioco! *Sembra* che i due contendenti disputino una partita, mentre in realtà – qualunque sia la scelta fatta di volta in volta dal giocatore di turno – l'esito della partita è stabilito a priori: e proprio questa, in effetti, è la caratteristica da notare in questo “gioco”!

La colonna 1 della tabella, relativa al caso di un solo gettone per ciascuno degli  $m$  colori, non è più interessante, se non in apparenza e per una curiosa circostanza, che a tempo debito sarà svelata.

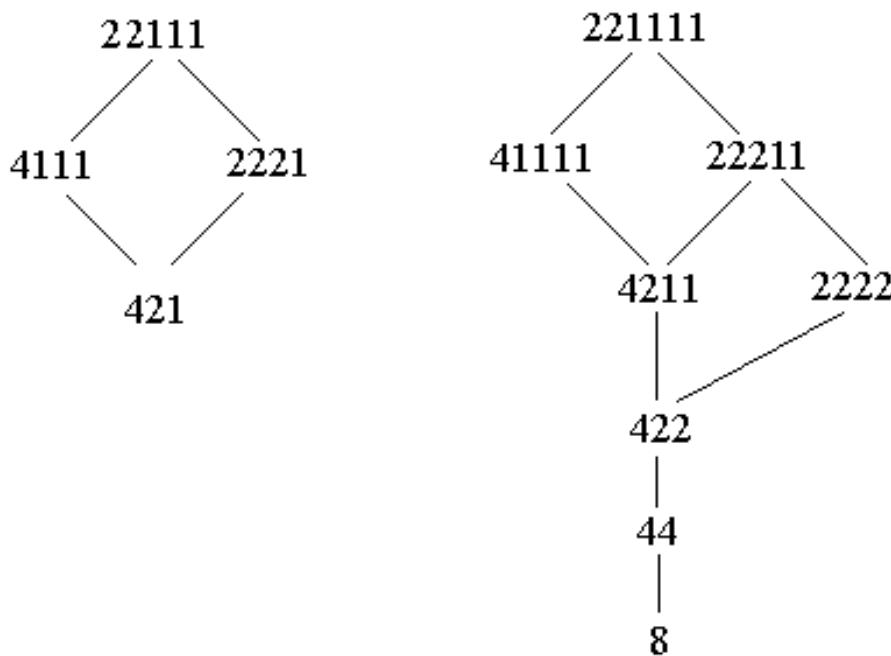
Questa volta le pile possono essere sovrapposte se e soltanto se hanno la *stessa altezza*; tanto vale allora, per non confondere le idee, prendere  $m$  gettoni di un unico colore e lasciare questa sola regola per la sovrapposizione delle pile, cosicché questa versione potrebbe essere chiamata *Babylone-one*!

Al variare di  $m$ , come si può decidere qual è il giocatore che ha la strategia vincente e quante pile, e di quali altezze, potranno rimanere al termine di una partita?

**Soluzione.** Si tratta ancora di un *falso gioco* in cui entrambi i giocatori sono *dummy*, sebbene sia un po' più difficile convincersene! In effetti, per ogni  $m$  fissato, tutte le partite che si possono giocare finiscono nello stesso stato (e con lo stesso numero di mosse), e dunque con la vittoria dello stesso giocatore.

Per ricavare una legge in funzione di  $m$ , occorre osservare che, al termine di una qualunque partita, sul tavolo ci sarà una pila di  $k$  gettoni per ogni addendo  $k$  nella rappresentazione di  $m$  come somma di potenze di 2.

Ad esempio, per  $m = 7$  e  $m = 8$  lo svolgimento delle possibili partite è schematizzato in figura (le prime due mosse sono omesse, in quanto obbligate).



Notiamo che, togliendo un gettone in tutti i nodi del grafo a sinistra, si ottiene il grafo relativo al caso  $m = 6$ ; analogamente, aggiungendo un gettone in tutti i nodi del grafo a destra, si ottiene il grafo relativo al caso  $m = 9$ : il gettone aggiunto, infatti, non aumenta il numero delle possibili mosse.

Dunque: per  $m = 6$  e  $m = 7$  le mosse sono 4 (e vince il secondo giocatore), mentre per  $m = 8$  e  $m = 9$  le mosse sono 7 (e vince il primo giocatore). Se disegniamo il grafo per  $m = 10$ , ci accorgiamo che alla fine rimangono una pila di 8 gettoni e una pila di 2, e per arrivarcì occorrono 8 mosse...

Ricordando che per formare una pila di altezza  $k$  occorrono esattamente  $k - 1$  mosse, il problema è risolto: basta scrivere il numero  $m$  in notazione binaria e pesare ciascuna cifra 1 con un'unità in meno della corrispondente potenza di 2. (Notiamo che allora il bit meno significativo può essere ignorato: una pila di altezza 1 è fatta in 0 mosse!) Si ottiene così il numero di mosse richieste precisamente da una “partita”: se questo è pari vince comunque il secondo giocatore, altrimenti vince comunque il primo.

A titolo d'esempio, per  $m = 25 = (11001)_2 = 2^4 + 2^3 + 2^0 = 16 + 8 + 1$ , alla fine rimarranno tre pile di altezza 16, 8 e 1, rispettivamente, e questo evento accadrà dopo  $(16 - 1) + (8 - 1) + (1 - 1) = 22$  mosse, e dunque vincerà il secondo giocatore.

Per  $m = 26 = (11010)_2 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2$ , alla fine rimarranno tre pile di altezza 16, 8 e 2, rispettivamente, e ciò accadrà dopo  $(16 - 1) + (8 - 1) + (2 - 1) = 23$  mosse, e dunque vincerà il primo giocatore. Stesso numero di mosse – e stesso vincitore – anche per  $m = 27$ , visto che l'ultimo bit non influisce.

Partendo da  $m = 1$ , la successione del vincitore è la seguente (qui è scritta fino a  $m = 50$ ):

2, 1, 1, 1, 1, 2, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, 2, 2, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 2, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, ...

Sarà, prima o poi, periodica? Prima di procedere nella lettura, esaminiamola attentamente: non ci ricorda qualcosa di già visto nel primo capitolo di questo libro?

L'aspetto rilevante della questione è nascosto proprio in questa successione. Se consideriamo anche il caso banalissimo dove  $m = 0$  (con nessun gettone è ovvio che vinca il secondo giocatore, dato che il primo non può muovere!) e ricordiamo quanto detto poc'anzi su un  $m$  pari e il suo successore, notiamo che la successione è formata da *coppie* di 2 e di 1. Sostituendo ciascuna coppia di 2 con un unico 0 e ciascuna coppia di 1 con un unico 1, ritroviamo l'onnipresente successione di Prouhet-Thue-Morse che, come sappiamo, non è periodica!

Concludiamo allora osservando che, se ci interessa sapere soltanto chi vincerà la partita, è sufficiente considerare il valore di  $m$  in notazione binaria, scartare la cifra meno significativa e contare le cifre 1: se queste sono dispari vincerà sicuramente il primo giocatore, se sono pari il secondo, indipendentemente dalla mossa scelta di volta in volta dal giocatore di turno.

## Il gioco di Grundy.

Si tratta di un caso semplice e classico: questa volta si parte con una pila di  $n$  gettoni già formata; il giocatore di turno deve spezzare una pila a sua scelta (alla prima mossa la scelta è ovviamente obbligata, poiché vi è una sola pila) in due nuove pile di *diversa* altezza. Il gioco finisce quando ciò non sia più possibile, ossia quando sul tavolo rimangano soltanto pile di altezza 1 o 2: in questa configurazione, il giocatore di turno non può più muovere e perde.

Consigliamo ai nostri lettori di costruire il grafo di gioco per alcuni valori di  $n$  (ad esempio 5, 6, 7, ...), osservando diversità e analogie con gli esempi sopra discussi. Questo non è un falso gioco! E per giunta non sono note le caratteristiche (quali l'eventuale periodicità) della successione che stabilisce, per ogni  $n$ , quale dei due giocatori abbia la strategia vincente, malgrado l'analisi si sia spinta molto avanti...

## Il gioco di Euclide.

Questo gioco fu introdotto da A. J. Cole e A. J. T. Davie nel 1969 (*A game based on the Euclidean algorithm and a winning strategy for it*, Mathematical Gazette, Vol. 53, No. 386, pp. 354-357), secondo regole assai semplici.

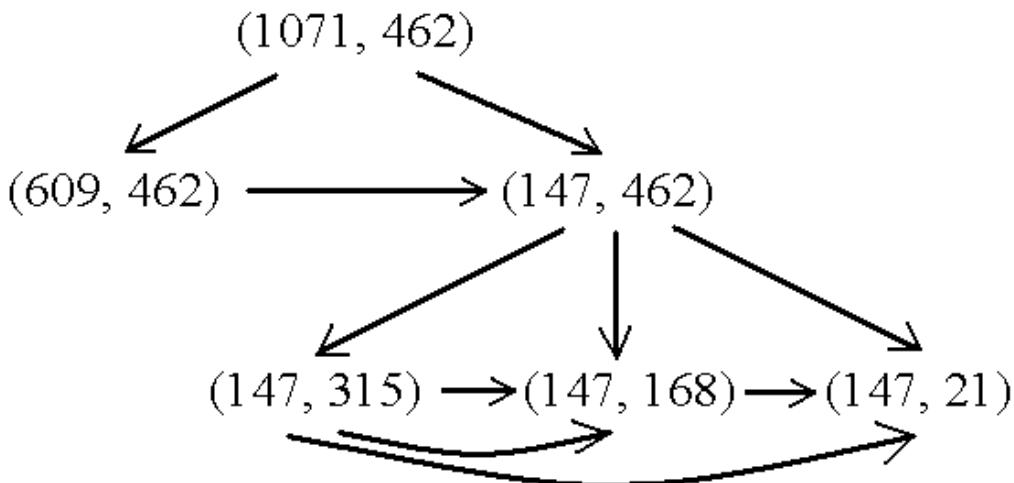
I due giocatori partono da una coppia (non ordinata) di numeri interi positivi  $(x, y)$ . Supponiamo che i due numeri siano diversi e che, ad esempio, sia  $x > y$ . Allora il primo giocatore deve togliere da  $x$  un multiplo intero positivo di  $y$ ,  $my$  (con  $m \geq 1$ ), pur di lasciare una quantità  $x - my \geq 0$ . Alla stessa maniera si dovrà comportare, di volta in volta, il giocatore di turno con i due numeri rimasti, finché sia possibile procedere. Vince il giocatore che riesce a ottenere una coppia che abbia uno dei due numeri uguale a zero.

Ovviamente, se si iniziasse da una coppia in cui uno dei due numeri è un multiplo dell'altro, il primo giocatore vincerebbe subito alla prima mossa; ad esempio, partendo da  $(6, 6)$  o da  $(6, 12)$  o da  $(6, 18)$ ... egli lascerebbe  $(6, 0)$ .

Vediamo un esempio di partita; per brevità chiamiamo  $A$  e  $B$  i due giocatori. Partendo dalla coppia  $(1071, 462)$ , il giocatore  $A$  vince in questo modo: alla prima mossa toglie una sola volta il  $462$  dal  $1071$ , ottenendo  $(609, 462)$ ;  $B$  è obbligato a toglierlo ancora una volta, passando a  $(147, 462)$ ; ora  $A$ , per vincere, deve togliere due volte il  $147$  dal  $462$ , ottenendo  $(147, 168)$ ;  $B$  è obbligato a passare a  $(147, 21)$ , e quindi  $A$  vince con  $(0, 21)$ , in quanto  $7 \cdot 21 = 147$ .

È chiaro che se alla prima mossa  $A$  avesse tolto due volte il  $462$  dal  $1071$ , ottenendo  $(147, 462)$ , adesso sarebbe  $B$  a vincere, passando a  $(147, 168)$ .

Il grafo in figura illustra tutti i possibili svolgimenti della partita fino alla coppia  $(147, 21)$ , che è “perdente” per chi la raggiunge.



Proponiamo un quesito: partendo dalla coppia  $(495, 364)$ , il giocatore  $B$  (che muove per secondo) vince, purché non commetta errori... Come si svolgerà la partita? [Suggerimento: ciascuno dei due giocatori farà quattro mosse.]

**Soluzione.** Partendo dalla coppia (495, 364), il giocatore *A* avrà sempre la mossa obbligata: alla prima, deve passare a (131, 364). Il giocatore *B* deve sottrarre due volte il 131 dal 364, ottenendo (131, 102); *A* muove (29, 102); ora *B*, delle tre possibilità che ha, deve scegliere (29, 44); *A* muove (29, 15); a questo punto anche *B* non ha altra possibilità che (14, 15); *A* deve muovere (14, 1) e quindi *B* vince lasciando (0, 1).

Perché è stato chiamato “il gioco di Euclide”? È stato ispirato dal procedimento descritto da Euclide per calcolare il *massimo comun divisore*: il numero che, alla fine del gioco, rimane in coppia con lo zero è proprio il massimo comun divisore della coppia di numeri iniziale (e di ciascuna delle coppie ottenute successivamente). Come scrisse Donald E. Knuth, il metodo euclideo costituisce, tra quelli conosciuti, il più antico algoritmo non banale sopravvissuto fino ai nostri giorni: ha almeno 23 secoli! In versione “ricorsiva in coda” (*tail recursive*, che non necessiterebbe dello *stack* di sistema):

```
unsigned int mcd (unsigned int a, unsigned int b) {
    if (b == 0) return a;
    return mcd(b, a%b); // in C, % è l'operatore "modulo"
}
```

Qui non importa quale sia il maggiore dei due argomenti, né tantomeno se siano diversi; una sola osservazione: sulla coppia (0, 0) questa funzione restituisce zero (ed è il solo caso in cui fornisce tale risultato), mentre il massimo comun divisore non è definito su (0, 0).

Tornando al gioco proposto: si può dimostrare che il primo giocatore ha una strategia vincente quando i due numeri sono uguali (è uno dei casi banali, in cui vince con una sola mossa) oppure quando il rapporto tra il numero più grande  $x$  e quello più piccolo  $y$  è maggiore del *rapporto aureo*  $(1 + \sqrt{5}) / 2$  (eventualità che comprende anche gli altri casi banali, in cui il rapporto è un intero  $\geq 2$ ); altrimenti, è il giocatore che muove per secondo ad avere una strategia vincente.

Inoltre, ogni volta che vi sono più mosse possibili per un giocatore, l’analisi del gioco può limitarsi a considerare due mosse soltanto: togliere da  $x$  la quantità  $My$  oppure la quantità  $(M-1)y$ , dove  $My$  è il massimo multiplo di  $y$  tale che  $x - My \geq 0$ . Ad ogni passo, l’algoritmo di Euclide sottrae invece il più possibile (ovvero  $My$ ), sì da giungere al massimo comun divisore nel minor numero di passi.

Se  $N$  è il numero di passi richiesto dall’algoritmo di Euclide, si può provare per induzione (Gabriel Lamé, 1844) che i due numeri più piccoli per cui ciò è vero sono  $F_{N+1}$  e  $F_{N+2}$  della famosa *successione di Fibonacci* ( $F_0 = 0$ ,  $F_1 = 1$ ,  $F_{n+2} = F_{n+1} + F_n$  per ogni naturale  $n$ ).

Questo è soltanto uno dei casi in cui, nel nostro gioco, tutte le mosse sono obbligate – fino alla penultima mossa compresa, dopodiché il giocatore di turno sarebbe autolesionista se non scegliesse quella vincente!

Quindi, partendo dalla coppia  $(F_{N+2}, F_{N+1})$  vince il giocatore A (lasciando alla fine la coppia  $(0, 1)$ : i due numeri di partenza sono infatti primi fra loro) se e soltanto se N è dispari. In tal caso il rapporto  $F_{N+2} / F_{N+1}$  è maggiore del rapporto aureo, mentre se N è pari, esso è minore; e più grande è N, più tale rapporto si avvicina a quello aureo, per eccesso o per difetto in modo alterno.

La prova di Gabriel Lamé alla quale si è accennato, insieme col suo corollario, che stabilisce il massimo numero di passi per l'algoritmo di Euclide in 5 volte il numero di cifre decimali del più piccolo tra i due numeri di partenza, segnò l'inizio della teoria della complessità computazionale, nonché la prima applicazione pratica dei numeri di Fibonacci.

Infine, segnaliamo una variante del gioco di Euclide, che pure è stata studiata più recentemente (Jerrold W. Grossman, 1997: *A nim-type game, problem #1537*, Mathematics Magazine, Vol. 70, p. 382): la vittoria è assegnata al giocatore che riesce a ottenere una coppia di numeri uguali...

In questa ipotesi, chi vincerebbe nei due casi proposti (l'esempio contenuto nel testo e il quesito)?

A chi voglia provare ancora a giocare, verificando l'esito della partita sulla base delle nostre spiegazioni, e magari scoprendo qualche situazione particolare, suggeriamo alcune coppie di numeri da cui partire:  $(90, 168)$ ,  $(84, 55)$ ,  $(35, 78)$ ,  $(28, 45)$ ,  $(5568, 864)$ .

## MIN.

Un altro gioco della stessa famiglia è stato proposto nel 2010 da Grant Cairns e Nhan Bao Ho (*MIN, a combinatorial game having a connection with prime numbers*, Integers, Vol. 10, No. 6, pp. 765-770). Dai suoi autori chiamato MIN (abbreviazione del vocabolo inglese e latino *minus*, in italiano “meno”), una mossa consiste nel sottrarre dal numero più grande un numero positivo minore o uguale al numero più piccolo della coppia. Vince chi riesce a ridurre a zero uno dei due numeri della coppia.

A questo punto, siamo certi che i nostri lettori sapranno sviluppare da soli esempi e considerazioni interessanti su questo gioco... e forse anche a scoprire come vi sono coinvolti i numeri primi!

## I giochi del (non) ritorno e della (non) ripetizione.

Altri giochi – sempre combinatori imparziali e recentemente studiati – hanno come tavoliere un grafo *orientato* (o *diretto*: ciascun arco ha un verso di percorrenza) in cui vi sia, al più, un arco da un nodo a un altro (o anche da un nodo a sé stesso).

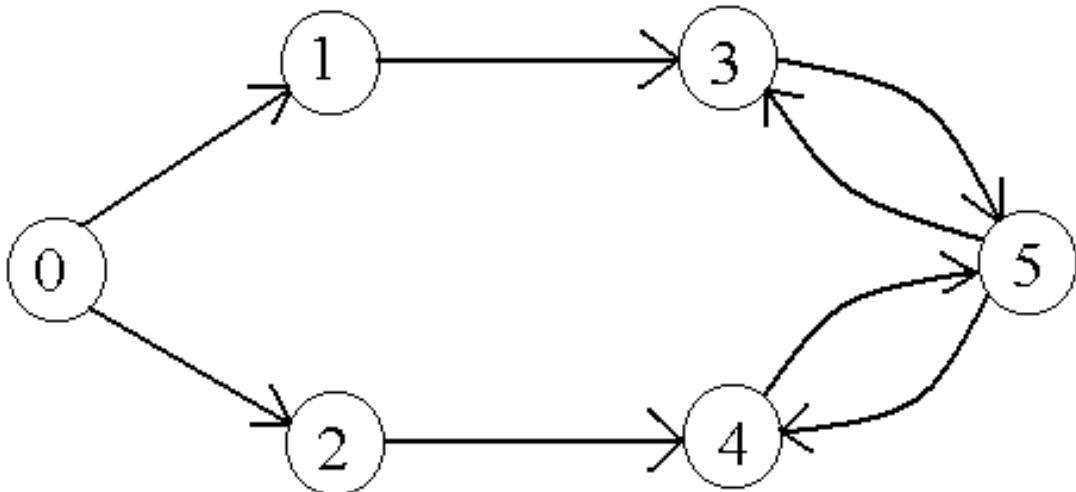
Posto un gettone sul nodo scelto come punto di partenza, ognuno dei due giocatori lo dovrà muovere, al proprio turno, in uno dei nodi adiacenti, ossia raggiungibili percorrendo un solo arco che si diparte dal nodo attualmente occupato dal gettone.

Le possibilità sono:

- 1) perde chi muove il gettone in un nodo (compreso quello di partenza) già occupato precedentemente dal gettone (*gioco del non ritorno*);
- 2) vince chi muove il gettone in un nodo (compreso quello di partenza) già occupato precedentemente dal gettone (*gioco del ritorno*);
- 3) perde chi percorre un arco già percorso in precedenza dall'uno o dall'altro dei due giocatori (*gioco della non ripetizione*);
- 4) vince chi percorre un arco già percorso in precedenza dall'uno o dall'altro dei due giocatori (*gioco della ripetizione*).

Può darsi che nessuno dei due giocatori disponga di una strategia vincente (anche quando il grafo contiene cicli); ciò accade, certamente, se il grafo è *aciclico*, cioè privo di cicli (per inciso, forma un ciclo anche un *cappio*, ossia un arco da un nodo al nodo stesso): in tal caso non può esservi alcun vincitore!

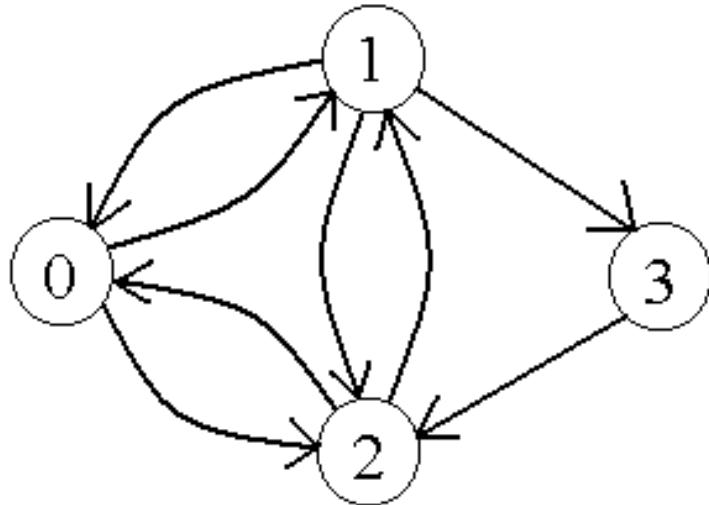
Nell'esempio di figura, partendo dal nodo 0, il giocatore che muove per secondo ha una strategia vincente in tutti e quattro i giochi, come i nostri lettori potranno facilmente constatare.



Nella gara *Kangourou dell'Informatica* di marzo 2011 fu proposta a tutti i concorrenti un'istanza dell'ultimo gioco, quello della *ripetizione*, che appresso riformuliamo a titolo d'esempio.

Supponendo di partire col gettone nel nodo 0 del grafo disegnato qui sotto, come si svolgerà la più lunga partita “senza errori”?

(Questo vuol dire che se il giocatore di turno ha una strategia vincente, allora sceglierà la mossa che gli permetterà di vincere nel modo più rapido; in caso contrario, cercherà di ritardare il più possibile la propria sconfitta.)

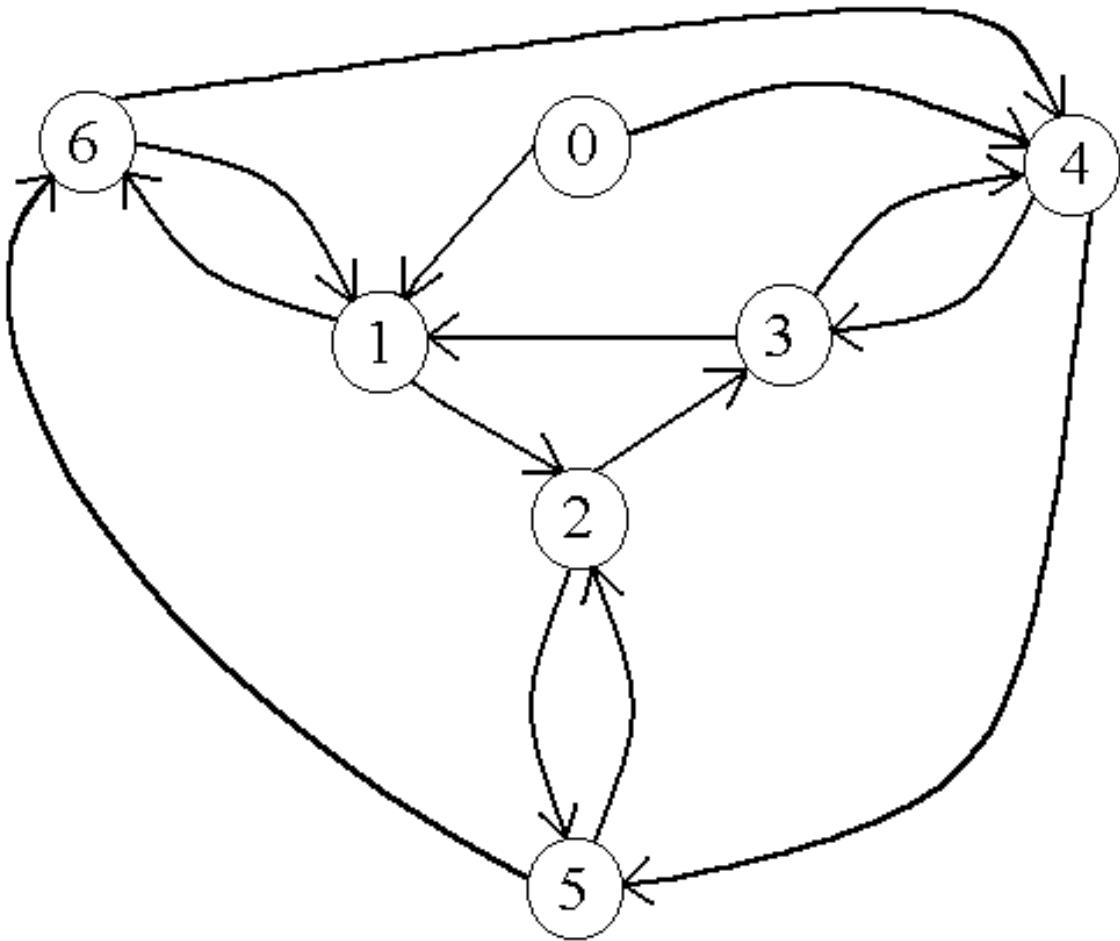


**Soluzione.** Vince il giocatore che muove per primo – chiamiamolo  $A$  – percorrendo l’arco  $(0, 2)$ , ossia spostando il gettone dal nodo 0 al nodo 2. Se il secondo giocatore,  $B$ , percorre l’arco  $(2, 0)$  allora  $A$  vince ripetendo  $(0, 2)$ ; se invece  $B$  percorre  $(2, 1)$  allora  $A$  deve scegliere  $(1, 3)$ , al che  $B$  è costretto a tornare nel nodo 2 e quindi  $A$  vince con  $(2, 1)$ .

Se all’inizio  $A$  scegliesse  $(0, 1)$ ,  $B$  risponderebbe con  $(1, 2)$  per poi vincere alla mossa successiva.

Agli altri tre giochi, invece, vincerebbe  $B$ , indipendentemente dalla scelta iniziale di  $A$  tra l’arco  $(0, 1)$  e l’arco  $(0, 2)$ ; nel caso del gioco del ritorno la verifica di questo fatto è immediata...

Proponiamo ora un’istanza del primo gioco, quello del *non ritorno*, e formuliamo il seguente quesito: sempre con partenza nel nodo 0, quante sono le partite possibili e chi vince?



**Soluzione.** Ha la strategia vincente il giocatore che muove per secondo, e le partite possibili sono 34. Per verificarlo, basta costruire l'albero che si ottiene “svolgendo”, “srotolando” il grafo (operazione che in inglese si chiama *unfolding*) a partire dalla radice 0, fino a che su ciascun ramo si arriva a ripetere un nodo.

Si scopre così che le due partite più brevi sono costituite da tre archi ciascuna: (0, 1), (1, 6), (6, 1) e (0, 4), (4, 3), (3, 4), in cui il primo giocatore perde, mentre le partite più lunghe sono dodici, costituite da sette archi ciascuna: dopo che tutti e sette i nodi sono stati toccati, inevitabilmente il primo giocatore è costretto a ripassare in un nodo già visitato (nella fattispecie, il nodo 1 o il nodo 4).

Per tutti e quattro i giochi, dato un grafo qualsiasi e scelto arbitrariamente un nodo di partenza, stabilire se uno dei due giocatori abbia una strategia vincente, e – in caso affermativo – quale sia dei due, è un problema NP-arduo: a questo proposito si può vedere lo studio di Jack R. Edmonds e Vladimir A. Gurvich *Games of no return* (RUTCOR Research Report, RRR-4-2010, Rutgers University) del febbraio 2010.

## Chi arriva a un euro?

Concludiamo la nostra breve rassegna di giochi tra due avversari illustrando un paio di casi classici, che nuovamente ci ricordano come talvolta sia immediato stabilire quale giocatore abbia la strategia vincente e in quale modo debba procedere, così come abbiamo visto per il gioco di Euclide. Introduciamo il primo con un quesito.

Aldo e Beppe hanno molte monetine da un centesimo, e pensano questo gioco: Aldo inizia a mettere sul tavolo una monetina, dopodiché, a turno, i due ragazzi devono aggiungere da un minimo di 1 a un massimo di 10 centesimi ogni volta; chi arriva a un euro vince l'intera somma. Chi dei due vincerà?

**Soluzione.** Vince Aldo, aggiungendo ogni volta il complemento a 11 del numero di monetine appena messe da Beppe; ad esempio, se Beppe mette 4 centesimi Aldo ne aggiunge 7, se poi Beppe ne mette 9 Aldo ne aggiunge 2...

Assumendo come parametri il numero  $N$  di monete da raggiungere e il numero massimo  $M$  di monete che il giocatore di turno può aggiungere (nel nostro caso 100 e 10, rispettivamente), possiamo affermare che Aldo ha la strategia vincente se  $N = 1 + (M+1) \cdot q$ , per un qualche  $q$  intero positivo, altrimenti la strategia vincente è di Beppe.

Del tutto analogo è uno dei tanti giochi che si fanno (o almeno si facevano una volta) con i fiammiferi: da una fila di  $N$  fiammiferi, i due giocatori prelevano a turno da 1 a  $M$  fiammiferi; perde chi toglie l'ultimo.

Molti anni fa, agli studenti che incominciavano a imparare l'uso di un linguaggio di programmazione, proponevo di scrivere un programmino che, fatto il conto, doveva decidere se iniziare il gioco oppure farlo iniziare all'utente, sì da arrivare a vincere sempre: ovviamente non ci si divertiva più di tanto, se non quando qualcuno di loro lo regalava ai fratelli più piccoli, ma serviva piuttosto bene come esempio d'impiego dell'istruzione condizionale e di un ciclo il cui corpo di istruzioni doveva essere eseguito un certo numero (precalcolabile) di volte.

Il "conto" che deve fare il programma consiste semplicemente in una divisione: tolto da  $N$  il fiammifero da lasciare al perdente, si deve calcolare quanti gruppi di  $M+1$  fiammiferi si riescono a formare e, soprattutto, quanti fiammiferi avanzano: se ne avanza qualcuno, allora ha la strategia vincente chi inizia la partita, il quale alla prima mossa dovrà eliminare proprio quei fiammiferi che costituiscono tale resto... Potete leggere il codice in C della procedura alla pagina seguente. Questo è dunque un gioco facilmente risolvibile: basta fare una divisione per sapere chi vincerà!

Lasciamo scoprire a voi come determinare la strategia vincente nel caso in cui si stabilisca che vinca chi riesce a prendere l'ultimo fiammifero (ovviamente anche non da solo, bensì con eventuali altri fiammiferi, purché in tutto non siano più di  $M$ ). [Suggerimento: questa volta si dovrà calcolare il resto della divisione intera  $N / (M+1)$ .]

Se le possibilità di presa sono limitate a un solo fiammifero oppure a due adiacenti (*Kayles*) e chi prende l'ultimo vince, allora ha sempre la strategia vincente chi inizia.

```

void gioco_dei_fiammiferi (int N, int M) {
    // Precondizione: 1 < M < N
    int r;
    r = (N-1)%(M+1);
    // r è il resto della divisione intera (N-1) / (M+1)
    // Il corpo del ciclo while, più sotto, sarà eseguito
    // esattamente q volte, dove q è il quoziente della divisione
    // intera (N-1) / (M+1), e quindi: q*(M+1) + r == N-1
    if (r == 0) printf("Inizia tu!\n");
    else {
        printf("Inizio io, e tolgo %d fiammiferi.\n", r);
        N = N-r;
        printf("Fiammiferi rimasti: %d\n", N);
    }
    while (N > 1) {
        printf("Quanti fiammiferi togli? ");
        scanf("%d", &r);
        // supponiamo che l'utente non imbrogli
        // e che quindi sia 1 <= r <= M
        printf("Io tolgo %d fiammiferi.\n", M+1-r);
        N = N-M-1;
        printf("Fiammiferi rimasti: %d\n", N);
    }
    printf("Rimane un solo fiammifero: ho vinto io!\n");
}

```

## NIM.

Il più celebre dei giochi con i fiammiferi è il *Nim*; nella sua versione originale, si parte con  $N$  file (o insiemi) di fiammiferi, l' $i$ -esima fila è costituita da  $F_i$  ( $> 0$ ) fiammiferi,  $i = 1, \dots, N$ . Il giocatore di turno deve togliere da una fila (non vuota) a sua scelta un numero qualsiasi di fiammiferi, al minimo uno, al massimo tutti. Qui di norma vince (perde è la variante) chi toglie l'ultimo fiammifero.

La strategia vincente si basa sull'operazione di OR esclusivo (EXOR) fatta bit a bit sui numeri  $F_i$  espressi in binario, e consiste nel lasciare l'avversario in una situazione in cui tutti questi EXOR diano risultato 0.

Ad esempio, se  $N = 3$ ,  $F_1 = 3 = (011)_2$ ,  $F_2 = 4 = (100)_2$ ,  $F_3 = 5 = (101)_2$ , l'EXOR bit a bit dà 010 (dunque non tutti zeri) e allora il giocatore di turno potrà vincere, perché gli sarà comunque possibile lasciare l'avversario in uno stato in cui tale operazione darà tutti zeri. Nel nostro caso l'unica mossa che porta a questo risultato è togliere due fiammiferi dalla fila di tre: l'EXOR bit a bit di 001, 100 e 101 dà 000, ed è facile verificare che alla mossa successiva l'avversario non avrà un'analogia possibilità!

Se scriviamo i numeri in colonna, l'operazione ora descritta corrisponde a un controllo di parità sulle colonne di bit: più precisamente, il risultato è formato dalla sequenza degli *even parity bit*, che rendono pari il numero di 1 in ciascuna colonna. (Ovvero, se preferite, calcolate le somme in binario, ignorando tutti i riporti.)

Vediamo un altro esempio, con  $N = 5$ , che ci fa pure capire che le mosse “giuste” possono essere più d'una.

$F_1 = 11$	0	1	0	1	1
$F_2 = 13$	0	1	1	0	1
$F_3 = 18$	1	0	0	1	0
$F_4 = 21$	1	0	1	0	1
$F_5 = 24$	1	1	0	0	0
<hr/>					
<i>even parity bit</i>	1	1	0	0	1

Dobbiamo *togliere* dei fiammiferi: quindi escludiamo subito le prime due file, per portare a 0 il primo bit di parità. In effetti, qui possiamo azzerare tutti i bit di parità in tre diversi modi:

- togliere 7 fiammiferi dalla terza fila, dove ne resterebbero  $11 = (01011)_2$ ;
- togliere 9 fiammiferi dalla quarta fila, dove ne resterebbero  $12 = (01100)_2$ ;
- togliere 23 fiammiferi dalla quinta fila, dove ne resterebbe uno solo...

Perché mai l'operazione eseguita ci permette di individuare le mosse vincenti?

Fu il matematico statunitense Charles L. Bouton, nel 1901, a dare il nome “Nim” a questo gioco (forse da un verbo dell’antico inglese e del tedesco, che significa “prendere”, o forse per il fatto che, capovolgendo la parola NIM, si ottiene WIN, “vincere”) e ad analizzarlo compiutamente, dimostrando che:

- i) partendo da una situazione con tutti i bit di parità a 0, la prima mossa porterà necessariamente a una situazione che non può conservare tale proprietà;
- ii) partendo invece da una situazione con i bit di parità non tutti a 0, esiste sempre almeno una mossa che azzeri tutti i bit di parità.

Corollario: poiché, banalmente, nella situazione di fine partita tutti i bit di parità sono 0 (non ci sono più fiammiferi, e chi dovrebbe muovere perde), ecco spiegato perché colui che deve muovere in una situazione con i bit di parità non tutti a 0 può forzare la sconfitta dell'avversario.

Il risultato di Bouton fu esteso – e valorizzato in una visione più profonda – da un teorema, mostrato indipendentemente da Roland P. Sprague nel 1935 e da Patrick M. Grundy nel 1939, che riguarda i giochi combinatori imparziali ove chi non può più muovere perde: *tutti* questi giochi sono riconducibili al caso paradigmatico del Nim, e il suddetto teorema permette, in ogni circostanza, un’analisi completa...

Lasciamo ancora ai nostri lettori qualche situazione di gioco su cui riflettere:

- le file sono tre, rispettivamente con 6, 9 e 13 fiammiferi. Tocca prendere a voi. Qual è l'unica mossa che vi garantisce di vincere e perché ce n'è una sola?
- Aggiungendo un fiammifero alla terza fila, ottenendo così 6, 9 e 14, vi è sempre un'unica mossa vincente, ma non è più la stessa: qual è?
- Mettiamo invece due fiammiferi in più nella prima fila: 8, 9 e 13. Adesso le mosse che vi garantiscono la vittoria sono ben tre: quali sono e come fate a scoprirlle?
- Se facciamo entrambe le aggiunte, partendo quindi da 8, 9 e 14 fiammiferi, ci sono di nuovo tre mosse vincenti, diverse da tutte le precedenti!

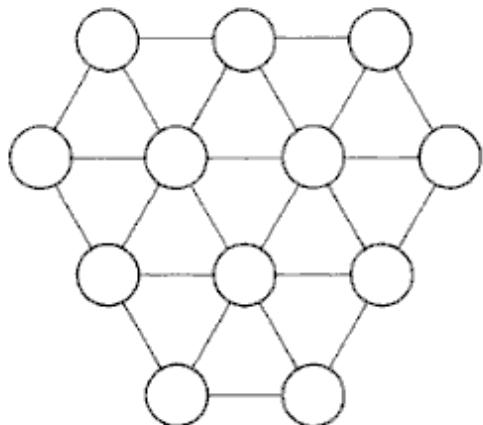
Piet Hein, il celebre fisico e poeta danese che nel 1942 inventò il gioco topologico (non imparziale) dell'Hex,<sup>29</sup> successivamente ideò alcune notevoli varianti del Nim, su tavolieri di varie forme geometriche, oggi comprese nei *Nim-like games*.

La prima di esse risale alla fine degli anni '40 e fu chiamata originariamente *Bulo* (in danese) e poi *Tac Tix* da Martin Gardner: 16 pedine sono disposte su una scacchiera  $4 \times 4$  e ciascuno dei due giocatori può prendere una o più pedine, purché adiacenti, da una riga o da una colonna; volendo generalizzare,  $n^2$  pedine su una scacchiera  $n \times n$ .

Se vince chi fa l'ultima presa, l'analisi è piuttosto facile: se  $n$  è dispari, vince il primo giocatore, prendendo la pedina centrale e poi rispondendo simmetricamente al secondo giocatore; se  $n$  è pari, vince il secondo, rispondendo da subito in maniera simmetrica rispetto al primo.

Se invece si gioca a *misère* (chi fa l'ultima presa perde), l'analisi è assai più difficile, salvo nel caso  $n=3$  in cui vince il primo prendendo la pedina centrale o una d'angolo o tutta la riga/colonna centrale; ma nel caso  $n=4$  vince il secondo...

Nel 1967 Hein propose il tavoliere qui sotto disegnato, detto *Nimbi*, dove le prese sono lecite anche in diagonale, sempre sotto la condizione di rimuovere pedine contigue, lungo la stessa direzione, in caso di presa multipla.



Aviezri S. Fraenkel e Hans Herda hanno dimostrato che qui ha la strategia vincente il secondo giocatore, sia che perda sia che vinca chi prende per ultimo (*Never Rush to Be First in Playing Nimbi*, Mathematics Magazine, Vol. 53, No. 1, gennaio 1980, pp. 21-26).

E per concludere davvero, già che abbiamo fatto trenta...

---

<sup>29</sup> Qui invito chi ancora non conoscesse Hex a fare un'indagine: si tratta di un gioco di collegamento, su un tavoliere simmetrico, in cui l'unico modo di impedire all'avversario di costruire una catena che gli permetta di vincere è quello di costruirsiene una che assicuri la propria vittoria! In virtù di queste considerazioni, e poiché aver piazzato una pedina in più rispetto all'avversario non può causare svantaggio, John F. Nash (il quale nel 1948, mentre studiava a Princeton, riscoprì Hex senza esserne a conoscenza) diede una dimostrazione (per assurdo, e non costruttiva) del fatto che dev'essere il primo giocatore ad avere una strategia vincente.

## ... Facciamo trentuno!

Proponiamo un vecchio gioco che – come ricorda Henry Dudeney nella sua prima raccolta, *The Canterbury Puzzles and Other Curious Problems*,<sup>30</sup> del 1907 – si faceva con le carte, ed era uno dei modi preferiti dai bari per mettere in atto una truffa negli ippodromi o sui treni; tuttavia, come si può già immaginare, la casualità non c’entra per niente, e non v’è neppure alcun inganno!

Il baro dispone le prime ventiquattro carte di un mazzo, dai quattro assi (ciascuno dei quali vale 1) ai quattro 6, e invita l’ignaro passante o viaggiatore a tentare la fortuna, o piuttosto l’abilità, per vedere chi per primo riesce a totalizzare 31, girando a turno una carta e sommando il suo valore a tutte quelle già girate da entrambi. Chi però “sballa”, superando 31, regala la vittoria all’avversario.

Anche senza le carte, si può giocare scrivendo quattro volte i numeri da 1 a 6 e depennando man mano quelli corrispondenti alle carte girate.

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6

Memore del calcolo da fare per arrivare a un euro con le monetine da un centesimo, qualcuno di voi potrebbe ritenere ovvia la condotta da tenere:  $6+1$  fa 7, e dividendo 31 (ora si parte da zero) per 7 il resto è 3, sicché chi inizia gira un 3 e può facilmente vincere girando ogni volta una carta “complemento a 7” di quella girata subito prima dall’avversario, realizzando così 10, 17, 24 e infine 31. Direbbe così al baro, che “baro” non è proprio per nulla: «Inizio girando un 3!»... e andrebbe clamorosamente a perdere! Perché? Chi dei due ha la strategia vincente e come deve comportarsi?  
[*Suggerimento*: in effetti vince chi inizia, ma deve cominciare girando un 5 oppure un 2 oppure un asso.]

Una speranza e un augurio: spero che la lettura di questo libro sia stata stimolante – o, almeno, non troppo noiosa – e, *nuovamente*, vi auguro di riuscire a realizzare tutti i programmi che desiderate!

**Parole chiave:** gioco combinatorio imparziale (finito), falso gioco, successione di Prouhet-Thue-Morse, quoziante e resto della divisione intera, algoritmo euclideo (per il calcolo del massimo comun divisore), successione di Fibonacci, percorsi su grafi orientati, OR esclusivo (EXOR), controllo di parità, regola di Bouton per il Nim.

---

<sup>30</sup> In rete si trova riprodotta la seconda edizione (<http://www.gutenberg.org/ebooks/27635>) pubblicata da Thomas Nelson and Sons, a Londra, nel 1919. In questo volume, così come negli altri testi “storici” menzionati nei capitoli precedenti, sono contenuti moltissimi quesiti che, ancor oggi, mettono a dura prova le nostre capacità di ragionamento!

# Appendice

Desidero arricchire questa seconda edizione con un centinaio abbondante di pagine, ove tratterò vari argomenti. Inizierò dicendo qualcosa di più sugli schemi di Sudoku, sulle tecniche per risolverli e sull'impostazione di un programma che impieghi la “forza bruta”, per poi passare al problema dell'esatta copertura di un insieme, a cui un puzzle di Sudoku può essere ricondotto. Proseguirò con i polimini e i puzzle che ne fanno uso; in particolare, mi soffermerò su alcuni puzzle con i pentamini, pur essi riconducibili a esatte coperture di insiemi. Con i pentamini sono stati ideati anche giochi per due o più contendenti: ne esamineremo i più classici... Concluderò con qualche trasformazione di piccoli quadrati latini o di schemi di Sudoku 4×4.

Prendendo poi spunto da quesiti delle gare Bebras, passerò a spiegare tipici problemi sui grafi, orientati o meno, nonché il problema della minima copertura di un insieme. Si tratta, in molti casi, di problemi per i quali non sappiamo se esistano metodi di risoluzione efficienti; illustrerò comunque degli algoritmi per risolverli esattamente oppure per trovare rapidamente una soluzione, che di solito però non è quella ottima. Rompicapi con biglie colorate serviranno a presentare questioni di fondamentale rilevanza nell'informatica, e non solo teorica, sebbene spesso non siano decidibili, in generale, per via automatica.

Avendo parlato di Awari, vedremo pure dei giochi di semina o di mietitura, da svolgere “in solitario”, tutti assai interessanti dal punto di vista computazionale e a fondo studiati. Dopodiché diremo ancora qualcosa sui giochi di strategia, come Forza Quattro, e sui giochi ove la logica abbia un ruolo predominante, come Mastermind. Giustappunto un quesito legato alla logica fornirà il pretesto per un discorso su di essa, ma intesa come paradigma di programmazione, e sulle basi di dati deduttive, con esempi brevi bensì alquanto significativi. Un successivo quesito, che coinvolge un piccolo *database*, ci permetterà di riflettere sui principî di buona progettazione di un archivio di dati e, in particolare, sul ruolo delle “chiavi”, le quali, quando devono mantenersi ordinate, richiedono l'impiego di adeguate strutture di dati, possibilmente gestite da procedure efficienti, come gli alberi rosso-neri...

La codifica dell'informazione e la compressione dei dati saranno introdotte da alcuni simpatici quesiti, ai quali seguiranno le presentazioni dei relativi algoritmi.

Infine, per concludere davvero, analizzeremo una variante della torre di Hanoi e un altro – ormai classico – problema: il ribaltamento delle frittelle!

Naturalmente, il lettore poco interessato a certi argomenti può anche tralasciare le relative pagine, senza tuttavia compromettere la comprensione delle successive, poiché tutti i paragrafi di questa appendice sono tendenzialmente autonomi.



## 1. Il Sudoku, un po' più in dettaglio.

Penso che tutti voi conosciate questo tipo di puzzle nell'usuale versione  $9\times 9$ : si tratta di completare lo schema dato, di modo che in ogni riga, in ogni colonna e in ognuno dei nove riquadri  $3\times 3$  delimitati da linee più marcate, compaiano le cifre da 1 a 9.

Nato in Giappone nel 1984 (*Su* = numero, *Doku* = solitario), ma ispirato dal *Number Place* creato dall'architetto Howard Garns e già pubblicato negli Stati Uniti dal 1979, si diffuse a livello mondiale due decenni più tardi, a partire dall'autunno del 2004, quand'ebbe inizio la sua costante presenza sul *Times* di Londra, anche grazie a un programma che sfornava sempre nuovi schemi, realizzato (nell'arco di sei anni!) da un giudice di Hong Kong, Wayne Gould, "importatore" del gioco giapponese.

Già Leonhard Euler (Eulero), nel XVIII secolo, aveva studiato i cosiddetti "quadrati latini", ove si tiene conto soltanto di righe e colonne, senza il vincolo imposto dai riquadri. I quadrati latini  $9\times 9$ , contati nel 1975 senza considerare alcuna simmetria, sono oltre  $5.5 \cdot 10^{27}$ ; un loro "piccolo" sottoinsieme sono i circa  $6.67 \cdot 10^{21}$  possibili schemi finali di Sudoku, contati nel 2005 (sequenza A107739 in OEIS, *The On-Line Encyclopedia of Integer Sequences*). Tuttavia, tenendo conto di permutazioni (delle cifre, o delle righe/colonne di stesso riquadro, o dei blocchi di righe/colonne 1-3, 4-6, 7-9) e di rotazioni/ribaltamenti, gli schemi essenzialmente differenti sono "soltanto" quasi cinque miliardi e mezzo (sequenza A109741 in OEIS). Più avanti, vi pro porrò alcuni quesiti, di non eccessiva difficoltà, riguardanti appunto le simmetrie, sia dei quadrati latini  $3\times 3$  sia degli schemi di Sudoku  $4\times 4$ .

Un buon articolo introduttivo a questo solitario è *La scienza del Sudoku*, di Jean-Paul Delahaye, su *Le Scienze* di agosto 2006, pp. 61-67, apparso nell'edizione francese a dicembre 2005 col titolo *Le tsunami du Sudoku*. Successivamente, sono stati provati interessanti risultati: nel 2013, ad esempio, è stata confermata la necessità di almeno 17 numeri nello schema iniziale affinché questo possa avere una sola soluzione,<sup>1</sup> e al momento è noto un solo puzzle con 16 numeri che ammetta soltanto due soluzioni (tutti gli altri che si conoscono ne hanno di più): eccolo qui sotto.

5	2		4					
	7	1						3
7	2							
1								
6		2						
		3						1
4								

Un'osservazione: ammesso che questo schema sia risolvibile, è presto provato che le soluzioni debbano essere almeno due; infatti, non vi compaiono le cifre 8 e 9, le quali dunque sono intercambiabili.

Una raccolta di Sudoku "minimi", con sole 17 caselle inizialmente assegnate, è stata fatta da Gordon Royle, della University of Western Australia; si veda la pagina [http://www.csse.uwa.edu.au/~gordon/sudoku\\_min.php](http://www.csse.uwa.edu.au/~gordon/sudoku_min.php).

<sup>1</sup> La ditta giapponese Nikoli, che fu la prima a pubblicare i Sudoku, nel 1986 aggiunse una regola "estetica": gli schemi iniziali dovevano rispettare una simmetria rotazionale di 180 gradi delle caselle contenenti i numeri già assegnati, come accade, ad esempio, nel primo schema alla pagina successiva. Se vigesse questa regola, non basterebbe assegnare inizialmente 17 caselle: l'eventuale unicità della soluzione ne richiederebbe almeno 18.

Uno degli schemi più facili da risolvere che mi sia capitato di trovare è il seguente:

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	1 79	13 79	2 8	36 79	5	136 79	136 79	46 79	
R2	15	6 135 79	1	379	4	135 789	123 789	279	
R3	4 8	135 79	16	2	179	135 679	136 79	679	
R4	3	124	8 12 46	469	129	679	26 79	5	
R5	9	245	456	7 45 68	3	68	268	1	
R6	7	125	156	12 56	56	12 89	4	26 89	3
R7	256	235 79	356 79	25	1	27	36 79	4	8
R8	12 68	124 79	146 79	3	478	278	16 79	5	679
R9	158 57	134 57	134 57	9	45 78	6	2	137	7

Le caselle in giallo sono quelle inizialmente assegnate; in ciascuna delle altre sono riportate, in verde, le cifre che vi potrebbero stare, considerate appunto le imposizioni iniziali. Ad esempio, in R2C1 (riga 2, colonna 1) possono stare soltanto 1 o 5, poiché le altre sette cifre sono già assegnate a caselle del primo riquadro (quello in alto a sinistra) o della seconda riga o della prima colonna. In realtà vi potrà stare soltanto 5, poiché 1 è la sola cifra assegnabile alla casella R1C1 (e pure a R2C4). Vi sono tre caselle (per l'appunto R1C1 e R2C4, e poi anche R9C9) nelle quali può stare una sola cifra, che pertanto può essere fissata (cioè scritta in nero) e cancellata dalle altre caselle (in verde) dei rispettivi riquadri/righe/colonne. Procedendo ripetutamente in questo modo, qui si arriva senza alcun intoppo a completare tutto lo schema!

Secondo esempio: il seguente schema iniziale è assai più complesso da risolvere...

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	2 234	35	7	24 56	256	8	9	1	
R2	127 47	123	135 79	13 45	8	12 59	6	247	47
R3	12 78	6	17 89	14	12 49	129	3	247	5
R4	4	127	167	8	12 67	3	179	5	679
R5	9	12 37	13 67	156	125 67	125 67	147	13 47	8
R6	178	5	136 78	9	167	4	17	137	2
R7	3	9	2	145	14 57	15 78	14 57	6	47
R8	5	178	178	146	3	167 89	124 79	124 78	479
R9	6	178	4	2	15 79	157 89	15 79	13 78	379

In R1C1 può stare soltanto 2, che quindi fissiamo e cancelliamo dalle altre caselle di stesso riquadro/riga/colonna. Per i passi successivi, controlliamo se c'è qualche riga

o colonna o riquadro in cui una certa cifra, in verde, compare in una sola casella: allora lì può essere scritta, in nero, e cancellata dalle altre caselle di stesso riquadro/ riga/colonna. In questo caso, con riferimento alle caselle cerchiate in rosso:

- considerando la riga 7, la cifra 8 può stare soltanto in R7C6 ...
- considerando la colonna 4, la cifra 3 può stare soltanto in R2C4 ...
- considerando la colonna 7, la cifra 2 può stare soltanto in R8C7 ...
- considerando la colonna 9, la cifra 3 può stare soltanto in R9C9 ...
- considerando la colonna 9, la cifra 6 può stare soltanto in R4C9 ...
- considerando la riga 4, la cifra 9 può stare soltanto in R4C7 ...
- considerando la colonna 9, la cifra 9 può stare soltanto in R8C9 ...

Gli ultimi due passi sono resi possibili dai precedenti. Arriviamo così al seguente schema:

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	2	34	35	7	456	56	8	9	1
R2	17	147	15 79	3	8	12 59	6	247	47
R3	178	6	17 89	14	12 49	129	3	247	5
R4	4	127	17	8	127	3	9	5	6
R5	9	12 37	13 67	156	125 67	125 67	147	13 47	8
R6	178	5	136 78	9	167	4	17	137	2
R7	3	9	2	145	14 57	8	14 57	6	47
R8	5	178	178	146	3	167	2	14 78	9
R9	6	178	4	2	15 79	15	157	178	3

nel quale notiamo che due caselle in seconda riga sono le sole possibili sia per la cifra 5 sia per la cifra 9, sicché da R2C3 si possono cancellare 1 e 7, e da R2C6 si possono cancellare 1 e 2. Ciò fatto, notiamo che in seconda riga la cifra 2 rimane soltanto in R2C8; e pertanto, in questa casella, scriviamo in nero la cifra 2, che tosto cancelliamo dalle altre caselle di stesso riquadro/riga/colonna.

Otteniamo dunque la situazione illustrata qui sotto.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	2	34	35	7	456	56	8	9	1
R2	17	147	59	3	8	59	6	2	47
R3	178	6	17 89	14	12 49	129	3	47	5
R4	4	127	17	8	127	3	9	5	6
R5	9	12 37	13 67	156	125 67	125 67	147	13 47	8
R6	178	5	136 78	9	167	4	17	137	2
R7	3	9	2	145	14 57	8	14 57	6	47
R8	5	178	178	146	3	167	2	14 78	9
R9	6	178	4	2	15 79	15	157	178	3

Consideriamo ora l'intersezione tra seconda riga e primo riquadro (segnata in rosso): la cifra 1 non appare altrove in seconda riga, per cui possiamo cancellare 1 dalle altre caselle del primo riquadro (qui tale cancellazione interessa R3C1 e R3C3).

Allo stesso modo, considerando l'intersezione tra seconda colonna e settimo riquadro (quello in basso a sinistra), notiamo che la cifra 8 non appare altrove in seconda colonna, e pertanto la cifra 8 può essere cancellata dalle altre caselle del settimo riquadro (in effetti, soltanto da R8C3). Giungiamo così allo schema seguente:

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	2	34	35	7	456	56	8	9	1
R2	17	147	59	3	8	59	6	2	47
R3	78	6	789	14	12 49	129	3	47	5
R4	4	127	17	8	127	3	9	5	6
R5	9	12 37	13 67	156	125 67	125 67	147	13 47	8
R6	178	5	136 78	9	167	4	17	137	2
R7	3	9	2	145	14 57	8	14 57	6	47
R8	5	178	17	146	3	167	2	14 78	9
R9	6	178	4	2	15 79	15 79	157	178	3

in cui notiamo che due caselle della terza colonna contengono soltanto la coppia di cifre 1 e 7: a una delle due (non sappiamo ancora quale) sarà assegnato 1, all'altra 7; però possiamo cancellare sia 1 sia 7 dalle altre caselle della terza colonna!

Nella fattispecie, così procedendo, in R5C3 restano le cifre 3 e 6, e questo ci permette di riconoscere una particolare configurazione, nota col nome *XY-Wing*, evidenziata nello schema sottostante: R1C3, R5C3 e R1C6 contengono le tre possibili coppie formate dalle cifre 3, 5 e 6.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	2	34	35	7	456	56	8	9	1
R2	17	147	59	3	8	59	6	2	47
R3	78	6	89	14	12 49	129	3	47	5
R4	4	127	17	8	127	3	9	5	6
R5	9	12 37	36	156	125 67	125 67	147	13 47	8
R6	178	5	368	9	167	4	17	137	2
R7	3	9	2	145	14 57	8	14 57	6	47
R8	5	178	17	146	3	167	2	14 78	9
R9	6	178	4	2	15 79	15 79	157	178	3

Quale ragionamento ci permette di fare un (piccolo) passo avanti? Se R1C3 = 3, allora R5C3 = 6 e R5C6 ≠ 6; se invece R1C3 = 5, allora R1C6 = 6 e R5C6 ≠ 6. Quindi, in entrambi i casi, R5C6 non può contenere 6: cancelliamo 6 da R5C6.

E adesso? L'ottimo programma *The Sudoku Susser* di Robert Woodhead, in versione 2.5.8 (il documento [www.brophy.net/Downloads/SudokuSusser.pdf](http://www.brophy.net/Downloads/SudokuSusser.pdf) ne contiene un'esauriente descrizione, lunga ben 99 pagine), è stato costretto ad applicare uno dei suoi metodi di deduzione più avanzati (le cosiddette *Trebors's Tables*) per trovare le inferenze che, a partire dallo stato attuale, portano alla soluzione dello schema; a questo scopo, nel caso in esame, il programma ha generato ed esaminato ben 17370 implicazioni! Tali metodi, che costituiscono un soprainsieme di tutti gli altri, sono in grado di risolvere *quasi* tutti i puzzle...

Terzo esempio: a prescindere dalle caselle inizialmente assegnate, supponiamo di giungere a questo schema, che si rivelerà univocamente risolubile (in verde le cifre che legittimamente potrebbero occupare ognuna delle caselle libere, stando alle cifre in nero fissate al momento nelle altre).

Anche qui, con un ragionamento deduttivo (che gli “iniziati” chiamano *Simple X-Wing*), si può eliminare una cifra da una casella, e precisamente 2 da R6C8; infatti, nella quarta e nella nona colonna, la cifra 2 può stare soltanto nelle caselle colorate, o nelle due rosse o nelle due blu, ma comunque non in R6C8.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	3	56	7	26	1	9	8	4	256
R2	69	4	8	7	26	5	1	29	3
R3	569	1	2	3	8	4	56	579	567
R4	4	2	1	5	3	7	9	6	8
R5	567	56	9	8	4	26	3	257	1
R6	567	8	3	26	9	1	4	257	257
R7	2	9	4	1	56	3	7	8	56
R8	1	7	6	4	25	8	25	3	9
R9	8	3	5	9	7	26	26	1	4

A questo punto entra in gioco una delle strategie di soluzione più “esoteriche”, la regola delle *coppie bicolori*, più generale di *X-Wing*, *Swordfish* e altre. Vediamo in che cosa consiste. In quarta colonna, la cifra 6 può stare in sole due caselle: ne coloriamo una in rosso (prima riga) e una in blu (sesta riga).

Ma anche nella sesta riga e nel secondo riquadro la cifra 6 può stare in sole due caselle, e quindi possiamo colorare in rosso una casella della sesta riga e in blu una casella del secondo riquadro. Ora possiamo colorare in rosso R2C1, unica altra casella della seconda riga dove 6 può stare, e così concludiamo per l’impossibilità del fatto che la cifra 6 occupi le caselle rosse: infatti si troverebbero due 6 in prima colonna. Quindi 6 deve stare in entrambe le caselle blu e in nessuna delle rosse.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	3	56	7	26	1	9	8	4	256
R2	69	4	8	7	26	5	1	29	3
R3	569	1	2	3	8	4	56	579	567
R4	4	2	1	5	3	7	9	6	8
R5	567	56	9	8	4	26	3	57	1
R6	567	8	3	26	9	1	4	257	257
R7	2	9	4	1	56	3	7	8	56
R8	1	7	6	4	25	8	25	3	9
R9	8	3	5	9	7	26	26	1	4

Una volta fissato 6 in una delle caselle blu, il completamento dello schema è immediato, come nel primo esempio che abbiamo visto. Si noti che anche altre sei caselle contenenti la cifra 6 potevano essere colorate in rosso (R5C6, R7C5 e R9C7) o in blu (R3C7, R7C9 e R9C6): si veda lo schema in alto alla pagina successiva; e qualora, così procedendo, fosse venuto a crearsi un riquadro/riga/colonna con una casella rossa e una blu, allora la cifra 6 avrebbe potuto essere cancellata dalle altre caselle dello stesso riquadro/riga/colonna. (Ad esempio, se – per mera ipotesi – la casella R2C1 fosse risultata blu, allora avremmo potuto cancellare la cifra 6 sia da R3C1 sia da R5C1, che stanno sulla stessa colonna di due caselle di diverso colore.)

R1	3	56	7	26	1	9	8	4	256
R2	69	4	8	7	26	5	1	29	3
R3	569	1	2	3	8	4	56	579	567
R4	4	2	1	5	3	7	9	6	8
R5	567	56	9	8	4	26	3	57	1
R6	567	8	3	26	9	1	4	257	257
R7	2	9	4	1	56	3	7	8	56
R8	1	7	6	4	25	8	25	3	9
R9	8	3	5	9	7	26	26	1	4

Non sempre, però, la tecnica delle coppie bicolori risolve il problema! Se provassimo ad applicarla al caso dell'esempio precedente, non otterremmo alcun successo...

Ovviamente, avremmo potuto procedere per tentativi. Torniamo un attimo al primo schema della pagina precedente, e supponiamo di assegnare 6 alla casella R1C4; tra le conseguenze, ci interessano in particolare: in R1C2 e in R6C4 rimangono soltanto 5 e 2, rispettivamente, per cui 5 può essere cancellato da R5C2 e così 2 da R5C6; ma poiché in R5C2 resta soltanto 6, questa cifra può essere cancellata da R5C6, sicché questa casella rimane *vuota*, senza più cifra che possa occuparla! Ciò ci permette di concludere che il tentativo fatto era sbagliato, e quindi a R1C4 bisogna assegnare 2.

Quarto esempio: questo è lo schema iniziale più insidioso che sono riuscito a trovare per mettere in crisi *The Sudoku Susser*.

R1	169	2	56 89	4	16 89	3	7	156 89	156 89
R2	146 79	156 789	456 789	156 78	167 89	56 79	158	3	2
R3	136 79	156 789	356 789	156 78	126 789	256 79	158 89	156 89	4
R4	13 69	4	35 69	2	36 89	69	13 58	7	135 689
R5	8	16 79	236 79	367	5	46 79	12 34	124 69	13 69
R6	236 79	56 79	235 679	36 78	346 789	1	234 58	245 689	356 89
R7	5	678 78	246 67	13 467	123 467	24 67	9	12 48	138
R8	246	3	24 68	9	12 46	24 56	124 58	124 58	7
R9	24 79	79	1	357	23 47	8	6	245	35

Già al primo passo, il programma è costretto ad applicare la regola dell'intersezione: considerando le tre caselle indicate nell'ovale (intersezione tra la settima colonna e il sesto riquadro), si può notare che la cifra 3 non appare altrove nella settima colonna,

e dunque la cifra 3 può essere cancellata dalle altre caselle del sesto riquadro (nella fattispecie, tale cancellazione interessa soltanto le tre caselle di questo riquadro che appartengono all'ultima colonna). E poi? Non riuscendo più ad applicare alcuna regola di deduzione, nemmeno tra le più avanzate di cui dispone, il programma si arrende, e risolve il puzzle per tentativi, ricorrendo alla tecnica del *backtracking*, ben nota ormai ai nostri lettori (*cfr.* “Un algoritmo di esaurimento”, pp. 205-208).

Oltre a *The Sudoku Susser*, sono stati sviluppati tanti programmi per risolvere puzzle, e anche per comporli. Come abbiamo accennato, certi risolutori – quelli più completi, e anche complessi – contemplano tutta una varietà di *regole di deduzione*, più o meno euristiche, e cercano di applicarle a partire dalle più semplici, quelle praticabili pure con carta e matita, ma senza gomma, fino a quelle più avanzate, in grado di generare ed esaminare persino decine di migliaia di implicazioni “in avanti”. Esisteranno regole di deduzione talmente sofisticate da poter risolvere qualsiasi schema iniziale, senza mai dover ricorrere alla forza bruta? La questione non è così importante, a mio parere, poiché nel cercare la catena di inferenze, che dallo stato attuale dello schema portano alla sua soluzione, è insita una sorta di *trial-and-error*. Per non perdere lo spirito del gioco, si possono combinare *backtracking* e strategie logiche elementari, facilmente codificabili, come fa ad esempio il programma di Peter Norvig del 2011, scritto in Python e basato su due funzioni mutuamente ricorsive (*Solving every Sudoku puzzle*, <http://www.norvig.com/sudoku.html>). Si trovano poi in rete altri software (commerciali) con strategie avanzate e relative spiegazioni, tra i quali *Sudoku Dragon* (<http://www.sudokudragon.com/>).

Come si presenta un programma in grado di risolvere schemi di Sudoku, procedendo esclusivamente per tentativi? Ad ogni passo, esso deve associare a ciascuna casella l’insieme delle cifre che potrebbero occuparla; ogni volta che assegna una cifra plausibile a una casella, propaga ricorsivamente gli effetti di tale assegnazione eliminando, dalle altre caselle dei rispettivi riquadri/righe/colonne, tutte le cifre che sono rimaste univocamente fissate. Qualora trovi almeno una casella priva di cifre assegnabili – e basta questo controllo per decidere il fallimento! – il programma ripristina lo stato che precedeva l’ultimo tentativo fatto e, se può, procede con un altro tentativo sulla stessa casella: a mano, dunque, servirebbe una gomma, per poter tornare sui propri passi evitando di sciupare il disegno! Se lo schema può essere risolto, allora prima o poi una soluzione è trovata: è sufficiente controllare che sia rimasta una sola possibilità per ciascuna casella.

Inutile dire che il progetto e la realizzazione di tecniche deduttive più sofisticate (che comunque, prima di fare una scelta, devono in qualche modo “dimostrare” che sia una scelta giusta) sono decisamente più complessi – e alla fine si rivelano pure meno efficienti – rispetto all’applicazione del metodo di “forza bruta”, con *backtracking*, al quale ora abbiamo accennato: in ultima analisi, quando arriva a lasciare una sola possibilità in una casella, questo procedimento ha qualche somiglianza con una dimostrazione “per assurdo” della scelta giusta da fare.

Scriviamo in pseudo-codice il procedimento descritto sopra a parole, per trovare una soluzione di uno schema  $S$  o concludere che non ne esistono. Si osservi che si tratta di una funzione che termina con “successo” o con “insuccesso”, ricorsiva (in giallo) all’interno di un ciclo (in azzurro) che considera a uno a uno i tentativi sulla casella prescelta, e con un eventuale effetto collaterale (in verde): la stampa della soluzione trovata! Il calcolo si arresta quando si trova una casella vuota (e in tal caso lo schema al quale si è giunti non ammette soluzione) o quando ciascuna casella contiene una sola cifra (e in tal caso si può star sicuri che lo schema attuale rappresenta una soluzione di quello di partenza, poiché le assegnazioni man mano fatte sono corrette).

```
funzione risolvi lo schema S
  se in S esiste una casella dove non può stare alcuna cifra
    allora termina con "insuccesso";
    altrimenti
      se in S non esiste alcuna casella dove possano stare almeno due cifre
        allora
          stampa lo schema S;
          termina con "successo";
        altrimenti
          sia  $Ric_j$  una casella di  $S$  in cui possono stare almeno due cifre; (*)
          per ogni cifra  $N$  che può stare in  $Ric_j$ 
            fai una copia  $S'$  dello schema  $S$ ;
            lascia soltanto  $N$  nella casella  $Ric_j$  di  $S'$  e cancella  $N$  dalle
            altre caselle di stesso riquadro/riga/colonna (e, ricorsivamente,
            ogni volta che rimane una sola cifra in una casella, cancellala
            dalle altre caselle di stesso riquadro/riga/colonna);
            se risolvi lo schema  $S'$  termina con "successo"
            allora
              termina con "successo";
            altrimenti
              termina con "insuccesso";
```

(\*) *"Miglioramento"*: per cercare di rendere minimo il numero di tentativi (ma non è detto che ci si riesca) converrebbe scegliere una delle caselle col minor numero ( $> 1$ ) di possibilità.

Un po’ più compatta è la codifica di una procedura che trova tutte le soluzioni di uno schema  $S$  (e se termina senza stampare nulla, significa che non vi sono soluzioni):

```
procedura stampa le soluzioni dello schema S
  se in ogni casella di  $S$  può stare almeno una cifra
    allora
      se in  $S$  non esiste alcuna casella dove possano stare almeno due cifre
        allora
          stampa lo schema S;
        altrimenti
          sia  $Ric_j$  una casella di  $S$  in cui possono stare almeno due cifre; (*)
          per ogni cifra  $N$  che può stare in  $Ric_j$ 
            fai una copia  $S'$  dello schema  $S$ ;
            lascia soltanto  $N$  nella casella  $Ric_j$  di  $S'$  e cancella  $N$  dalle
            altre caselle di stesso riquadro/riga/colonna (e, ricorsivamente,
            ogni volta che rimane una sola cifra in una casella, cancellala
            dalle altre caselle di stesso riquadro/riga/colonna);
            stampa le soluzioni dello schema  $S'$ ;
```

Il programma codificato in linguaggio C, sulla falsariga di questo pseudo-codice, risolve lo schema del secondo esempio proposto (si veda alla pagina 304) in soli 34 tentativi (fare un tentativo significa provare a fissare una cifra in una casella che ammette più possibilità) e, con altri 4 soltanto, verifica che lo schema ammette una sola soluzione; e pensare che il programma è proprio “bruto”: per fare un tentativo non cerca neppure una casella che presenti il minor numero ( $> 1$ ) di possibilità, bensì prende in considerazione la prima che trova, la quale presenta almeno due possibilità! (Tuttavia, come s’è detto, non è affatto garantito che un tale accorgimento conduca al minor numero complessivo di tentativi.)

Infine, illustriamo un procedimento per creare uno schema iniziale che ammetta una e una sola soluzione.

disponi “a caso” alcune cifre in uno schema S, inizialmente vuoto;  
stampa le soluzioni dello schema S;

**finché** lo schema S non ha alcuna soluzione

modifica lo schema S, togliendo una cifra da una casella;  
stampa le soluzioni dello schema S;

**finché** lo schema S ha più di una soluzione

scegli tra queste una soluzione, sia essa S';  
modifica lo schema S, aggiungendo in una delle caselle vuote  
la cifra che sta nella corrispondente casella di S'; (\*\*)  
stampa le soluzioni dello schema S;

(\*\*) Se S' fosse la sola soluzione che ha tale cifra in quella casella,  
allora al passo successivo si otterrebbe lo schema desiderato!

Ovviamente, anziché “stampare” le soluzioni, è sufficiente calcolarne il  
numero e, quando sono più di una, ricordarne una soltanto.

Abbiamo detto che i metodi di deduzione più sofisticati (oltre a *Trebors Tables*, citiamo *Bowman Bingo* e *Nishio*) di cui dispongono certi programmi permettono di risolvere “senza gomma” la stragrande maggioranza dei puzzle – ma non tutti, come testimonia l’ultimo esempio fatto.

Vediamo qualche altro esempio di puzzle particolarmente impegnativo. Lo schema qui a destra è noto come *The Toughest Known Puzzle*; tuttavia *The Sudoku Susser* lo risolve sì con l’impiego di due dei suoi metodi di deduzione più avanzati (*Trebors Tables* e *Bowman Bingo*), ma senza dover ricorrere alla forza bruta.

Il mio programma di forza bruta trova la soluzione facendo 238 tentativi, e con altri 931 conferma la sua unicità.

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	12 56	128 68	125 68	13 68	7	123 68	9	4	12 36
R2	126 78	124 68	124 68	13 68	9	123 468	23 68	136	5
R3	3 89	124 689	124 689	168	12 48	5	268	7	126
R4	259 289	7	4	23 58	236 89	1	35 69	23 69	
R5	4	6	3	15 89	12 58	12 89	257	59	279
R6	12 59	129 59	12 59	135 69	12 35	7	234 56	8	234 69
R7	8 49	123 69	124 79	135 45	13 49	13 49	345 67	135 69	134 679
R8	7 49	13 69	14 59	13 45	13 45	13 49	34 56	2	8
R9	19	5	149	2	6	134 89	347	139	134 79

Questo schema è stato chiamato *AI Escargot*: qui *AI* non sta per *Artificial Intelligence*, bensì sono le iniziali di Arto Inkala, il matematico finlandese che lo ha composto nel 2006.

Come dice il nome (*escargot* è la lumaca, in francese) è uno schema particolarmente lungo da risolvere; tuttavia bastano strategie piuttosto semplici...

Anche quest'altro schema, del 2010, è opera dello stesso autore, che lo ha definito «il puzzle più difficile che io abbia mai creato».

A un certo punto, *The Sudoku Susser* applica *Trebors Tables* o *Nishio*, ma poi si arrende e per giungere alla soluzione deve ricorrere alla forza bruta.

Questo è il sesto schema che figura nella lista di 11 scelti da Peter Norvig tra i più difficili (il precedente era al secondo posto).

Dopo aver fissato 1 in R8C3 (è la sola casella della terza colonna in cui può stare la cifra 1), *The Sudoku Susser* applica più volte la regola *Nishio*, ma poi anche qui è costretto a impiegare la forza bruta.

R1	C1	C2	C3	C4	C5	C6	C7	C8	C9
	8	5	13 69	36 79	16 79	2	4	136	13 67
R2	7	2	136	345 68	14 56	134 58	135 68	135 68	9
R3	169	369	4	356 789	156 79	135 89	1235 678	123 568	135 678
R4	69	689	689	1	45 69	7	35 68	35 68	2
R5	3	678	5	24 68	246	48	9	168	146 78
R6	12 69	4	126 89	235 689	25 69	35 89	135 678	135 68	135 678
R7	245 69	369	23 69	24 59	8	14 59	123 56	7	13 56
R8	245 69	1	7	24 59	24 59	459	235 68	235 689	35 68
R9	259	89	289	25 79	3	6	12 58	4	158

R1	C1	C2	C3	C4	C5	C6	C7	C8	C9
	12 69	249	5	3	246 89	246 78	146 89	146 79	14 78
R2	8	349	169	46 79	45 69	467	14 69	2	13 47
R3	23 69	7	269	46 89	1	24 68	5	469	348
R4	4	289	267 89	16 89	689	5	3	179	127
R5	259	1	289	489	7	348	249	459	6
R6	56 79	59	3	2	469	146	149	8	14 57
R7	12 37	6	12 78	5	23 48	123 478	12 48	14	9
R8	125 79	25 89	4	16 78	268	126 78	12 68	3	12 58
R9	12 35	23 58	128	14 68	234 68	9	7	14 56	124 58

R1	C1	C2	C3	C4	C5	C6	C7	C8	C9
	1	25 68	24 68	458	345	7	246	9	346
R2	457	3	46	14 59	2	159	14 67	467	8
R3	24 78	278	9	6	134	138	5	23 47	134
R4	24 78	278	5	3	167	126	9	46 78	146
R5	479	1	34	579	8	569	467	345 67	2
R6	6	27 89	238	125 79	15 79	4	178	35 78	135
R7	3	256 89	268	245 789	456	256 89	24 68	1	45 69
R8	25 89	4	12 68	125 89	135 69	1235 689	268	25 68	7
R9	25 89	256 89	7	124 589	145 69	125 689	3	245 68	45 69

Lo schema qui a destra ha soltanto 17 caselle inizialmente assegnate, e si noti che ben tre cifre, 2, 7 e 9, non vi compaiono, per cui non potrebbe avere una sola soluzione; in realtà, non ne ha nessuna!

Per decidere che non vi è soluzione, il programma di Norvig del 2011 impiegò 24 minuti; il mio in versione “migliorata” (si veda la nota (\*) a p. 310), su un personal computer coeve, giunse allo stesso risultato in meno di 9 minuti, facendo oltre 22 milioni di tentativi.

Anche questo schema ha soltanto 17 caselle inizialmente assegnate (e si noti che non vi compaiono le cifre 1 e 7), ma ha quasi 150 milioni di soluzioni!

Per trovarne una, il mio programma fa poco più di 4500 tentativi in circa un decimo di secondo; tuttavia, nella sua versione “migliorata” impiega 72 secondi, facendo oltre un milione e mezzo di tentativi!

Per trovarle tutte quante, ci son voluti 113 minuti e i tentativi fatti sono stati quasi 350 milioni!

Come quest’ultimo caso attesta, scegliere una casella col minor numero ( $> 1$ ) di possibilità ognqualvolta si debba fare un tentativo, sperando così di risparmiare tempo, non sempre migliora l’efficienza, ma potrà farlo soltanto in senso statistico.

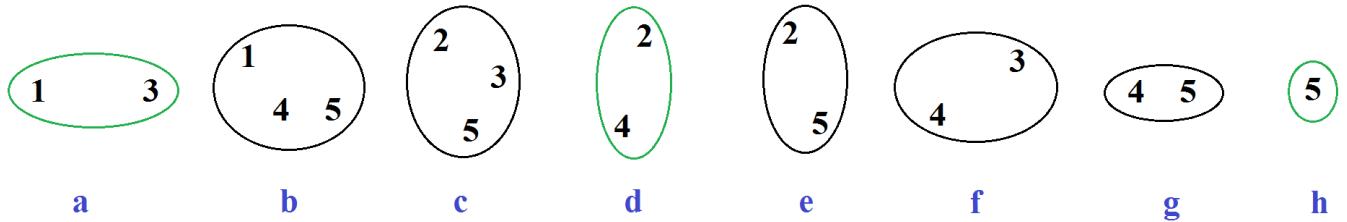
Un puzzle di Sudoku può essere ricondotto a un problema di colorazione di un grafo (non orientato): nel caso classico  $9 \times 9$ , il grafo ha 81 nodi (uno per ogni casella) e 810 archi (ciascun nodo è collegato con un arco ad ognuno dei 20 nodi associati alle caselle del suo stesso riquadro/riga/colonna), e alcuni nodi (almeno 17) sono già “colorati” (a ciascuno di essi è stato assegnato, in modo corretto, un colore fra i 9 possibili). Si tratta allora di colorare anche gli altri nodi, in modo tale che due nodi collegati da un arco abbiano comunque colori diversi. Come sappiamo, in generale questo è un problema NP-arduo, e in effetti è stato provato che risolvere puzzle di Sudoku  $n^2 \times n^2$  rientra nei compiti di tale complessità. (Per inciso, non si conosce nemmeno il numero degli schemi finali di Sudoku per  $n > 3$ .)

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	124 679	246 79	1234 679	234 79	234 79	5	126 79	8	26 79
R2	27 89	257 89	257 89	6	27 89	1	25 79	4	3
R3	1246 789	2456 789	12345 6789	234 789	234 789	234 789	125 679	125 79	26 79
R4	246 789	1	246 789	5	234 789	234 789	2346 789	23 79	267 89
R5	247 89	245 789	245 789	1	234 789	6	234 789	23 79	27 89
R6	3	246 789	246 789	247 89	247 89	247 89	1246 789	12 79	5
R7	5	3	247 89	247 89	247 89	247 89	27 89	6	1
R8	126 789	267 89	126 789	237 89	1235 6789	237 89	235 789	235 79	4
R9	1246 789	246 789	1246 789	234 789	12345 6789	234 789	235 789	235 79	27 89

	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	134 78	13 78	14 78	124 579	134 579	6	1234 579	123 479	123 579
R2	134 67	5	9	12 47	13 47	12 47	123 467	123 467	8
R3	2	13 67	147	145 79	134 579	8	1345 679	134 679	135 79
R4	17 89	4	5	126 789	167 89	12	1236 789	1236 789	123 79
R5	17 89	127 89	3	1245 6789	1456 789	124	126 789	126 789	12 79
R6	17 89	127 89	6	127 89	17 89	3	127 89	5	4
R7	147 89	17 89	14	3	2	5	147 89	147 89	6
R8	1345 6789	1236 789	124 78	146 789	146 789	14	1234 5789	1234 5789	123 579
R9	1345 6789	1236 789	124 78	146 789	146 789	14	1234 5789	1234 5789	123 579

## 2. L'esatta copertura di un insieme.

Risolvere un puzzle di Sudoku può pure ricondursi a determinare un'esatta copertura di un insieme. Questo problema, detto *exact cover set* nella letteratura anglosassone, fu dimostrato NP-completo, in versione decisionale, da Richard M. Karp nel 1972; una sua semplice istanza è la seguente: è possibile scegliere alcuni degli otto insiemi sotto disegnati, in modo che sia preso *una e una sola volta* ciascun numero da 1 a 5?



Per rispondere affermativamente, occorre però determinare una scelta che soddisfi il requisito; in questo caso ce n'è soltanto una: a, d, h.

I dati necessari a un programma al computer, atto a risolvere il problema in generale, possono essere semplicemente messi nella forma di una matrice di bit: l'elemento di riga  $i$  e colonna  $j$  è 1 se e soltanto se  $j$  appartiene all'insieme  $i$ ; nel nostro caso:

	1	2	3	4	5
a	1	0	1	0	0
b	1	0	0	1	1
c	0	1	1	0	1
d	0	1	0	1	0
e	0	1	0	0	1
f	0	0	1	1	0
g	0	0	0	1	1
h	0	0	0	0	1

Dobbiamo estrarre, se possibile, un insieme di righe che presenti esattamente un 1 in ciascuna colonna. Nell'esempio fatto un modo c'è, ma uno solo; se aggiungessimo un nono insieme, costituito dai numeri 1, 2 e 3, allora le soluzioni sarebbero due; tre, se aggiungessimo anche l'insieme formato da 1 e 2.

Ovviamente qualcuno potrebbe essere interessato a trovare una soluzione che comporti la scelta del minor numero di insiemi; ma ora non è questa la nostra preoccupazione: se un puzzle di Sudoku è *ben costruito*, ammette una e una sola soluzione... ma per saperlo, bisogna provarlo!

Vediamo dunque come un puzzle di Sudoku può essere ricondotto a un'istanza del problema dell'esatta copertura di un insieme, considerando per semplicità il caso di uno schema  $4 \times 4$  (la più facile versione del Sudoku, detta Shidoku), anziché  $9 \times 9$ .

34	1	23	234
2	4	13	134
1	2	4	123
14	3	12	12

All'incrocio tra la riga 3 e la colonna 4 potrebbe stare il 2, che non è già messo né in riga 3 né in colonna 4 e neppure nell'ultimo riquadro (ma nello schema finale non sarà così: il 2 in terza riga è obbligato a stare nella seconda casella).

Se immaginiamo di fissare lì il 2, allora:

- la casella (3, 4) non è più occupabile (vincolo di casella);
- nella terza riga c'è il 2 (vincolo di riga);
- nella quarta colonna c'è il 2 (vincolo di colonna);
- nel quarto riquadro c'è il 2 (vincolo di riquadro).

	Vincoli di casella	Vincoli di riga	Vincoli di colonna	Vincoli di riquadro
	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
	1234123412341234	1234123412341234	1234123412341234	1234123412341234
r.c.#	-----	-----	-----	-----
3.4.2	...	1	1	1
	...			

Nella riga 3.4.2 (riga.colonna.numero) della matrice di bit vi sono dunque quattro 1 (così come in ognuna delle altre righe) e i restanti bit sono 0. Nel caso di Sudoku 4×4 la matrice di bit è quadrata (64×64), ma è un caso; ogni numero fissato nello schema iniziale esclude tuttavia tre righe, e dunque nell'esempio qui sopra le righe sono 52.

In generale, nel Sudoku  $n^2 \times n^2$ , la matrice di bit avrà  $4n^4$  colonne e  $n^6 - k$  ( $n^2 - 1$ ) righe, se  $k$  sono i numeri fissati nello schema iniziale; in ciascuna riga vi saranno comunque quattro 1. E così, nel Sudoku 9×9, la matrice avrà 324 colonne e, al più, 593 righe, poiché le caselle assegnate inizialmente sono almeno 17.

Per risolvere il puzzle bisogna determinare un sottoinsieme di  $n^4$  righe (81, nel Sudoku classico) che presenti esattamente un 1 in ciascuna colonna.

Il problema decisionale dell'esatta copertura di un insieme è NP-completo, dunque troppo impegnativo dal punto di vista computazionale quando la dimensione diventa abbastanza grande; nel caso del Sudoku è noto il numero dei sottoinsiemi (righe) da scegliere, e occorre esibire una soluzione (se esiste, a prescindere dalla sua unicità) anziché limitarsi a decidere, ma tutto ciò non altera la complessità del problema: non si conosce un metodo efficiente per risolverlo in generale, né si sa se un tale metodo esista. Si possono applicare algoritmi specifici, che nei casi più sfortunati richiedono tuttavia un tempo di calcolo che cresce in modo esponenziale con la dimensione del problema da risolvere – nella fattispecie, le dimensioni della matrice di bit... E quando si deve trattare con una crescita del genere, come ben sappiamo, si fa presto ad arrivare a tempi di calcolo proibitivi, anche disponendo dei computer più veloci!

Donald E. Knuth – un grande nome dell’informatica! – ha ideato un procedimento per risolvere il problema in generale, basato sul *backtracking* ma realizzato con una tecnica particolare, chiamata *dancing links*, le cui prestazioni si sono rivelate ottime persino in casi di ragguardevoli dimensioni. Dopo aver trasformato un puzzle di Sudoku in un’istanza di *exact cover set* – ciò che si può fare in modo efficiente – possiamo quindi applicare *Algorithm DLX*, che lo risolverà quasi sempre in tempo brevissimo, anche quando lo schema è  $16 \times 16$  o  $25 \times 25$  o ancor più grande. In rete è disponibile l’articolo originale di Knuth *Dancing Links* (Stanford University, 2000):  
<http://www-cs-faculty.stanford.edu/~knuth/papers/dancing-color.ps.gz>

Una compatta realizzazione in Python di *Algorithm X*, che però usa insiemi anziché liste bidirezionali circolari (e applicata al Sudoku  $9 \times 9$  richiede qualche centesimo di secondo, al più alcuni secondi per i puzzle più difficili), è pure disponibile in rete:

Ali Assaf, *Algorithm X in 30 lines!* (2013)

[http://www.cs.mcgill.ca/~aassaf9/python/algorithm\\_x.html](http://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html)

In altri istruttivi articoli, *Algorithm DLX* è spiegato in dettaglio e applicato a vari divertenti rompicapi, oltre alla creazione di nuovi schemi iniziali di Sudoku e alla loro risoluzione: quadrati latini e *polimini* (in particolare pentamini, di cui diremo nel prossimo paragrafo, e Kanoodle). In rete potete trovare i seguenti:

Andrzej Kapanowski, *Python for Education: The Exact Cover Problem*, The Python Papers 6, 2 (2011)

<http://ojs.pythonpapers.org/index.php/tpp/article/view/227>

David Austin, *Puzzling Over Exact Cover Problems*, American Mathematical Society (2015)

<http://www.ams.org/samplings/feature-column/fcarc-kanoodle>

Team # 3140, *The Application of Exact Cover to the Creating of Sudoku Puzzle*

<http://www.math.utah.edu/~yzhang/teaching/1030/Sudoku.pdf>

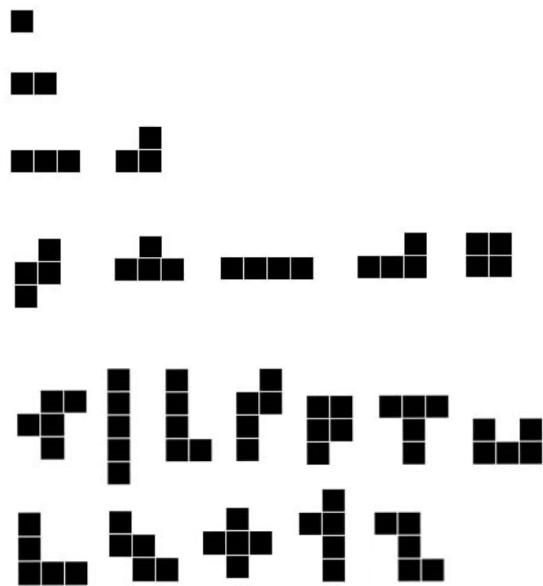
Un’ultima considerazione, a proposito della creazione di schemi iniziali di Sudoku: ha senso chiedersi se uno schema, che ammetta una sola soluzione e che presenti più di 17 caselle inizialmente assegnate, sia *minimale*, vale a dire se, per ognuna delle assegnazioni stabilite, detto schema si trovi ad ammettere più soluzioni qualora sia rimossa tale assegnazione. Finora non è precisamente noto quanti siano gli schemi iniziali minimali.

### 3. Polimini, in particolare pentamini.

Col nome di “polimini” si indicano particolari figure geometriche, ovvero “pezzi di puzzle”, composti da quadratini di ugual lato, ciascuno dei quali ha un lato in comune con almeno un altro quadratino. Essi sono stati ideati nel 1953 da Solomon W. Golomb, allora studente all’università di Harvard, che l’anno dopo pubblicò un articolo al proposito (*Checkerboards and Polyominoes*, The American Mathematical Monthly, Vol. 61, No. 10, dicembre 1954, pp. 675-682). A Martin Gardner, al quale nulla di interessante sfuggiva, si deve la loro divulgazione, su *Scientific American* del dicembre 1957. Molto materiale sull’argomento si trova ovviamente in rete, ad esempio la pagina <http://www.bitman.name/math/article/752/> di Mauro Fiorentini, con una ricca bibliografia, ma il testo di riferimento rimane il libro di Golomb *Polyominoes. Puzzles, Patterns, Problems, and Packings. Revised and expanded second edition* (Princeton University Press, 1994).

Abbiamo detto che si tratta di poligoni costituiti da  $n$  quadrati unitari, ove ciascun quadrato condivide almeno un lato con un altro quadrato; e dunque, considerando figure identiche quelle ottenute l’una dall’altra mediante rotazione e/o ribaltamento:

- per  $n = 1$ , si ha un *monomino*
- per  $n = 2$ , un *duomino*
- per  $n = 3$ , si hanno due *trimini*
- per  $n = 4$ , cinque *tetramini*
- per  $n = 5$ , dodici *pentamini*

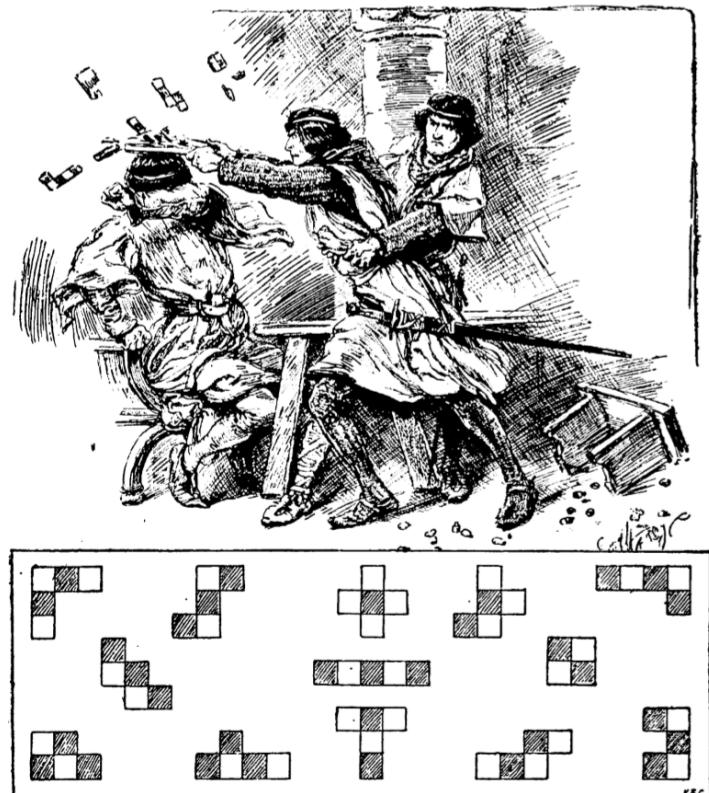


Per  $n = 6, 7, 8, 9, 10, 11, 12, \dots$  le figure diverse sono, rispettivamente: 35, 108, 369, 1285, 4655, 17073, 63600, ... (sequenza A000105 in OEIS).

Il merito di aver introdotto i pentamini va tuttavia riconosciuto all’inglese Henry Ernest Dudeney, uno dei maggiori inventori di enigmi e giochi matematici di tutti i tempi, del quale abbiamo già parlato diffusamente. L’illustrazione riportata alla pagina successiva compare nella sua prima raccolta, *The Canterbury Puzzles* (cfr. p. 300), pubblicata a Londra da William Heinemann nel 1907, a mo’ di introduzione al problema numero 74, *The broken chessboard* (ivi, pp. 90-92 e 174-175).

Dudeney soleva premettere una storiella alla spiegazione del gioco. In questo caso, l’autore riporta un aneddoto che si riferisce a Roberto ed Enrico, due dei figli di

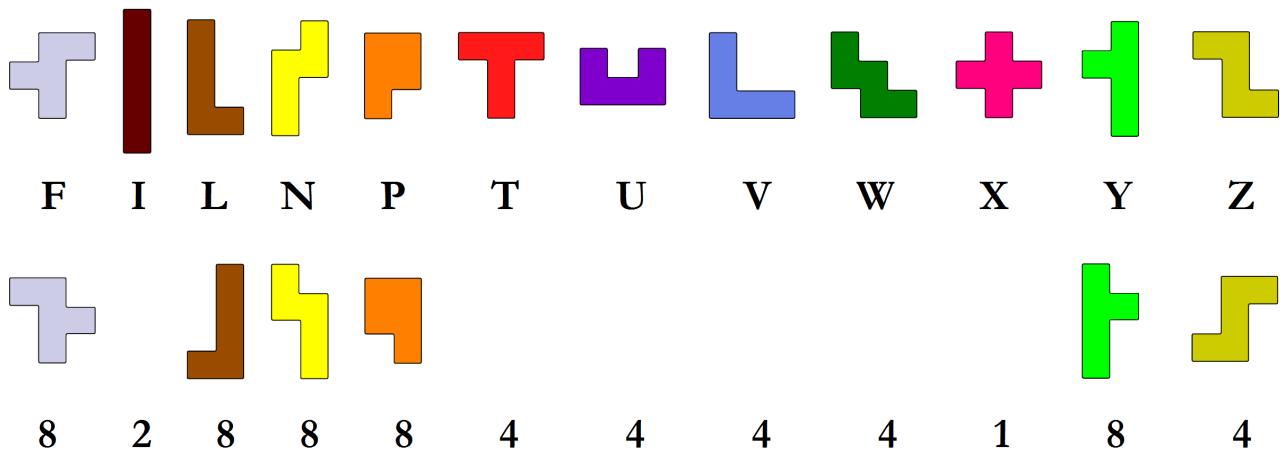
Guglielmo I il Conquistatore, re d'Inghilterra. Siamo nella seconda metà del secolo XI; i due fratelli, in visita a Costanza presso la corte francese, trovano pure il tempo per svagarsi. Enrico gioca a scacchi con Luigi, Delfino di Francia, e lo vince ripetutamente, finché Luigi, innervosito oltre ogni limite, lancia i pezzi sul viso di Enrico, il quale a sua volta, benché trattenuto dal fratello Roberto, riesce a rompere la scacchiera sulla testa di Luigi...



In seguito, Enrico non solo tolse il ducato di Normandia al fratello primogenito Roberto, ma pure, approfittando dell'assenza di questi impegnato nelle Crociate, gli usurpò il trono alla morte del fratello terzogenito Guglielmo II il Rosso, succeduto al padre, e divenne Enrico I, re d'Inghilterra... ma questa è (un'altra) storia!

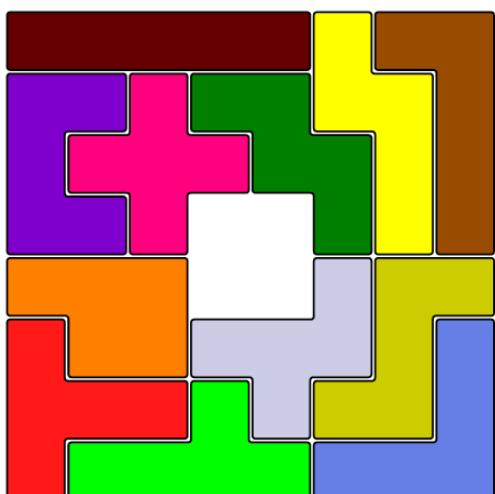
Dudeney immagina che la scacchiera si sia rotta nei tredici frammenti, tutti di forma diversa: i dodici pentamini più il tetramino quadrato, mostrati nell'illustrazione; invita il lettore a costruirli, ritagliandoli da un cartoncino quadrettato, e gli assicura un divertente passatempo nel cercare di ricomporre la scacchiera originaria – senza tuttavia precisare se le caselle siano bianche o nere anche sul retro, e se quindi i vari pezzi possano essere ribaltati o meno... Dudeney ne dà una possibile soluzione, senza ribaltamenti; a voi il piacere di trovarne almeno una!

Alla pagina seguente potete vedere i dodici pentamini, in altrettanti colori uniformi; per ognuno di essi sono riportati la lettera che usualmente lo denota e il numero dei diversi orientamenti che può assumere, considerando rotazioni di 90 gradi, o multipli, e/o ribaltamenti lungo l'asse orizzontale, o verticale. Almeno sei pentamini devono essere colorati su entrambe le facce: infatti, le figure in basso non si ottengono con semplici rotazioni dalle corrispondenti in alto, ma occorre un ribaltamento!



Uno dei primi programmi che hanno impiegato la tecnica del *backtracking* è dovuto all’eminente logico Dana S. Scott, che lo realizzò nel 1958 con l’aiuto di Hale F. Trotter, proprio allo scopo di calcolare tutte le soluzioni essenzialmente differenti di un puzzle combinatorio con i 12 pentamini: si trattava di formare un quadrato  $8 \times 8$ , con un “buco”  $2 \times 2$  al centro, come mostrato nella figura qui sotto (*Programming a combinatorial puzzle*, Technical Report No. 1, Department of Electrical Engineering, Princeton University, 10 giugno 1958).

Come ora vedremo, questo problema può essere ricondotto a tre distinte istanze del già noto *exact cover set*, a seconda della collocazione del pentamino X (quello a forma di croce greca, che ha un solo orientamento possibile) sulla “scacchiera”  $8 \times 8$ .

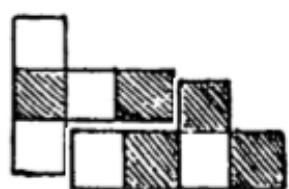


In circa tre ore e mezza, il programma di allora trovò le 65 possibili soluzioni, di cui:

- 19 con X centrato in (2, 3);
  - 20 con X centrato in (2, 4);
  - 26 con X centrato in (3, 3) (come quella a lato) e il pentamino P (quello subito sotto a X in figura a fianco) non ribaltato, per evitare di contare due volte le soluzioni simmetriche rispetto alla direzione NordOvest-SudEst.
- Perché proprio P? Perché esso presenta il maggior numero di possibili collocazioni, che in tal modo vengono dimezzate!

Prima di procedere, un’osservazione: se immaginassimo di sistemare i pezzi del problema di Dudeney come nella figura qui sopra, aggiungendovi il tetramino quadrato al centro, non risolveremmo il puzzle, anche potendo ribaltare i pezzi della scacchiera (si veda l’immagine a fianco).

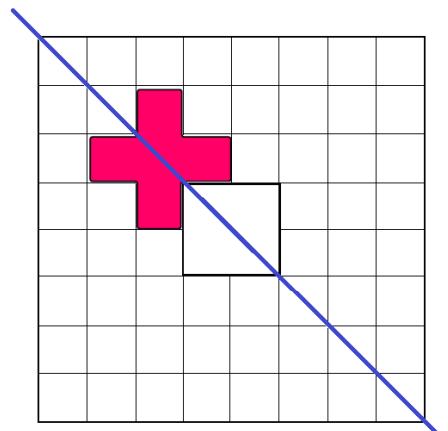
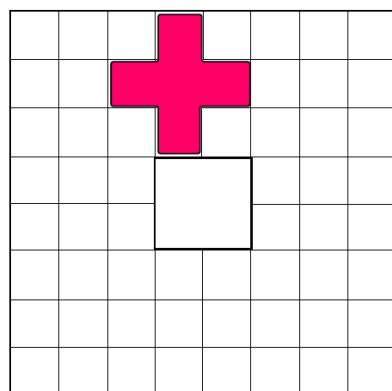
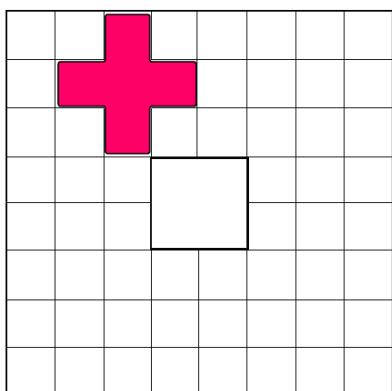
D’altronde, Dudeney non richiede che il tetramino quadrato sia al centro – e infatti nella sua soluzione non lo è!



Per ciascuno dei tre casi illustrati nella prossima figura, in cui X è già collocato:

- ognuno degli altri 11 pezzi deve essere collocato una e una sola volta;
- ognuna delle 55 caselle rimaste libere deve essere occupata da uno e uno solo degli altri 11 pezzi (non vi devono essere sovrapposizioni, seppur parziali).

Nel terzo caso, come si è detto, il pentamino P non sarà ribaltato.

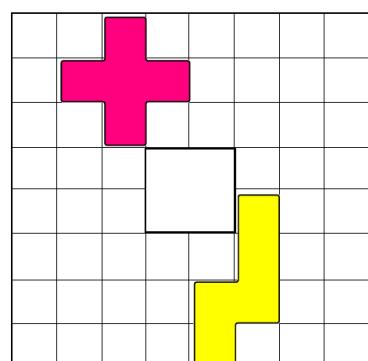


Pertanto, ciascuna delle tre istanze del problema di “esatta copertura” ha come dati una matrice di bit con 66 colonne (11 i pezzi, collocato X, e 55 le caselle rimanenti) e una riga (con esattamente sei 1) per ogni collocazione ammissibile di ciascuno degli 11 pezzi rimanenti.

Sia ciascun pezzo sia ciascuna casella devono essere presi una e una sola volta, e dunque ad ogni sottoinsieme costituito da 11 righe, che presenti esattamente un 1 in ciascuna colonna, corrisponde una soluzione.

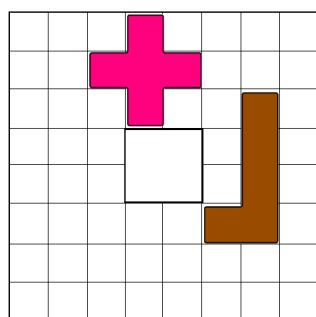
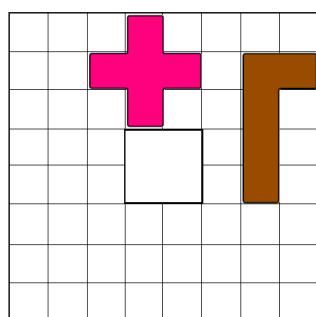
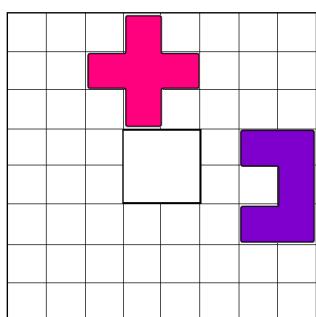
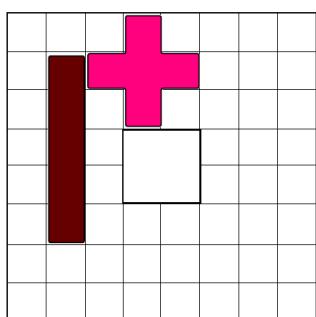
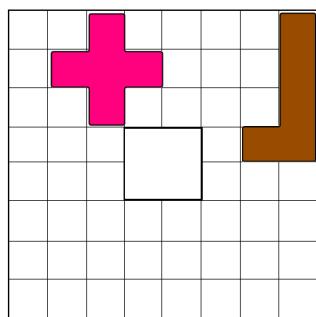
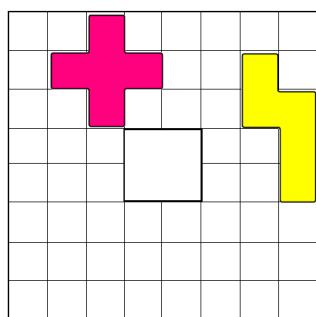
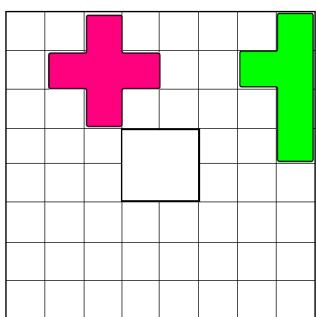
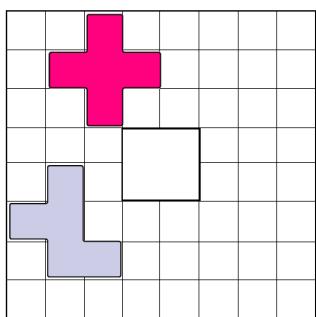
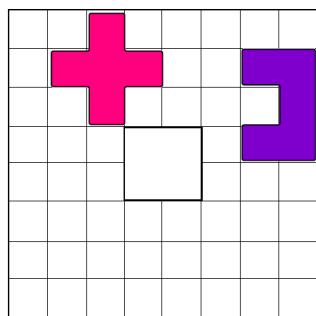
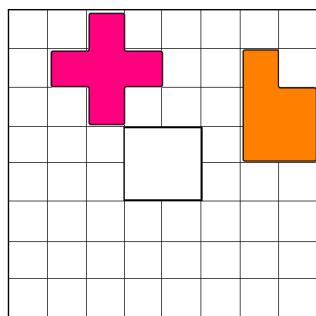
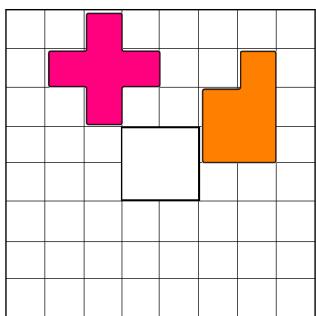
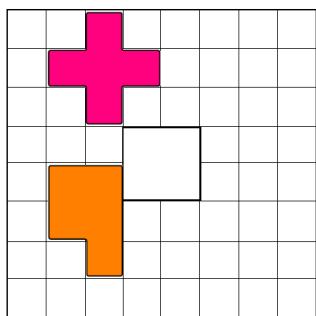
Ma quante saranno le righe di queste tre matrici? Se contassimo tutte – ma proprio tutte! – le collocazioni dei singoli 11 pezzi, anche le banalmente impossibili, come quelle parzialmente sovrapposte a X o come, ad esempio, quelle del pentamino U girato in su a occupare parte delle prime due traverse in alto della scacchiera, allora le righe delle prime due matrici sarebbero più di 1500, e oltre 1400 quelle della terza. Ecco qui sotto un esempio, in apparenza meno evidente, di collocazione impossibile.

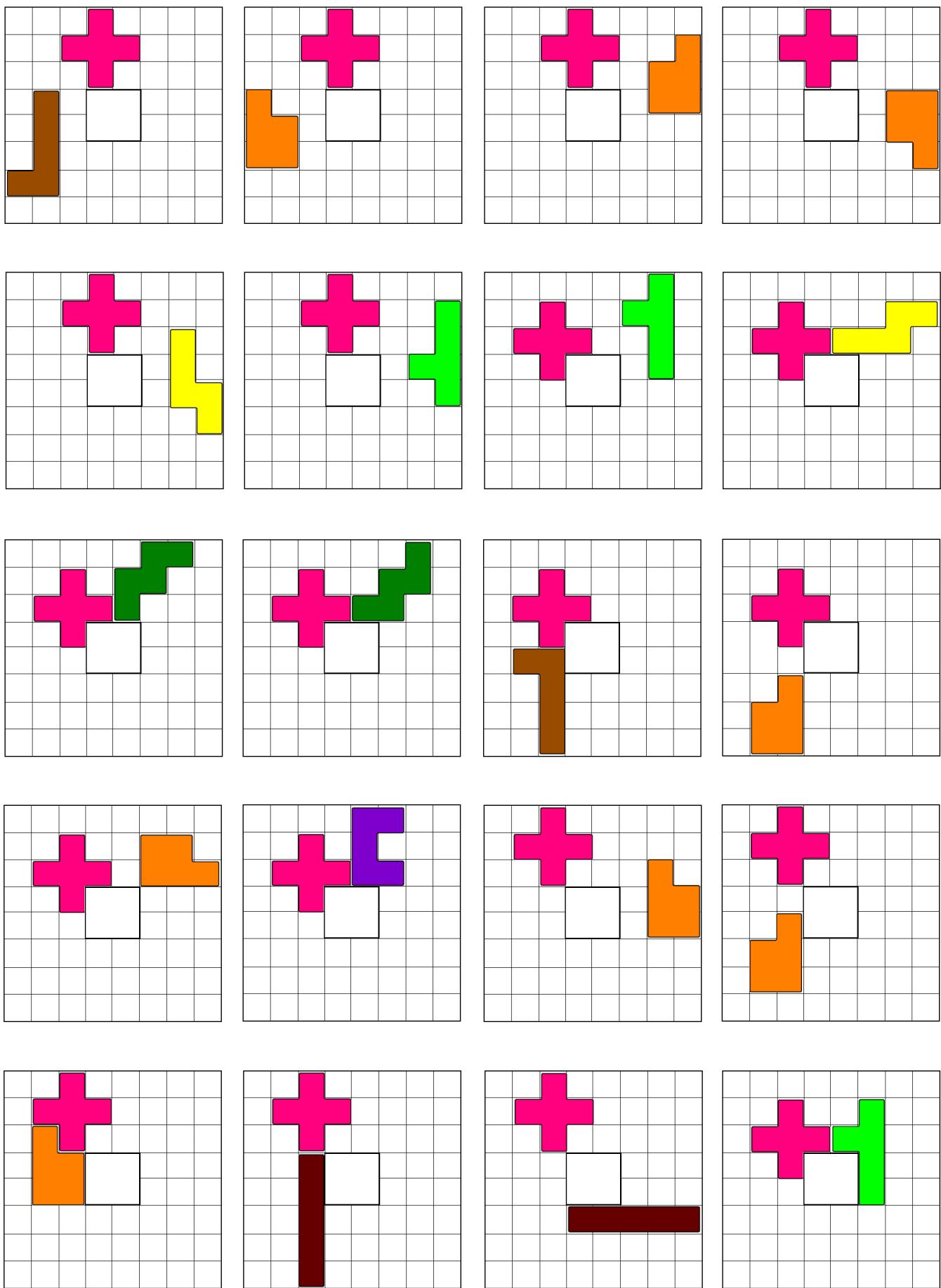
La corrispondente riga della matrice relativa al primo problema presenta, come tutte le altre, sei 1: uno nella colonna associata al pentamino N e gli altri cinque nelle colonne associate alle caselle (5, 6), (6, 6), (7, 5), (7, 6) e (8, 5) della scacchiera... ma tale riga può essere certamente scartata, poiché qui il pentamino N divide la scacchiera in due parti separate che però non hanno un numero di caselle multiplo di 5.



Una considerazione di carattere generale: talvolta può risultare conveniente cercare di ridurre – persino a mano! – la dimensione del problema, sebbene a questo scopo si debba impiegare necessariamente alquanto tempo; ciò infatti potrebbe comportare un notevole risparmio sul successivo tempo di elaborazione al computer. Al giorno d’oggi, con le attuali velocità di calcolo, questo discorso potrebbe apparire ridicolo, ma certamente non lo era una sessantina d’anni fa!

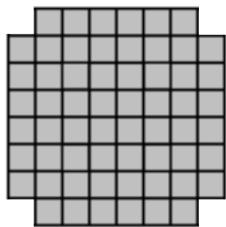
Mi son divertito a scovare le collocazioni impossibili, palesemente o meno, di un singolo pezzo, e con relativa facilità sono riuscito a ridurre le righe delle prime due matrici a poco più di un terzo, e a poco più della metà quelle della terza matrice. Di seguito sono illustrate alcune di queste “collocazioni impossibili”: sapreste motivarne l’impossibilità, sebbene in qualche caso (specialmente tra gli ultimi presentati alla pagina successiva) questa non sia affatto evidente?



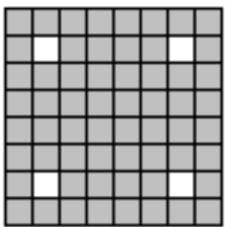


Ipotizzando che il “buco”  $2 \times 2$  possa stare ovunque sulla scacchiera  $8 \times 8$ , quindi non necessariamente al centro, le diverse soluzioni del puzzle aumenterebbero da 65 a ben 16146 (D. E. Knuth, *Dancing Links*, cit.): pertanto, le soluzioni del problema di Dudeney sono certamente meno, dovendo rispettare l’alternanza dei colori bianco e nero!

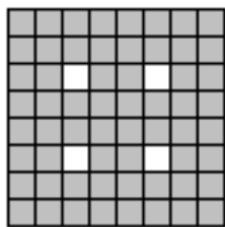
Ricordiamo anche alcune varianti con le quattro caselle vuote separate, ma disposte rispettando particolari simmetrie:



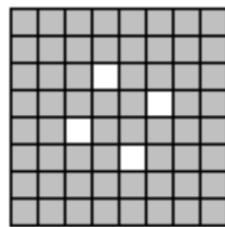
2170



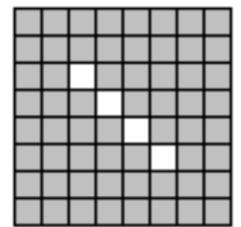
188



21



126



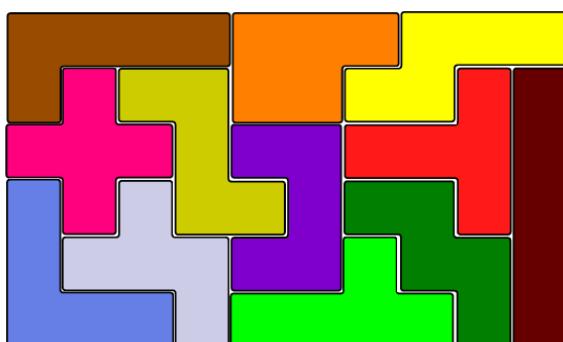
74

Sotto ciascuna di queste scacchiere di 60 caselle è riportato il numero delle possibili diverse collocazioni dei 12 pentamini su di essa.

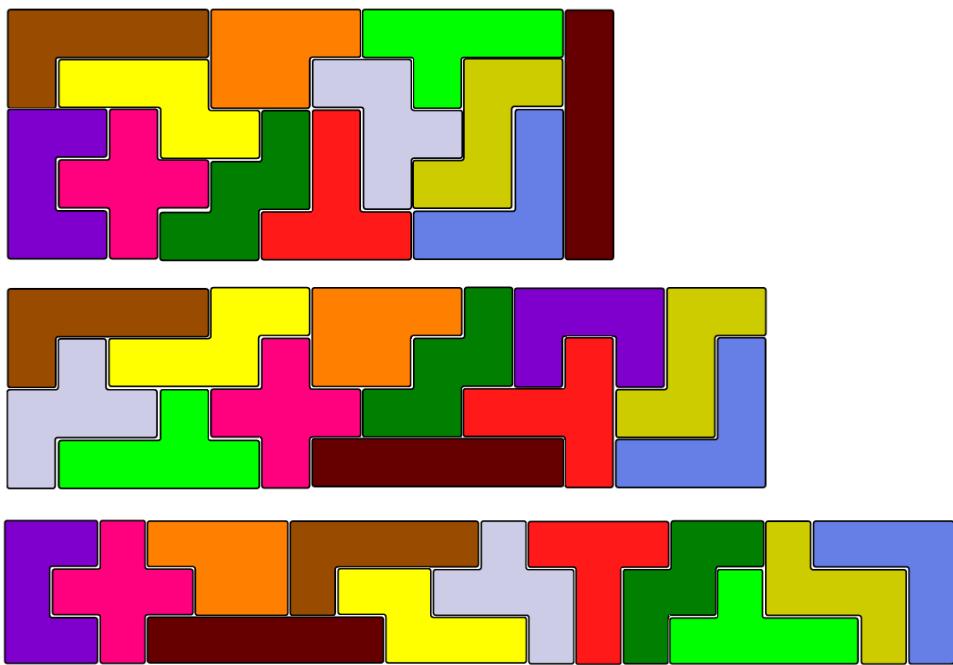
Accenniamo ad altri veri e propri puzzle con i 12 pentamini; ad esempio, comporre rettangoli di differenti formati:  $6 \times 10$ ,  $5 \times 12$ ,  $4 \times 15$ ,  $3 \times 20$ . Sempre oneroso, dal punto di vista computazionale, è il compito di trovare tutte le diverse soluzioni.

Il caso  $6 \times 10$  fu risolto per primo, in modo esauriente, da Colin Brian Haselgrove e sua moglie Jenifer nel 1960 (*A computer program for pentominoes*, Eureka, Vol. 23, No. 2, ottobre 1960, Cambridge, England: The Archimedians, pp. 16-18).

Il programma realizzato da John G. Fletcher (*A program to solve the pentomino problem by the recursive use of macros*, Communications of the ACM, Vol. 8, No. 10, ottobre 1965, pp. 621-623, <http://www.cs.virginia.edu/~skg5n/fletcher.pdf>), ottimizzato per questo particolare problema, comprendeva 765 istruzioni nel suo *loop* più interno; venne eseguito da un calcolatore IBM 7094, che aveva una frequenza di clock di 0.7 MHz e ad ogni singolo ciclo di clock accedeva a due parole di memoria di 36 bit. In circa 10 minuti trovò tutte le 2339 diverse soluzioni già calcolate dagli Haselgrove; questa è una:



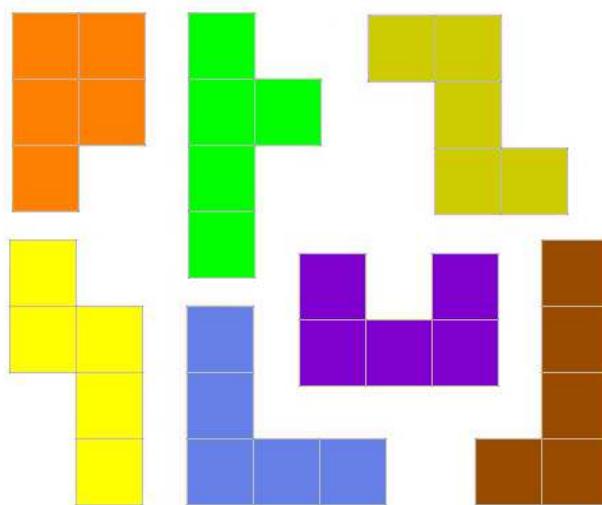
Il rettangolo  $5 \times 12$  ammette 1010 soluzioni, quello  $4 \times 15$  ne ha 368, e infine nel caso  $3 \times 20$  vi sono soltanto due modi diversi di comporre il rettangolo: l'altro, che non è raffigurato qui sotto, è disegnato nel rapporto tecnico di Scott del 1958.



Anch'io mi sono cimentato nella codifica di programmi che cercassero tutte le soluzioni di questi puzzle, ma tutt'altro che ottimizzati: procedendo con una ricerca di forza bruta e senza sfruttare simmetrie, l'esecuzione sul mio vecchio PC del 2011 ha richiesto qualche ora, addirittura 10 ore proprio nel caso del rettangolo  $3 \times 20$ .

Nel paragrafo che segue parleremo ancora, brevemente, di rompicapi con i polimini; in quello successivo vedremo invece qualche gioco, per due contendenti, in cui sono usati i pentamini. Ma per concludere simpaticamente questo paragrafo, voglio ancora ricordarvi un puzzle coi pentamini.

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				



Utilizzando quotidianamente sei dei sette pentamini raffigurati (P, Y, Z, U, N, V, L), ruotati e/o ribaltati, coprite tutte le caselle del calendario mensile che vedete a sinistra, tranne quella del giorno corrente.

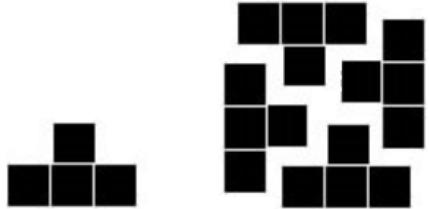
#### 4. Altri puzzle con i polimini.

Questo problema fu proposto da David A. Klarner, quand'era studente all'Università di Alberta, in Canada: dato un polimino ed eventuali sue copie, ruotabili e ribaltabili, è possibile costruire un quadrato o un rettangolo? E, in caso di risposta negativa, dimostrare l'impossibilità.

A parte i casi banali, è possibile sia col trimino sia col tetramino raffigurati qui a fianco, e sono sufficienti due copie di ciascuno, una delle quali sarà ruotata...



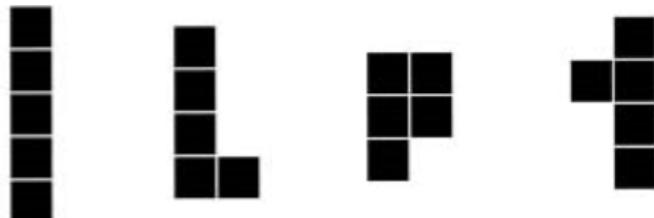
Di un altro tetramino servono invece quattro copie, tre delle quali opportunamente ruotate, come mostrato nella figura qui a destra: si riesce così a formare un quadrato  $4 \times 4$ .



Con quest'altro tetramino è invece impossibile, anche sfruttando il ribaltamento: come si può provare?



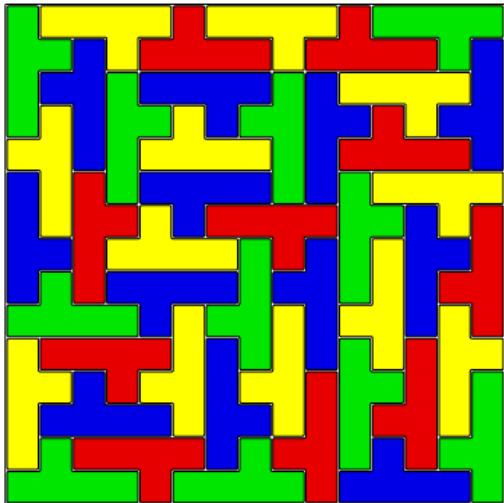
Passando ai pentamini, il problema è risolvibile soltanto con I, L, P e Y:



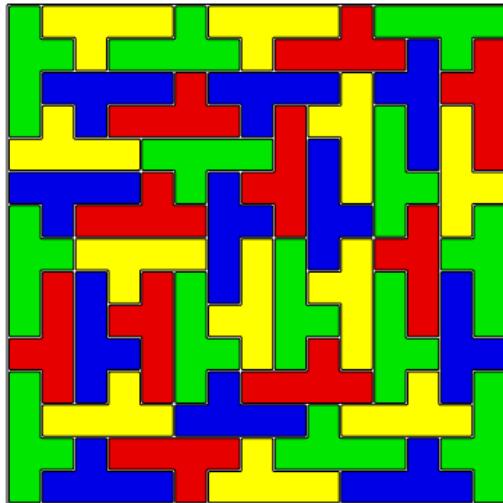
Banalmente, del pentamino I basta un solo esemplare, che è già un rettangolo, mentre sia di L sia di P sono sufficienti due copie, di cui una ruotata. Il caso assai meno ovvio è l'ultimo: del pentamino Y sono infatti necessarie almeno 10 copie, e si deve sfruttare il ribaltamento... Come può essere costruito un rettangolo  $5 \times 10$  con 10 copie di Y?

Una domanda ancor più difficile è la seguente: qual è il quadrato o rettangolo di area *minima* realizzabile con un numero *dispari* di copie di Y? La risposta è un quadrato  $15 \times 15$ , costruibile con 45 pentamini Y.

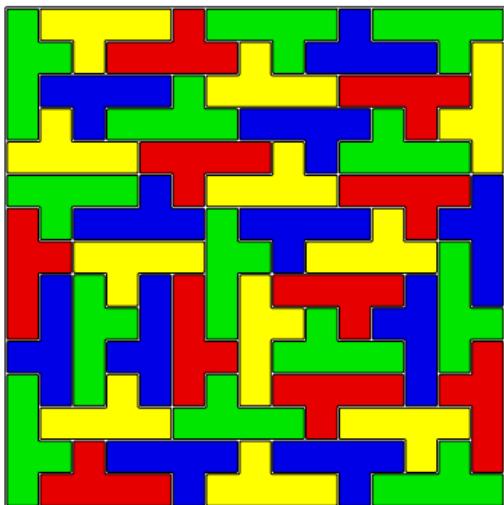
Una singola soluzione fu trovata da Jenifer (Haselgrove) Leech (*Packing a square with Y-pentominoes*, Journal of Recreational Mathematics, Vol. 7, No. 3, 1974, p. 229), mentre la lista di tutte le 212 diverse soluzioni si deve a Knuth (*Dancing Links, cit.*): nella prossima figura, ripresa dal citato articolo di Knuth, è mostrata la ripartizione delle soluzioni in quattro sottoinsiemi (di cui uno vuoto), a seconda di come possono essere collocati i quattro pentamini Y agli angoli del quadrato.



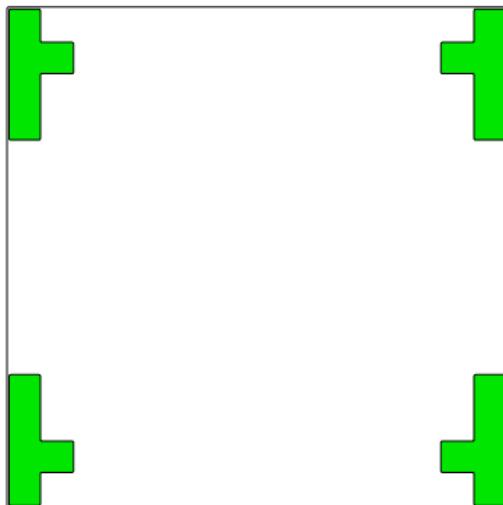
92 solutions, 14,352,556 nodes, 1,764,631,796 updates



100 solutions, 10,258,180 nodes, 1,318,478,396 updates



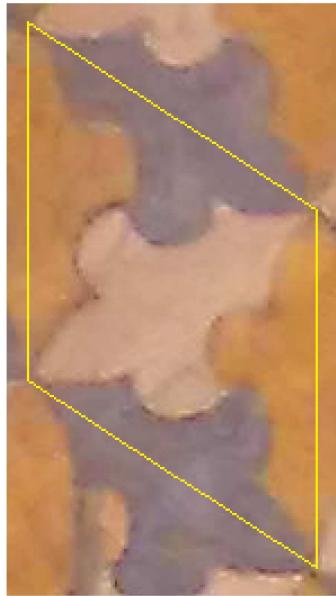
20 solutions, 6,375,335 nodes, 806,699,079 updates



0 solutions, 1,234,485 nodes, 162,017,125 updates

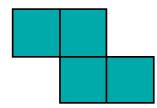
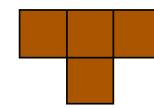
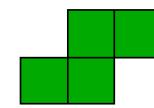
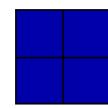
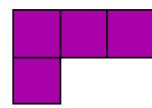
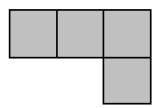
Con ognuno dei 12 pentamini si può tuttavia realizzare una tassellatura periodica del piano, disponendo di un'infinità di copie del pentamino prescelto... Tassellare il piano (infinito) significa ricoprirlo interamente, senza spazi vuoti né sovrapposizioni, usando soltanto le forme (poligonali) appartenenti a un certo insieme finito: nel nostro caso, una forma soltanto, costituita da un singolo pentamino. Dei pentamini F, T e U occorre considerare anche le copie ruotate di 180 gradi, mentre non c'è bisogno di ribaltarne alcuno: provare per credere!

Alla pagina successiva potete osservare il dettaglio di un affresco di Lorenzo e Jacopo Salimbeni, che si trova nell'Oratorio di San Giovanni a Urbino. Sebbene qui la forma usata dai due pittori non sia poligonale, l'illustrazione rende bene l'idea di tassellatura *periodica*, vale a dire che si ripete lungo due direzioni indipendenti (cioè non parallele); dunque, vi si può individuare un parallelogrammo periodico, come evidenziato nell'immagine a destra.



I tetramini ispirarono un *video game* che divenne assai popolare: il Tetris, originario dell’Unione Sovietica, dove venne progettato da Aleksej L. Pažitnov e programmato da Vadim Gerasimov nel 1984; i suoi autori non lo brevettarono, sicché si diffuse liberamente in tutto il mondo e ancor oggi si trova già predisposto nella maggior parte dei dispositivi elettronici.

I pezzi sono sette, poiché due dei cinque tetramini devono pur essere ribaltati; infatti, durante lo svolgimento del gioco, essi cadono dall’alto dello schermo, uno alla volta, e l’utente può traslarli e/o ruotarli, ma non ribalzarli.



I

J

L

O

S

T

Z

Lo scopo è il completamento di righe orizzontali di quadratini: bisogna cercare di disporre i pezzi, man mano che cadono, in modo da non lasciare spazi liberi sulle righe sottostanti. I tetramini J e L possono arrivare a completare fino a tre righe, il tetramino I anche quattro: quest’ultimo evento è chiamato *tetris*, e dà il punteggio più alto. Ogni volta che la riga più in basso è completata, essa sparisce e quelle sopra scendono.

Naturalmente sono state realizzate tante varianti di questo gioco, che differiscono tra loro in particolare per le possibilità di movimento dei pezzi durante la caduta e per l’attribuzione del punteggio alle varie situazioni che possono verificarsi nel corso di una partita.

Concludiamo questo paragrafo ricordando un bel puzzle ispirato dai polimini, noto come Fillomino (o Allied Occupation) e pubblicato, come già il Sudoku, dalla ditta giapponese Nikoli, specializzata in giochi e rompicapi logici.

Fillomino presenta una griglia di caselle, quadrata o rettangolare; alcune di esse contengono un numero. Bisogna suddividere la griglia in blocchi di caselle, ciascuna delle quali abbia un lato in comune con almeno un'altra casella dello stesso blocco; un blocco deve contenere la stessa quantità di caselle indicata dal numero riportato in ognuna delle caselle del blocco stesso. Infine, blocchi con uguale quantità di caselle non devono confinare, né in orizzontale né in verticale.

In altre parole, ciascuna casella appartiene a un polimino di  $n$  quadretti, dove  $n$  è il numero scritto nella casella; e due polimini con lo stesso numero di quadretti non possono avere lati di caselle in comune.

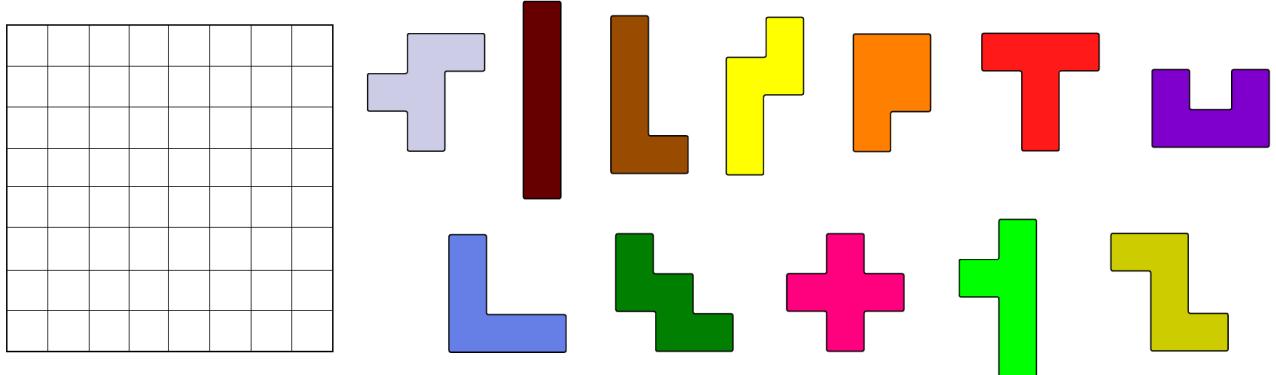
	2		8			8		5
			4	3		7		8 5
3					7			
3			5	4				4
	4	3			5	5		
			6	5			6	7
	6				5	4		
			2					5
2	4		4		3	6		
4		5			3		1	

2	2	8	8	8	8	8	8	8	5
3	5	5	4	3	3	7	4	8	5
3	4	5	4	4	3	7	4	4	5
3	4	5	5	4	1	7	7	4	5
4	4	3	3	3	5	5	7	7	5
6	6	6	6	5	5	4	6	7	3
6	6	5	5	1	5	4	6	3	3
2	4	5	2	2	4	4	6	5	5
2	4	5	4	4	3	6	6	5	5
4	4	5	4	4	3	3	6	1	5

Qui vediamo un esempio, con la sua soluzione. Ammesso che questa sia unica, è legittimo chiedersi – come nel caso del Sudoku – se tutti i numeri dati inizialmente siano necessari per l'esistenza di una sola soluzione, vale a dire se lo schema di partenza sia minimale.

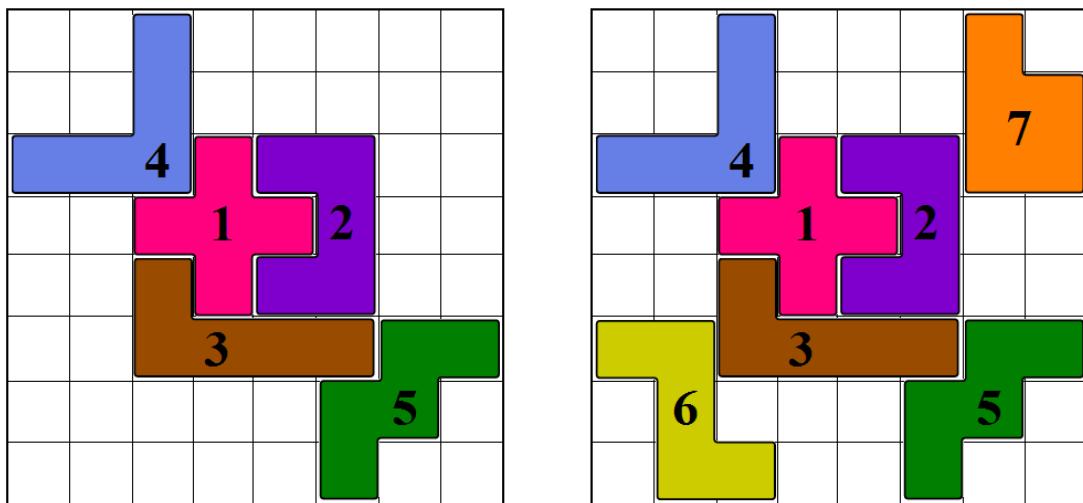
## 5. Giochi per due con i pentamini.

Per il classico gioco ideato da Golomb, occorrono una scacchiera  $8 \times 8$  e un esemplare di ciascuno dei 12 pentamini; a turno, i due giocatori scelgono uno dei pentamini rimasti e lo collocano sulla scacchiera, anche capovolto, a occupare cinque caselle ancora libere. Chi non può muovere, perché non c'è posto per alcun pentamino tra quelli rimasti o perché tutti i pentamini sono stati collocati sulla scacchiera, perde la partita.



Le partite sono brevi, da 5 a 12 mosse, e non è possibile il pareggio: è infatti un gioco *combinatorio imparziale*, a informazione perfetta (*cfr.* l'undicesimo capitolo).

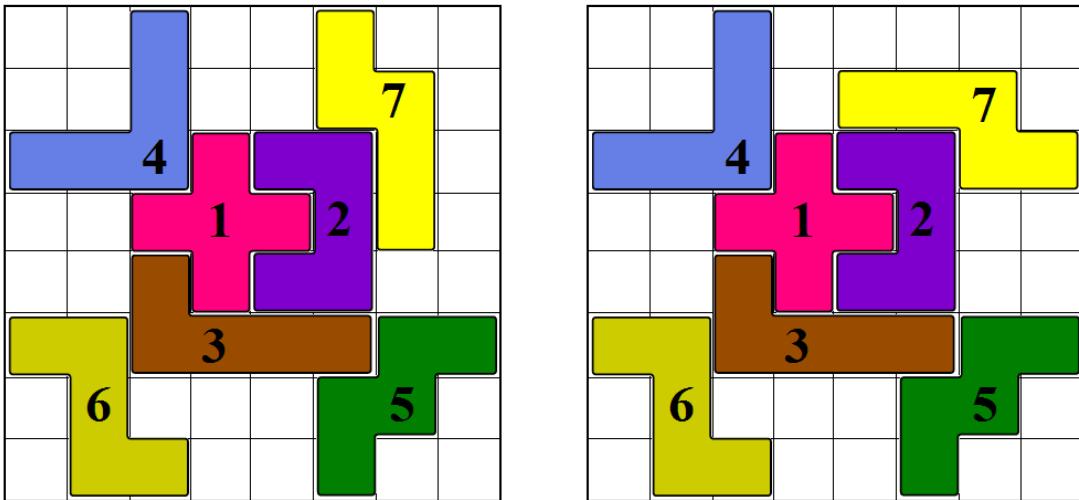
Vediamo un esempio di partita, indicando all'interno di ciascun pentamino il numero d'ordine della sua collocazione sulla scacchiera (i numeri dispari si riferiscono dunque al primo giocatore, i pari al secondo).



La mossa 4 è certamente un errore fatale per il secondo giocatore! Infatti, dopo la mossa 5, rimangono due aree, di 16 caselle ciascuna, di ugual forma...

Dopo la mossa 6, il primo giocatore avrebbe altre mosse vincenti, oltre alla 7 sopra illustrata? Si noti che, dopo la mossa 6, se il primo giocatore sceglie il pentamino P è obbligato a piazzarlo nell'area di Nord-Est, avendo ovviamente l'accortezza di non lasciare alcuna possibilità all'avversario di collocarvi un altro pezzo.

L'unica alternativa vincente per il primo giocatore è scegliere il pentamino N e sistemarlo nell'area di Nord-Est, in modo da lasciarvi spazio per il pentamino I – ma non per P, altrimenti ve lo piazzerebbe il secondo giocatore, vincendo! Ecco dunque le sole altre possibilità vincenti per il primo giocatore alla mossa 7, che prolungano la partita fino alla mossa 9:

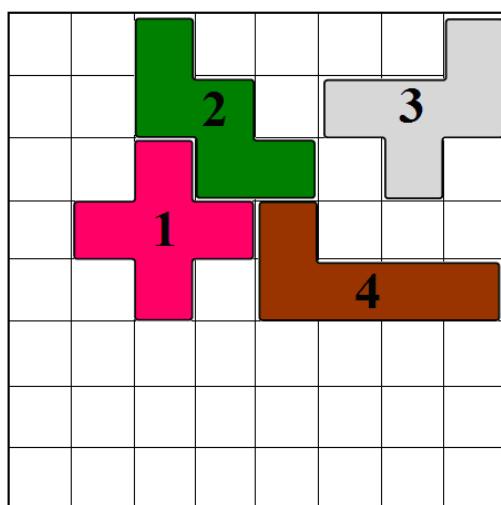


In realtà, esiste una strategia vincente per il primo giocatore.

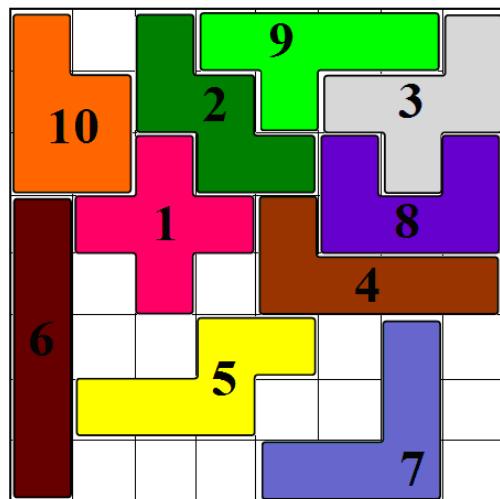
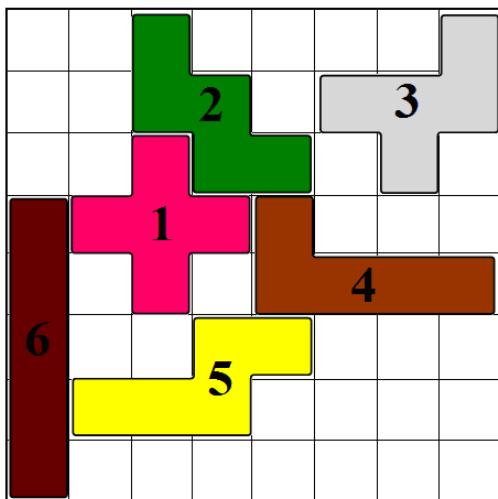
Hilarie K. Orman trovò una prima mossa vincente nel 1996, mediante due computer DEC Alpha, con processore a 64 bit e frequenza di clock 175 MHz, che nell'arco di circa due settimane esaminarono 22 miliardi di posizioni sulla scacchiera. Il gioco fu così risolto, sebbene in senso debole (*Pentominoes: A First Player Win*, University of Arizona, <http://www.msri.org/publications/books/Book29/files/orman.pdf>).

Una variante più interessante: a turno, i due giocatori scelgono uno dei pentamini rimasti, fino ad averne sei a testa; successivamente, ognuno colloca sulla scacchiera uno dei propri pentamini, a iniziare dal primo giocatore che ha scelto.

Supponiamo, ad esempio, che il primo giocatore, G1, giochi con i pentamini N, Y, V, T, F, X; il secondo, G2, con gli altri sei: P, U, L, I, Z, W; e che la partita sia giunta alla situazione mostrata qui sotto.

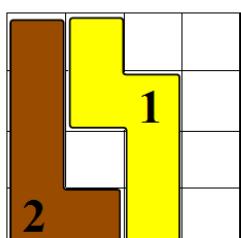


Com'è andata finora la partita? Dopo le prime due mosse, che collocano i pentamini X e W, ritenuti da Golomb "fastidiosi", con la mossa 3 G1 crea un posto adatto al suo Y, e così subito dopo G2 ne crea uno per il proprio U. Dopo la mossa 4, G1 ha ancora i pentamini N, Y, V e T, mentre a G2 rimangono P, U, I e Z.



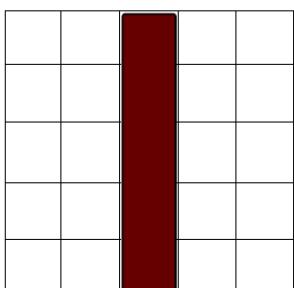
Supponendo che la partita continui come mostrato dallo schema a sinistra nella precedente figura (ove a G1 restano i pentamini Y, V e T, a G2 P, U e Z), dopo la mossa 6 G1 ha sicuramente perduto, perché può collocare ancora o V o T in basso (oltre a Y in alto a destra) ma non può impedire all'avversario di piazzare P in alto a sinistra (oltre a U nel posto precedentemente preparato). Così, oramai, se G1 colloca V, la partita non può che arrivare alla situazione raffigurata sopra a destra, in cui G2 ha ancora il pentamino Z, mentre G1 rimane col solo pentamino T e perde perché ora non può più collocarlo.

Alcune varianti prevedono diverse possibilità, quali la distribuzione casuale di sei pentamini a testa, o la partecipazione di più giocatori, ad esempio a coppie (due contro due), oppure ancora scacchiere di vari formati: qui sotto, i casi più piccoli...



Al classico gioco di Golomb, sulla scacchiera 4×4 vince il secondo giocatore.

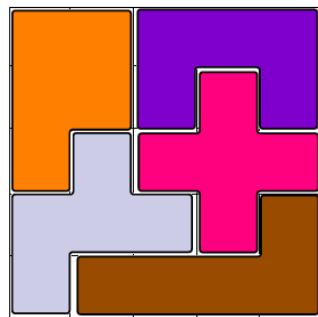
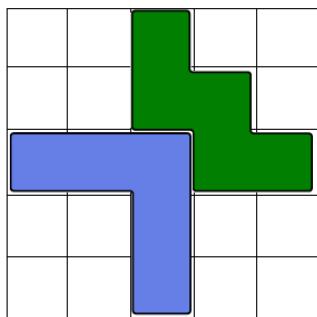
Per inciso, quello illustrato qui a sinistra è il solo caso in cui la mossa vincente per il secondo giocatore è unica.



Invece, con la scacchiera 5×5, vince il primo giocatore: banalmente, se inizia collocando il pentamino I come mostrato qui a fianco, l'avversario non ha scampo!

Anche sulla scacchiera 5×6 vince il primo giocatore...

Ecco la partita più breve e una delle più lunghe sulla scacchiera 5×5:



Provate voi a trovare le partite più brevi su scacchiere quadrate fino a 13×13...

I polimini hanno ispirato non soltanto videogiochi, come il già citato Tetris, ma pure tanti altri giochi in commercio, anche tridimensionali: i polimini solidi sono cubetti che condividono almeno una faccia...

Concludiamo l'argomento lasciando alcuni riferimenti a chi desidera approfondirlo. Il primo è a una pagina *web* curata da David Goodger, che presenta un buon numero di puzzle con i pentamini, nonché i *link* a tutte le rispettive soluzioni e ad altre pagine contenenti spiegazioni e variazioni sul tema.

<http://puzzler.sourceforge.net/docs/pentominoes.html>

Il secondo riferimento è alla pagina dedicata ai polimini nell'ambito del Progetto Polymath del Politecnico di Torino, che a sua volta comprende una ricca *sitografia*.

<http://areeweb.polito.it/didattica/polymath/htmls/probegio/GAMEMATH/Polimini/Polimini.htm>

Un piacevole libro che si può ancora trovare in commercio è *Pentomino Puzzles: 365 Teasers to Keep Your Brain in Shape*, scritto da Eric C. Harshbarger (Sterling Publishing Company, New York, 2011).

<http://www.ericharshbarger.org/>

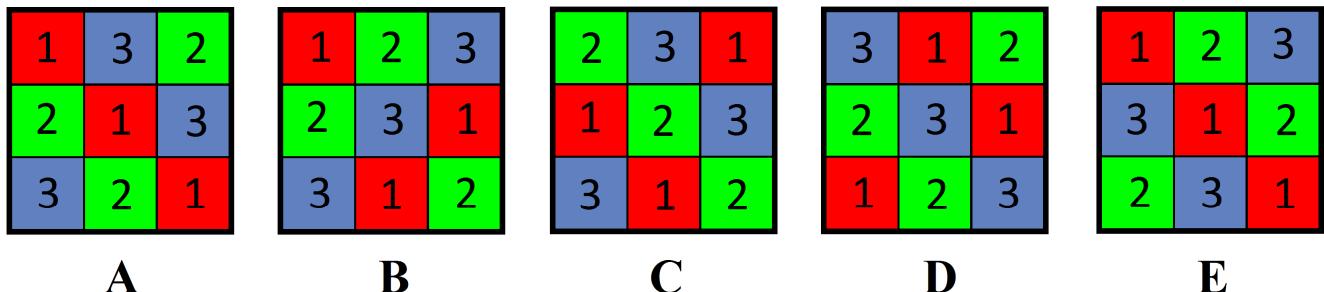
L'ultimo riferimento è invece a un libro di Stewart T. Coffin, assai stimolante e liberamente accessibile in rete, *The Puzzling World of Polyhedral Dissections*, dove potrete trovare moltissimi spunti di riflessione da un'incredibile varietà di puzzle, in due e in tre dimensioni.

<http://www.johnrausch.com/PuzzlingWorld/>

## 6. Quadratini colorati.

Come anticipato, vorrei proporre alcuni quesiti riguardanti le simmetrie dei quadrati latini  $3 \times 3$  e degli schemi di Shidoku, il Sudoku  $4 \times 4$ ; iniziamo con i quadrati latini.

Leonardo e Arturo stanno giocando con tanti quadratini di tre colori; essi hanno composto questi cinque quadrati, in cui su ciascuna riga e su ciascuna colonna vi sono un quadratino rosso, uno verde e uno blu:



Leonardo osserva che uno soltanto di questi cinque quadrati *non* può essere ottenuto da qualcuno degli altri quattro semplicemente mediante una *permutazione di colori*.

Una permutazione di colori consiste in uno o più scambi di colore; ad esempio: sostituire il rosso con il verde e viceversa; oppure sostituire il rosso con il verde, il verde con il blu e il blu con il rosso (ciò che si può ottenere in due passi: scambiare tra loro il rosso e il verde, poi il rosso e il blu).

Suggerite ad Arturo qual è il quadrato notato da Leonardo.

**Soluzione.** È il quadrato B. Infatti, ad esempio:

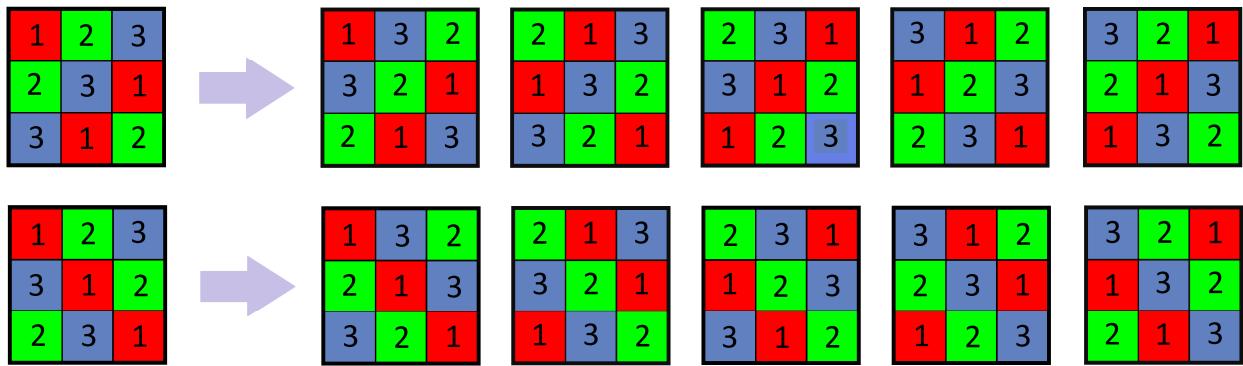
- A può essere ottenuto da E scambiando il verde col blu, e quindi E può essere ottenuto da A mediante lo stesso scambio;
- C può essere ottenuto da E scambiando rosso-verde-blu con verde-blu-rosso;
- D può essere ottenuto da E scambiando rosso-verde-blu con blu-rosso-verde.

Notate che B può essere ottenuto da E scambiando la seconda riga con la terza.

I quadrati latini di ordine  $n$  (quadrati  $n \times n$  dove gli  $n$  simboli distinti, qui colori o cifre, compaiono in ciascuna riga e in ciascuna colonna) furono studiati da Eulero nella seconda metà del '700 e ripresi un secolo dopo dal matematico inglese Arthur Cayley.

I quadrati latini di ordine 3 che presentano le cifre 1, 2, 3 in prima riga sono soltanto due; permutando le cifre, da ciascuno di essi se ne ottengono altri cinque, come potete vedere nella figura in alto alla pagina successiva. In effetti, i quadrati latini di ordine 3 sono 12, che è il prodotto di 2 per il fattoriale di 3.

Passando all'ordine 4, i quadrati latini che presentano le cifre 1, 2, 3, 4 in prima riga sono 24, e dunque i quadrati latini di ordine 4 sono 576, cioè il prodotto di 24 per il fattoriale di 4; si vedano in OEIS le sequenze A000479 e A002860, rispettivamente.



Gli schemi finali di Shidoku sono particolari quadrati latini di ordine 4: infatti, i quattro simboli (colori o cifre) devono essere presenti anche in ciascuno dei quattro riquadri  $2\times 2$  in cui lo schema è suddiviso.

Gary e Ugo si divertono a comporre schemi finali di Shidoku, usando quadratini di quattro colori; essi hanno composto finora questi sei schemi:

1	2	3	4
3	4	1	2
2	1	4	3
4	3	2	1

A

1	2	3	4
3	4	1	2
2	3	4	1
4	1	2	3

B

1	2	3	4
4	3	2	1
2	4	1	3
3	1	4	2

C

1	2	3	4
4	3	2	1
3	1	4	2
2	4	1	3

D

1	2	3	4
3	4	1	2
4	3	2	1
2	1	4	3

E

1	2	3	4
4	3	2	1
3	4	1	2
2	1	4	3

F

Gary nota che ciascuno degli schemi C, D, E, F può essere trasformato in *uno e uno soltanto* degli schemi A e B, eseguendo *eventualmente più volte* qualcuna delle operazioni di seguito elencate:

- scambiare tra loro due colori
- ribaltare lo schema lungo l'asse orizzontale
- ruotare lo schema di 90 gradi in senso orario
- scambiare le righe 1 e 2, o le righe 3 e 4, o le righe 1-2 con le righe 3-4
- *idem* per le colonne.

Tenendo presente che, in ognuno dei casi qui contemplati, al più tre delle operazioni elementari sopra elencate sono necessarie, e non più di una volta ciascuna, aiutate Ugo a trovare gli esatti abbinamenti notati da Gary.

**Soluzione.** Gli schemi C e D possono essere trasformati in B, gli schemi E ed F in A. Infatti, ad esempio:

- scambiando le colonne 3 e 4 di C e poi scambiando blu e giallo, si ottiene B;
- scambiando le righe 3 e 4 di D, si ottiene C (e quindi, procedendo come sopra, si ottiene B);
- scambiando le righe 3 e 4 di E, si ottiene A;
- partendo da F ed eseguendo la sequenza di operazioni: scambio delle righe 3 e 4, scambio delle colonne 3 e 4, scambio dei colori blu e giallo, si ottiene A.

Continuando il loro gioco, Gary e Ugo compongono altri quattro schemi:

1	2	3	4
3	4	2	1
2	1	4	3
4	3	1	2

G

1	2	3	4
3	4	2	1
4	3	1	2
2	1	4	3

H

1	2	3	4
4	3	1	2
2	1	4	3
3	4	2	1

I

1	2	3	4
4	3	1	2
3	4	2	1
2	1	4	3

K

E proprio come prima, ognuno di questi può essere trasformato in uno e uno soltanto degli schemi A e B; tuttavia, qui il procedimento è un poco più laborioso: occorrono infatti da tre a sei operazioni elementari fra quelle precedentemente elencate.

**Soluzione.** Tutti e quattro gli schemi si trasformano in B. Infatti, ad esempio:

- partendo da G ed eseguendo la sequenza di operazioni: ribaltamento lungo l'asse orizzontale, rotazione di 90 gradi in senso orario, scambio dei colori verde e blu, si ottiene B;
- scambiando le righe 3 e 4 di H, si ottiene G (e quindi, procedendo come sopra, si ottiene B);
- partendo da I ed eseguendo la sequenza di operazioni: scambio delle colonne 1 e 2, ribaltamento lungo l'asse orizzontale, rotazione di 90 gradi in senso orario, scambio dei colori rosso-verde-blu con blu-rosso-verde (che, come detto, si realizza mediante la sequenza di due scambi di colore), si ottiene B;
- scambiando le righe 3 e 4 di K, si ottiene I (e quindi, procedendo come sopra, si ottiene B).

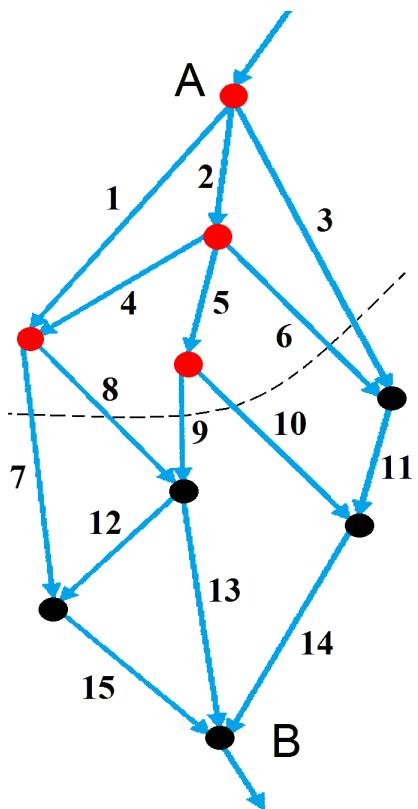
Avrete certamente notato che i dieci schemi composti da Gary e Ugo presentano le cifre 1, 2, 3 e 4 ordinate nella prima riga: ai nostri amici ne mancano soltanto due (trasformabili in B: quali?) con tale caratteristica. In effetti, permutando i colori, gli schemi finali di Shidoku sono in tutto  $12 \cdot (4!) = 288$  (si veda la sequenza A107739 in OEIS): la metà dei quadrati latini in questo caso del tutto particolare.

Rispetto alle suddette operazioni elementari, applicabili anche più volte, i 288 schemi si ripartiscono in due classi di equivalenza soltanto: quella di A e quella di B (sequenza A109741 in OEIS). Questi risultati sono stati ottenuti indipendentemente da Gary McGuire e Hugo van der Sanden.

## 7. La linea più ampia e la minima copertura di un insieme.

In questo paragrafo e nel prossimo, illustrerò alcuni classici problemi su grafi.

Nel 2013 il gruppo Bebras dei Paesi Bassi propose un quesito – ripreso l'anno successivo per la nostra gara Kangourou – che coinvolgeva una struttura alquanto rilevante in parecchi contesti informatici, e nota ai nostri lettori già dal primo capitolo: un *grafo orientato privo di cicli*, e per di più *planare*.



Il testo racconta che una comunità di castori desidera ispezionare il fiume, e tutti i tratti in cui il fiume si ramifica devono essere perlustrati: per ognuno dei 15 tratti, bisogna dunque che passi almeno un castoro. Evitando di nuotare contro corrente, i castori-ispettori partiranno dal punto A e si ritroveranno nel punto B, e dunque, non potendo tornare indietro, ciascuno di loro non passerà mai due volte per uno stesso tratto. Qual è il minimo numero di castori necessari per fare l'ispezione?

Poiché i castori non vogliono nuotar contro corrente, il modello di fiume con diramazioni da loro adottato è appunto un grafo orientato. Onde percorrere tutti i tratti di fiume (gli archi del grafo), i castori devono organizzarsi in modo che per ogni diramazione (nodo del grafo) passino almeno tanti di loro quanti sono i tratti uscenti da quella diramazione.

Osserviamo che quattro castori devono scendere da A lungo il tratto centrale (numerato 2 in figura); non possono esser di meno, perché alla prima diramazione che incontrano ne occorrono almeno due che possano proseguire ancora nel tratto centrale (numerato 5) per poi dividersi (uno lungo il tratto 9, l'altro sul 10). Pertanto da A devono partire almeno sei castori: uno scende lungo il tratto 1, un altro sul 3, e i restanti quattro lungo il tratto centrale. Possiamo constatare facilmente che, per come proseguono i rami del fiume, questi sei castori sono anche sufficienti allo svolgimento del compito.

Il quesito presentato è un caso assai particolare del problema del taglio massimo, o più propriamente dell'*insieme di taglio massimo*. Dato un grafo del tutto generico, dividiamo i suoi nodi in due sottoinsiemi (e questo è un taglio) e consideriamo gli archi che collegano un nodo del primo sottoinsieme con un nodo del secondo (e questo è un insieme di taglio, *cut-set* in inglese); notate che l'ordine dato ai due sottoinsiemi è importante quando il grafo è orientato, e in tal caso si considerano o i soli archi che vanno da un nodo del primo sottoinsieme a un nodo del secondo o soltanto gli insiemi di taglio formati esclusivamente da archi con tale proprietà.

Il problema in generale, che consiste dunque nel determinare un insieme di taglio costituito dal maggior numero possibile di archi, appartiene alla classe degli NP-ardui, per i quali – ribadiamo – non si conosce e non si sa nemmeno se esista un algoritmo risolutivo efficiente, ossia con complessità polinomiale rispetto al tempo. Sui grafi *pesati*, ove a ciascun arco è assegnato un peso positivo, si può cercare un insieme di taglio che abbia il peso (dato dalla somma dei pesi degli archi che ad esso appartengono) massimo; tale problema, che coincide col precedente quando i pesi sono tutti unitari, è detto MAX-CUT ed è, in generale, NP-arduo (in senso forte, come il problema del commesso viaggiatore ed altri già incontrati: a meno che  $P = NP$ , non esiste alcun algoritmo pseudo-polinomiale che lo risolva esattamente). Nella sua versione decisionale, esso è l'ultimo della lista dei 21 problemi NP-completi elencati da Karp nel 1972 (<https://people.eecs.berkeley.edu/~luca/cs172/karp.pdf>).

Nei grafi *bipartiti* (quelli in cui è possibile dividere i nodi in due sottoinsiemi in modo tale che *ogni* arco abbia un estremo in un nodo di uno dei due sottoinsiemi e l'altro estremo in uno dei nodi dell'altro sottoinsieme, a prescindere dall'eventuale orientamento) la soluzione del problema è immediata (e per verificare se un grafo è bipartito è sufficiente una visita in ampiezza). Ma anche per i grafi *planari* (quelli che si possono disegnare su un foglio senza incrociare archi, proprio come accade nel nostro quesito) sono stati ideati algoritmi efficienti (F. Hadlock, 1975).

Su grafi orientati si parla di MAX-DICUT (DI sta per *directed*), e la complessità non cambia, nemmeno nel caso di grafo aciclico. Vi sono algoritmi di approssimazione tempo-polynomiali, con fattore costante piuttosto buono non soltanto per MAX-CUT (1.139: M. X. Goemans e D. P. Williamson, 1995), ma anche per MAX-DICUT (1.165: U. Feige e M. X. Goemans, 1995, o persino 1.1442: M. Lewin e altri, 2002; il limite teorico è  $12/11$ , se  $P \neq NP$ : J. Håstad, 2001), mentre i migliori algoritmi esatti hanno complessità esponenziale nel numero di archi o nodi, con base  $< 1.74$  ( $e > 1$ ).

Invece, i problemi di “taglio minimo” (MIN-CUT) sono risolubili efficientemente; in particolare, se pensiamo ad esempio a una rete di trasporto, il massimo flusso che può passare dalla sorgente al pozzo egualgia proprio la somma delle capacità degli archi appartenenti al taglio minimo. Perciò non si confonda il problema del taglio massimo con quello del *flusso massimo* (cfr. “Una vacanza in Grecia”, pp. 20-22): quest’ultimo si pone soltanto su particolari grafi orientati, che sono appunto le reti di trasporto, e può risolversi in ogni caso (anche non planare) in tempo polinomiale.

In un grafo *orientato e privo di cicli*, esiste almeno un nodo *sorgente* (senza archi entranti) e almeno un nodo *pozzo* (senza archi uscenti). Nel caso del quesito, il grafo ha una sola sorgente, il nodo di partenza A, e un solo pozzo, il nodo di arrivo B; vi si dovranno considerare i tagli che abbiano A nel primo sottoinsieme, B nel secondo, e nessun arco che vada da un nodo del secondo sottoinsieme a un nodo del primo.

Nella figura alla pagina precedente è già stato tratteggiato uno dei due insiemi di taglio massimi (i nodi del primo sottoinsieme sono colorati in rosso, in nero quelli del secondo); provate a tracciare una linea “ugualmente ampia” per individuare l’altro insieme di taglio massimo!

Ora complicheremo inutilmente il problema proposto: ciò ci permetterà, tuttavia, una riflessione di un certo interesse, come vedremo. Poiché gli insiemi di taglio massimi sono costituiti da sei archi, saranno necessari sei cammini diversi (vale a dire che differiscono a due a due per almeno un arco) affinché ciascun arco sia percorso da almeno un castoro. In effetti, il nostro quesito può essere trasformato in un'istanza del *problema della copertura minima di un insieme* (*minimum set cover*, in inglese), che in generale è NP-arduo, proprio come trovare l'*exact cover set* di cui abbiamo parlato: ora non importa se un elemento è coperto più volte, ma quel che conta è scegliere quanti meno possibili insiemi la cui unione formi l'insieme “universo”.

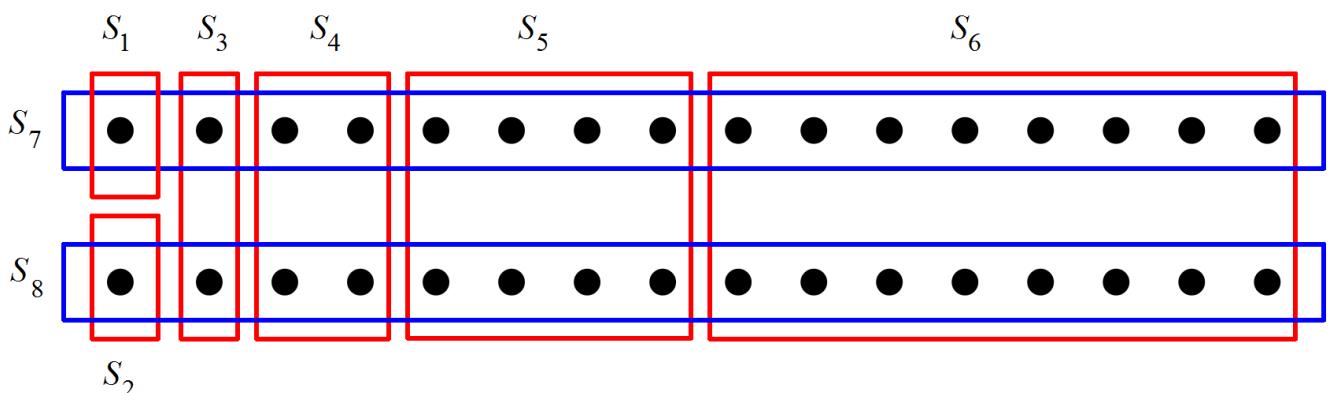
Contare tutti i possibili cammini da A a B è facile: poniamo  $c(B) = 1$  e, per ogni altro nodo  $u$ , andando a ritroso, calcoliamo  $c(u) = \text{somma dei } c(v) \text{ per ogni arco } (u, v)$ . Alla fine,  $c(A)$  sarà il numero dei cammini; per elencarli tutti basterà applicare lo stesso schema di calcolo (a ritroso), che è quello della *programmazione dinamica* (cfr. “Un percorso a stadi successivi”, pp. 18-20).

Trovare tutti i cammini in un grafo del genere è una questione importante in informatica: ha a che fare, ad esempio, con la preparazione dei *piani di collaudo* di un programma e con lo *scheduling di task* (con vincoli di precedenza e possibilità di elaborazioni parallele). Nel caso in esame, vi sono 11 cammini diversi da A a B:

[1, 7, 15]	[1, 8, 12, 15]	[1, 8, 13]	[2, 4, 7, 15]
[2, 4, 8, 12, 15]	[2, 4, 8, 13]	[2, 5, 9, 12, 15]	[2, 5, 9, 13]
[2, 5, 10, 14]*	[2, 6, 11, 14]*	[3, 11, 14]*	

Dobbiamo dunque trovare il minimo numero di cammini (ciascuno dei quali è costituito da un sottoinsieme di archi) che ricopra tutti e 15 gli archi. Notiamo che gli ultimi tre cammini (contrassegnati da \*) sono necessari, poiché sono i soli che comprendono gli archi 10, 6 e 3, rispettivamente. Tra i rimanenti, per completare la copertura, è possibile scegliere una fra quattro diverse terne (a voi scovarle!): in ogni caso, dunque, i cammini (e quindi i castori) necessari sono 6.

Come si può affrontare, in generale, il problema di *minimum set cover*? L'approccio *greedy* consiste semplicemente nella scelta, ad ogni passo, dell'insieme che copre il maggior numero di elementi non ancora coperti. Provate ad applicarlo al caso qui sopra. Considerando un universo  $U$  di  $2^k$  elementi, con  $k$  grande a piacere, si può però costruire un esempio *ad hoc*, che rivela chiaramente il limite di tale approccio.



Per illustrarlo, consideriamo la precedente figura, dove i 32 elementi di  $U$  sono ripartiti negli insiemi mutuamente disgiunti  $S_1, \dots, S_6$ , rispettivamente di 1, 1, 2, 4, 8, 16 elementi (segnati in rosso); inoltre, alla famiglia di insiemi data appartengono anche gli insiemi  $S_7$  e  $S_8$ , di 16 elementi ciascuno, costruiti come mostrato in figura (ove sono segnati in blu). Al primo passo, l'algoritmo *greedy* può scegliere tra  $S_6, S_7$  e  $S_8$ , ciascuno dei quali copre 8 elementi; supponiamo che scelga  $S_6$ , e così pure, ai passi successivi, le sue scelte cadano man mano su  $S_5, S_4, S_3, S_2$  e  $S_1$ , sicché infine gli insiemi scelti siano sei, quando invece ne bastano due:  $S_7$  e  $S_8$ . Generalizzando, con  $2^k$  elementi, nel caso peggiore sarebbero scelti  $k + 1$  insiemi, anziché 2.

Con un po' di pazienza, è possibile modificare l'esempio, in modo che la scelta sia unica ad ogni passo, ma le cose continuino ad andar male nella stessa maniera; e si può anche dimostrare che peggio di così sostanzialmente non può andare, per cui il rapporto di approssimazione dell'algoritmo *greedy* tende a crescere, col numero di elementi di  $U$ , al più come il suo logaritmo (naturale): ben lungi dall'essere costante! Per inciso, è stato proprio lo studio dei problemi di *set covering* a portare allo sviluppo di tecniche basilari nell'intero campo degli *algoritmi di approssimazione* tempo-polynomiali, usando i quali si ha una misura di quanto la soluzione trovata (rapidamente) possa essere lontana dalla migliore, che rimane sconosciuta.

Per risolvere però *esattamente* un'istanza qualsiasi di *minimum set cover* dobbiamo affidarci a un algoritmo di complessità tempo-esponenziale; gli attuali migliori sono basati sull'approccio *branch-and-bound* (cfr. pp. 114-119), abbinato ad altre tecniche sofisticate, e impiegano un tempo dell'ordine di  $c^{n+m}$ , dove  $c$  è una costante di poco inferiore a 1.24,  $n$  è il numero di elementi di  $U$ , e  $m$  il numero di insiemi considerati. Qui ci limitiamo a delineare un possibile procedimento, non troppo raffinato.

Data una famiglia  $F$  di insiemi  $S_1, \dots, S_m$  (finiti) tali che la loro unione sia uguale all'insieme universo  $U$  (dunque il problema è risolubile), chiamiamo  $C$  la copertura minima che costituirà la soluzione, inizialmente vuota. Anzitutto, *finché è possibile*, applichiamo l'una o l'altra delle seguenti *regole di riduzione*:

- se, nella famiglia  $F$ , si verifica che un insieme  $S_i$  è l'unico a possedere un certo elemento di  $U$ , allora tale insieme deve far parte di qualsiasi copertura (si ricordino, ad esempio, i cammini marcati con \* alla pagina precedente), e quindi aggiungiamo  $S_i$  alla soluzione  $C$ , togliendo tutti i suoi elementi sia da  $U$  sia dagli altri insiemi della famiglia  $F$ ;
- se, nella famiglia  $F$ , si verifica che un insieme  $S_i$  è contenuto (anche non strettamente) in un altro insieme  $S_j$ , allora eliminiamo l'insieme  $S_i$  dalla famiglia  $F$ , poiché certamente c'è una copertura minima a cui  $S_i$  non appartiene.

Si osservi che eventuali insiemi in  $F$ , che abbiano un solo elemento, necessariamente rientrano in uno dei casi previsti da queste regole; inoltre, se dopo un'applicazione della prima regola un insieme della famiglia  $F$  rimane vuoto, sarà poi eliminato da un'applicazione della seconda.

(A questo punto, se tutti gli insiemi rimasti in  $F$  hanno due soli elementi, allora una soluzione ottima può essere calcolata con un apposito algoritmo tempo-polonomiale,

che trovi il cosiddetto *maximum matching*, ma qui – senza perdita di generalità – non prevediamo di trattare a sé questo caso particolare.)

Siano  $U'$  ed  $F'$  l'universo e la famiglia di insiemi dopo le eventuali riduzioni fatte.

Se  $F'$  è vuota, una soluzione è già stata costruita applicando le regole di riduzione, e quindi il procedimento termina con soluzione  $C$ .

Altrimenti, tra gli insiemi in  $F'$ , si sceglie un  $S'_k$  con il maggior numero di elementi, dopodiché si eseguono (possibilmente in parallelo) due passi:

- si ipotizza che  $S'_k$  appartenga a una soluzione ottima, e quindi si pone in una nuova copertura  $C_1$ ; si costruisce una nuova istanza del problema, che abbia come universo l'insieme  $U'$  privato degli elementi in  $S'_k$  e come famiglia la famiglia  $F'$  tolto l'insieme  $S'_k$  e tolti gli elementi di  $S'_k$  dagli altri suoi insiemi; si risolve dunque ricorsivamente il nuovo problema, aggiungendo la sua soluzione a  $C_1$ ;

- si ipotizza, contrariamente, che  $S'_k$  non appartenga ad alcuna soluzione ottima; si costruisce una nuova istanza del problema, che abbia come universo  $U'$  (vi si lasciano gli elementi che ci sono) e come famiglia la famiglia  $F'$  privata di  $S'_k$  (poiché per l'ipotesi fatta tale insieme è superfluo, ma comunque il problema avrà una soluzione in virtù delle applicazioni “a esaurimento” delle regole di riduzione); si risolve (ricorsivamente) anche questo problema, e sia  $C_2$  la soluzione calcolata.

Infine, confrontiamo le soluzioni così ottenute: se  $C_1$  contiene meno insiemi di quanti ne contiene  $C_2$ , aggiungiamo a  $C$  tutti gli insiemi in  $C_1$ , altrimenti aggiungiamo a  $C$  tutti gli insiemi in  $C_2$ . La soluzione  $C$  così confezionata è una copertura di  $U$ , ottenuta col minimo numero di insiemi appartenenti alla famiglia  $F$ .

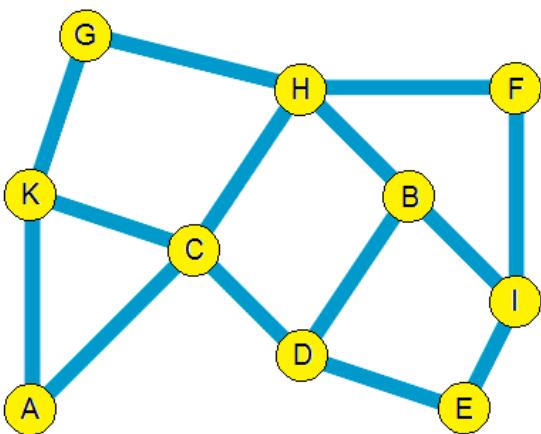
Una nota conclusiva: un altro modo di affrontare il quesito dell’ispezione fluviale è quello di formularlo come problema di *programmazione lineare intera*, in cui le variabili (aumentate di 1) sono le quantità di castori che percorrono ciascun arco e i vincoli imposti sono le equazioni di bilancio ai nodi. Se  $x_i + 1$  è il numero di castori che percorrerà il tratto  $i$ , bisognerà *minimizzare* il numero di castori che partono dal punto A, e dunque  $x_1 + x_2 + x_3$ , con i vincoli:

$$\begin{array}{ll|ll} x_1 + x_2 + x_3 - x_{13} - x_{14} - x_{15} & = 0 & x_5 - x_9 - x_{10} & = 1 \\ x_1 + x_4 - x_7 - x_8 & = 0 & x_7 + x_{12} - x_{15} & = -1 \\ x_2 - x_4 - x_5 - x_6 & = 2 & x_8 + x_9 - x_{12} - x_{13} & = 0 \\ x_3 + x_6 - x_{11} & = -1 & x_{10} + x_{11} - x_{14} & = -1 \end{array}$$

e  $x_i \geq 0$  per ogni  $i = 1, \dots, 15$ . La prima equazione stabilisce che tutti i castori che partono da A giungano in B, le successive esprimono il bilancio (tanti castori entrano quanti escono) per ognuno degli altri sette nodi. I termini noti sono interi e inoltre la matrice del sistema è totalmente unimodulare; in queste particolari circostanze, si verifica che la soluzione ottima è intera e inoltre il problema è risolvibile in modo efficiente, cioè in tempo polinomiale, come si poteva supporre, date le caratteristiche della semplice istanza qui considerata.

## 8. L'insieme dominante, la copertura per nodi e l'insieme indipendente.

Nel 2016 il gruppo Bebras della Svizzera propose un quesito riconducibile con facilità a un problema di *minimum set cover*; giustificheremo quest'affermazione.



Lo spunto era dato dalla solita comunità di castori che qui deve decidere in quali dei dieci punti allestire tre presidî di pronto soccorso, in modo tale che da ciascuno degli altri punti sia possibile raggiungere un pronto soccorso nuotando per un solo tratto di canale.

Questa volta il modello della rete di canali qui raffigurata è un *grafo non orientato*, i cui archi rappresentano i tratti di canale, percorribili in entrambi i sensi, mentre i nodi rappresentano i punti dove due o più tratti di canale si incontrano.

Si tratta dunque di scegliere tre nodi (ove collocare i presidî), in modo tale che ciascuno degli altri sette nodi sia collegato con un arco ad almeno uno di questi tre. Sarà possibile? Vale a dire: il problema proposto ammetterà almeno una soluzione? Questa, anzitutto, è la formulazione del *problema decisionale* e, salvo casi banali, per rispondere affermativamente... occorre trovare una soluzione!

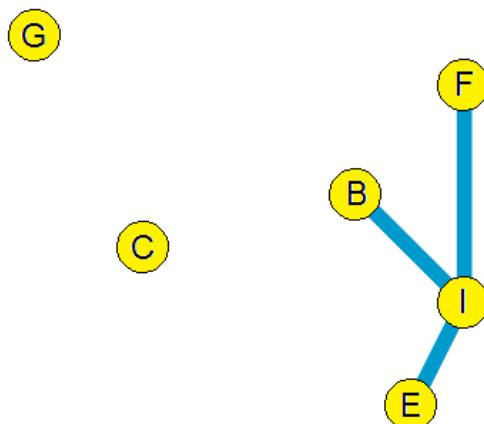
Le possibili combinazioni di tre nodi sono 120 (quanti i diversi sottoinsiemi che si possono formare scegliendo 3 elementi da un insieme di 10, detto altrimenti: il coefficiente binomiale 10 su 3), ma in questo caso un procedimento *greedy* ci sarà d'aiuto, e non saremo costretti a esaminarle tutte per ottenere una soluzione.

Possiamo iniziare considerando i nodi ai quali giungono più archi (quelli, come si dice, di *grado maggiore*), nel nostro caso C e H (abbiamo nominato i dieci nodi con altrettante lettere dell'alfabeto, come mostrato nel precedente disegno).

Proviamo a partire con C.

Scartiamo quindi i nodi ad esso collegati (A, D, H e K), poiché da questi si può arrivare a C percorrendo appunto un solo arco, e così pure tutti gli archi collegati a tali nodi, poiché non sono più rilevanti: otteniamo dunque il disegno qui a fianco.

Dovremo poi includere I (che permette di essere raggiunto da B, E ed F, i quali si possono scartare) e infine G (che è rimasto isolato). Pertanto, una soluzione è data dall'insieme di nodi {C, I, G}. Ovviamente possiamo fermarci qui: abbiamo trovato una risposta al nostro quesito!



Volendo invece cercare altre soluzioni, in modo analogo possiamo partire con H e scartare i nodi ad esso collegati (B, C, F e G), nonché gli archi che in questi hanno un estremo; dovremo poi includere E (collegato a D e I, da scartare) e infine o A o K. Pertanto, altre due soluzioni sono {H, E, A} e {H, E, K}.

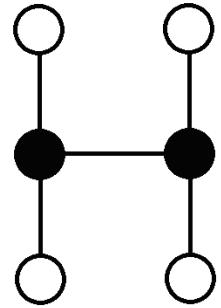
Se partissimo con *entrambi* i nodi C e H, scartando i nodi collegati all'uno o all'altro (A, B, D, F, G e K) e includendo uno dei due rimanenti (E e I), troveremmo ancora due soluzioni: {C, H, E} e {C, H, I}. Da queste si capisce che possono esservi, naturalmente, delle soluzioni che comprendono due nodi collegati da un arco.

Per il particolare problema proposto esistono altre tre soluzioni: {D, F, K}, {B, I, K} e {C, I, K}, di cui le ultime due contengono due nodi collegati da un arco, e l'ultima è quella che forse permette più opportunità. Il problema ammette dunque 8 soluzioni, e per trovarle *tutte* è stato necessario esaminare tutte (o quasi) le combinazioni!

Considerare dapprima i nodi di grado maggiore è una regola *euristica*, ovvero una regola di decisione dettata dal buon senso; ciò non basta però, in generale, a trovare una soluzione costituita dal numero di nodi prestabilito, ammesso che vi sia. Ma si può affermare che questa regola si traduce in un algoritmo *greedy* che individua un insieme di nodi *minimale*: se ne togliamo uno qualsiasi, non è più vero che ciascun altro nodo è collegato con un arco ad almeno uno dei nodi rimasti nell'insieme.

Chiaramente, tale algoritmo non trova sempre un insieme *minimo*, cioè costituito dal minor numero possibile di nodi.

Valga come semplice controsenso il caso illustrato qui a fianco: all'unica soluzione minima appartengono i due nodi neri, collegati da un arco, e quindi l'algoritmo *greedy*, che prima ordina i nodi per grado discendente e poi considera un nodo alla volta, non trova tale soluzione; trova, invece, una delle due soluzioni minimali (simmetriche) con tre nodi.



Un risultato significativo è che l'algoritmo *greedy* trova un insieme formato da non più di  $n + 1 - \sqrt{2m + 1}$  nodi, dove  $n$  è il numero di nodi e  $m$  il numero di archi del grafo (A. K. Parekh, 1991); ma è importante notare che non si tratta di un algoritmo di approssimazione, perché non dà alcuna garanzia su quanto il risultato fornito sia distante dall'ottimo.

Il quesito proposto è un'istanza del *problema dell'insieme dominante* (*dominating set*, in inglese), la cui versione decisionale è la seguente: dato un grafo non orientato  $G$  e dato un numero intero positivo  $k$ , si deve stabilire se esiste un insieme  $D$ , costituito da (al più)  $k$  nodi di  $G$ , tale che ciascun altro nodo di  $G$  sia collegato con un arco ad almeno uno dei nodi di  $D$ . Esso appartiene alla classe dei problemi NP-completi (e il corrispettivo problema di ottimizzazione, che consiste nel determinare *una* soluzione  $D$  col *minimo* numero di nodi, appartiene alla classe dei problemi NP-ardui): nei casi peggiori, si rischia di dover tentare tutte (o quasi) le combinazioni, o perlomeno, usando gli algoritmi esatti più avanzati, si deve prevedere un tempo di elaborazione dell'ordine di  $c^n$ , dove  $n$  è il numero di nodi di  $G$ , e  $c$  è una costante che – a seconda dei metodi impiegati – sta tra 1.26 (precisamente  $2^{1/3}$ ) e 1.53.

Le applicazioni in informatica riguardano ad esempio il calcolo distribuito, allorché si debba suddividere una rete in un piccolo numero di *cluster* locali.

Se consideriamo l'impegno di tempo richiesto per risolvere esattamente i due problemi di minimo che abbiamo testé incontrato, ossia il *minimum dominating set* e il *minimum set cover*, possiamo affermare che sono parimenti difficolosi.

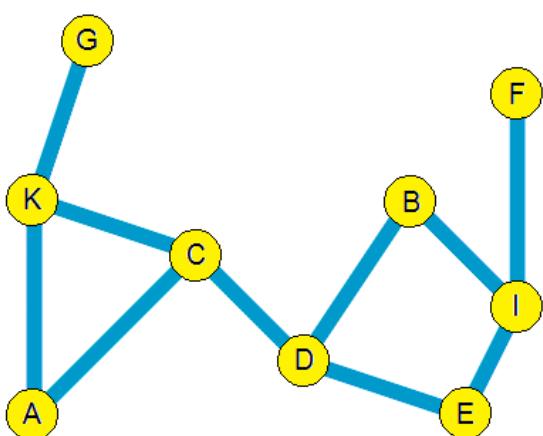
Mostriamo come il primo si riduce al secondo: molto semplicemente, ad ogni nodo associamo l'insieme di nodi costituito dal nodo stesso e dai suoi vicini, raggiungibili percorrendo un arco; una volta ricoperto col minor numero di tali insiemi l'insieme di tutti i nodi del grafo, basterà considerare – come soluzione del *minimum dominating set* – i nodi a cui sono associati gli insiemi usati per la copertura. Lascio al lettore il compito – un poco più impegnativo! – di ricondurre, in generale, il *minimum set cover* al *minimum dominating set*.

Oltre che per dimostrare l'equivalenza – quanto a onere computazionale – di due (o più) problemi mediante riduzione efficiente dall'uno all'altro, in informatica spesso può tornare comodo trasformare un problema in un altro (ad esso equivalente o più generale), per risolvere il quale già si dispone di un software apposito; resta inteso che poi bisogna sapere anche come interpretare (facilmente) i risultati per ottenere la soluzione del problema originario!

Rimanendo nello stesso contesto, propongo ora un altro quesito.

Nel loro villaggio, schematizzato nel disegno in alto a pagina 341, i castori pianificano la costruzione di torrette d'avvistamento. Dato che da ciascun punto si possono controllare tutti i tratti di canale che hanno un'estremità in quel punto, qual è il numero minimo di punti nei quali installare una torretta d'avvistamento, affinché tutti i tratti di canale siano sorvegliati?

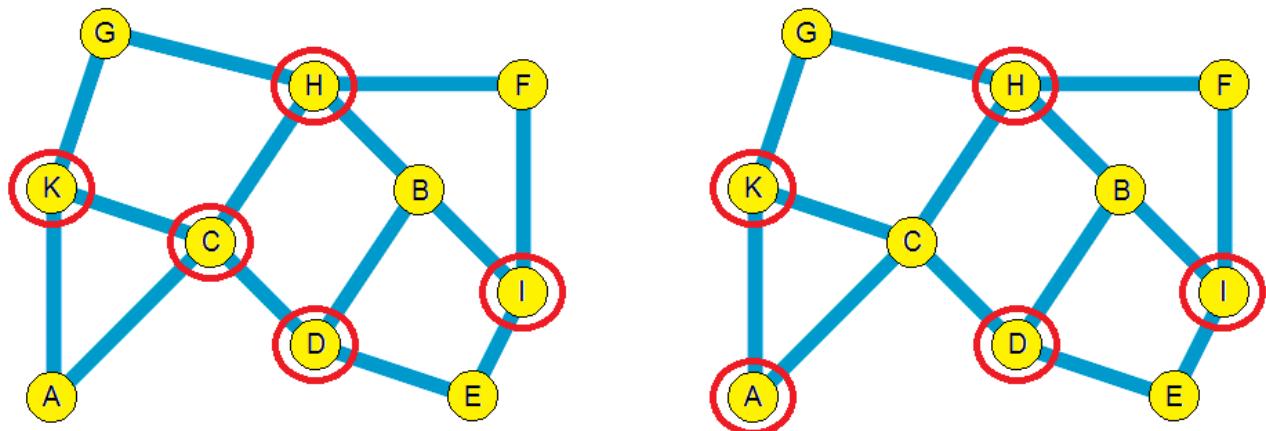
Tale questione è una semplice istanza di un altro problema di minimizzazione: quello della *copertura per nodi* (*minimum vertex cover*), anch'esso, in generale, NP-arduo. Vediamo se ci è d'aiuto un algoritmo *greedy* che si affidi, anche in questo caso, ai nodi di grado maggiore. Al primo passo, abbiamo due alternative: C o H; partiamo con H, lo aggiungiamo all'insieme "copertura" (al momento vuoto) e lo togliamo dal grafo insieme con gli archi che hanno un estremo in esso.



Al secondo passo, le possibilità sono quattro: K, C, D, I (nodi di grado 3). Se, ad esempio, aggiungiamo C alla copertura, togliendolo dal grafo con gli archi su di esso incidenti, è facile constatare che le scelte successive cadranno necessariamente su I, D e K (i nodi rimasti isolati si scartano).

Se al secondo passo scegliessimo D, poi dovremmo prendere I e K, e infine o C (ottenendo la stessa copertura di prima) o A.

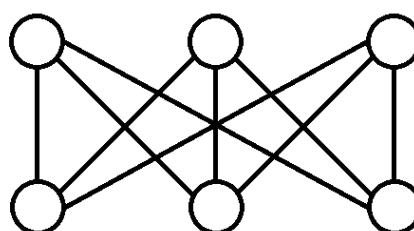
Abbiamo così trovato due coperture diverse, entrambe costituite da 5 nodi, che in effetti sono le sole ottime, cioè minime.



Ma se inizialmente avessimo scelto C (anziché H) e poi, al secondo passo, B (di grado 3), non saremmo giunti a una soluzione ottima. Come potevamo aspettarci, avendo premesso che il problema è in generale NP-arduo, questo algoritmo *greedy* non sempre funziona bene. Non solo: esso non è nemmeno un algoritmo di approssimazione con fattore costante, in quanto, nei casi peggiori, il rapporto tra il numero di nodi di una copertura trovata seguendo il procedimento testé illustrato e il numero di nodi di una copertura minima tende comunque a crescere, seppure in modo logaritmico, all'aumentare del numero di nodi del grafo; analogamente a quanto già visto per il *minimum set cover*, non è difficile – ma neppur banalissimo – individuare esempi di casi pessimi, questa volta su grafi (bipartiti) opportunamente costruiti.

Esiste un altro algoritmo *greedy*, all'apparenza più ingenuo: partiamo con la copertura vuota e, finché ci sono archi, scegliamo (a caso) un arco, diciamo quello tra i due nodi  $u$  e  $v$ , aggiungiamo sia  $u$  sia  $v$  alla copertura, e rimuoviamo dal grafo entrambi i nodi,  $u$  e  $v$ , e ogni arco incidente su  $u$  o su  $v$ .

Questa idea si basa sul fatto che, per ogni arco del grafo, almeno uno dei due nodi da esso collegati appartiene certamente a una copertura minima! Di conseguenza, nel caso peggiore, la copertura generata da questo secondo algoritmo *greedy* è grande il doppio della minima: possiamo concludere che è un algoritmo di approssimazione con fattore costante 2. Ed è facile costruire un esempio dove sicuramente prendiamo il doppio dei nodi necessari: basta considerare un grafo  $K_{n,n}$  (bipartito, con  $n$  nodi in ciascuna delle due parti, ogni nodo di una parte collegato con un arco a ciascun nodo dell'altra parte), le cui coperture minime contengono  $n$  nodi (quelli di una parte o quelli dell'altra); in figura, il caso  $n = 3$ , ove l'algoritmo dà una copertura di 6 nodi.



Per inciso, il teorema di Kuratowski, citato a pagina 105, stabilisce che *ogni* grafo non planare contiene, come *sottografo*,  $K_{3,3}$  o un grafo completo di 5 nodi,  $K_5$ ; questi ultimi sono non planari: se provate a disegnarli, pur prestando attenzione, non potrete fare a meno di incrociare l'ultimo arco con uno di quelli già tracciati!

Se poi vogliamo l'esempio più semplice, poniamo  $n = 1$ .

C'è ancora almeno un altro algoritmo di approssimazione tempo-polonomiale con fattore costante 2 (più complicato), ma non si sa se esistano con fattori costanti inferiori; certamente non inferiori a 1.36, se  $P \neq NP$  (I. Dinur e S. Safra, 2002).

Occorre precisare che, su qualsiasi grafo bipartito, il problema di *minimum vertex cover* è risolvibile efficientemente. Vi sono comunque esempi di grafi (non bipartiti) sui quali gli algoritmi *greedy* tendono a dare la loro prestazione peggiore.

Se applichiamo il secondo algoritmo *greedy* al problema dei castori, non abbiamo speranza di arrivare a una soluzione ottima; anzi, se siamo sfortunati, prendiamo tutti i nodi. Potremmo anche pensare, in generale, di abbinare i due criteri: scegliere un arco non "a caso", ma con la massima somma dei gradi dei due nodi estremi. Su un grafo  $K_{n,n}$  le cose però non cambierebbero affatto, perciò grazie all'accorgimento adottato potremmo attenderci prestazioni migliori soltanto statisticamente, non certo nei casi peggiori. Ad esempio, il problema dei castori sarebbe risolto con una copertura di 8 nodi, anziché i 5 della minima, ma si otterrebbe la risposta ottima (fornita anche dal primo algoritmo *greedy*) sul semplice grafo di pagina 342.

Il fatto che, scegliendo comunque un arco, almeno uno dei due nodi estremi farà di certo parte di una copertura minima, ci suggerisce però un procedimento – semplice da descrivere – per ottenere una soluzione ottima: anziché prevedere soltanto di prendere entrambi i nodi estremi, si può *anche* calcolare che cosa accade prendendo o solo il primo o solo il secondo di essi, tenendo presente che, quando si prende uno solo dei due, si devono per forza prendere nel contempo anche tutti i nodi adiacenti all'altro. Quindi, per risolvere un problema, si devono risolvere ricorsivamente – con lo stesso procedimento – tre sottoproblemi, alle cui soluzioni andranno poi aggiunti, rispettivamente, entrambi i nodi estremi dell'arco considerato, oppure solo il primo dei due più tutti gli altri nodi adiacenti al secondo, oppure solo il secondo dei due più tutti gli altri nodi adiacenti al primo. Naturalmente, fra le tre coperture così calcolate, si sceglierà quella col minor numero di nodi.

Si può dimostrare che, se  $k$  è il numero di nodi della copertura minima, la profondità a cui giunge il procedimento ricorsivo è, al più,  $k$ , e quindi la complessità rispetto al tempo presenterà un fattore  $3^k$ .

Il problema di *minimum vertex cover* è un caso particolare di *minimum set cover*: infatti, per ridurre il primo al secondo basta considerare, per ogni nodo del grafo, l'insieme degli archi su di esso incidenti e, come insieme "universo", l'insieme di tutti gli archi del grafo.

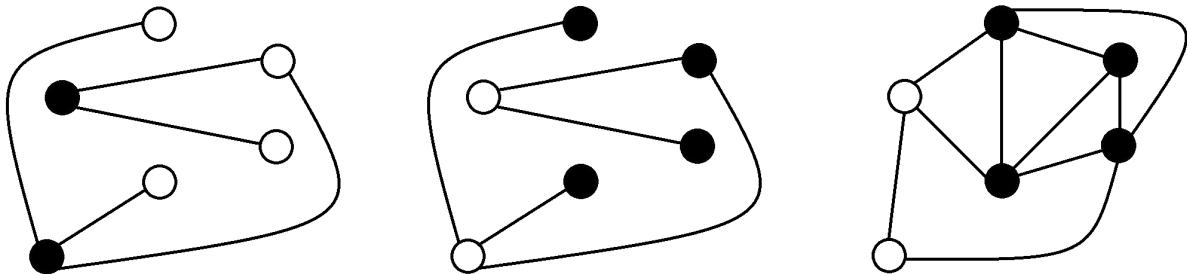
Inoltre, se consideriamo il complemento di una minima copertura per nodi, rispetto all'insieme di tutti i nodi del grafo, otteniamo una soluzione di un altro interessante problema di ottimizzazione: trovare il *massimo insieme indipendente* (cfr. p. 103).

Ad esempio, se i castori del villaggio raffigurato in alto a pagina 341 si chiedessero quale sia il numero massimo di punti (indicati dai cerchi) che, a due a due, *non* sono collegati da un tratto di canale, avendo già trovato le minime coperture per nodi, la risposta sarebbe immediata: 5. E per individuare tali punti basterebbe considerare i nodi *non* cerchiati nella figura in alto a pagina 344, che costituiscono i due massimi insiemi (di nodi) indipendenti: {A, G, B, E, F} e {G, C, B, E, F}.

In generale, detto  $I$  un sottoinsieme dei nodi di un grafo,  $I$  è un insieme indipendente se e soltanto se (sse) ogni arco è incidente, al più, su un nodo di  $I$  sse ogni arco è incidente su almeno un nodo non appartenente a  $I$  sse il complemento di  $I$  è una copertura per nodi. Quindi un *massimo* insieme indipendente è complemento di una *minima* copertura per nodi, e i due problemi di ottimizzazione (in inglese, *maximum independent vertex set* e *minimum vertex cover*) sono ugualmente impegnativi dal punto di vista computazionale.

Rammentiamo quanto già detto a pagina 103 a proposito del grafo complementare: dato un grafo  $G$ , non orientato e privo di cappi, il grafo complementare di  $G$  ha gli stessi nodi di  $G$ , nessuno degli archi di  $G$ , e tutti gli archi (tra due nodi diversi) che *non* sono in  $G$ . Avevamo anche osservato (a p. 103) che un insieme indipendente è costituito dai nodi di un sottografo completo (*clique*) nel grafo complementare, e pertanto ciascuna soluzione di un problema di *maximum independent vertex set* è anche soluzione del problema di *maximum clique* sul grafo complementare, e viceversa: si parla dunque di problemi equivalenti.

Nella figura che segue, da sinistra a destra, sono evidenziate (con i nodi in nero) le soluzioni (complementari) di *minimum vertex cover* e *maximum independent vertex set* su un piccolo grafo con 6 nodi, e infine la soluzione (uguale alla precedente) del problema di *maximum clique* sul grafo complementare.



Nel suo studio del 1972, Karp ridusse il problema di decidere se un grafo  $G$  con  $n$  nodi ha (almeno)  $k$  nodi mutuamente adiacenti al problema di decidere se il grafo complementare di  $G$  ha una copertura per nodi costituita da non più di  $n - k$  nodi, e anche la riduzione inversa è immediata.

Ragioniamo ancora un poco su come trovare un insieme indipendente massimo in un grafo qualsiasi. Se un nodo  $v$  appartiene a un insieme indipendente  $I$ , allora nessuno dei nodi adiacenti a  $v$  può essere in  $I$ . D'altro canto, supponendo che  $I$  sia massimo (e dunque massimale), qualora  $v$  non fosse in  $I$ , vi dovrebbe essere almeno uno dei suoi vicini; in conclusione, o  $v$  o almeno uno dei suoi vicini appartiene a  $I$ .

Ciò suggerisce che, per risolvere il nostro problema, si possano risolvere sue istanze ridotte, per poi prendere la migliore; nella fattispecie, si spera di contenere il numero di tali istanze, scegliendo di volta in volta un nodo col grado minore possibile.

Riepilogando, un algoritmo ricorsivo, che applichi questo metodo *branch-and-reduce* per risolvere esattamente il problema di *maximum independent vertex set*, può essere così delineato:

- costruisce un insieme di nodi  $I$ , inizialmente vuoto;
- se nel grafo vi sono nodi isolati, li aggiunge a  $I$ , togliendoli dal grafo;
- se vi è almeno un nodo nel grafo, ne sceglie uno di grado minimo: sia  $v_0$  il nodo scelto e siano  $v_1, \dots, v_m$  i suoi vicini ( $m > 0$ ); quindi risolve ricorsivamente  $m + 1$  sottoproblemi: per  $i = 0, \dots, m$ , costruisce un nuovo grafo, copia dell'attuale, da cui elimina  $v_i$  e tutti i suoi vicini (nonché gli archi ad essi collegati), e trova un insieme indipendente massimo  $I_i$  per questo nuovo grafo; infine, unisce a  $I$  il più grande (o uno dei più grandi) tra questi  $I_i$ , aggiungendovi anche il corrispondente nodo  $v_i$ ;
- restituisce  $I$ .

Come potete constatare, è lo stesso principio che ci ha portato agli algoritmi delineati per i problemi di *minimum set* e *minimum vertex cover*: risolvere, ricorsivamente e possibilmente in parallelo, sottoproblemi di dimensione ridotta, e poi confezionare una soluzione utilizzando una di quelle dei sottoproblemi che presenti determinate caratteristiche.

Un algoritmo per risolvere esattamente il problema di *maximum clique* in un grafo qualsiasi, la cui complessità rispetto al tempo è di ordine  $1.286^n$ , dove  $n$  è il numero dei nodi del grafo, fu descritto da Robert E. Tarjan in un rapporto tecnico alla Cornell University nel 1972. Chiaramente, seguendo l'idea ovvia che consiste nell'esaminare ciascun sottoinsieme di nodi, verificare se è una *clique*, e infine scegliere la massima *clique* trovata, la complessità sarebbe di ordine  $n \cdot 2^n$ , poiché  $2^n$  sono i possibili sottoinsiemi di nodi; pur partendo dai più grandi, nei casi sfortunati si arriverebbe ai più piccoli. L'algoritmo di Tarjan fu esteso e migliorato dall'autore stesso e da Anthony E. Trojanowski, per trovare un insieme indipendente massimo; la complessità risultò di ordine  $2^{n/3}$  (circa  $1.26^n$ ). Il rapporto tecnico che presentava il loro lavoro, dato alla Stanford University nel giugno del 1976, si concludeva con queste parole:

[...] even for NP-complete problems it is sometimes possible to develop algorithms which are substantially better in the worst case than the obvious enumeration algorithms. Whether the algorithm presented here can be improved substantially, and whether similar algorithms can be developed for other NP-complete problems, are open questions.

In effetti, per il problema di *maximum independent set* sono stati ideati algoritmi di pochissimo migliori, ad esempio uno di complessità  $1.2114^n$  (N. Bourgeois e altri, 2010)... e comunque, per fortuna, i “casi peggiori” non capitano molto spesso!

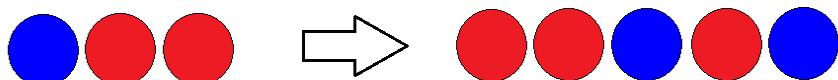
Concludiamo questi ultimi impegnativi paragrafi con una considerazione riassuntiva sugli algoritmi di approssimazione, ai quali abbiamo fin qui accennato: oltre a fornire una soluzione di un dato problema di ottimizzazione NP-arduo, che però non è detto sia una delle migliori, un algoritmo di approssimazione garantisce un certo limite al rapporto (maggiore di 1) tra la soluzione da esso trovata e quella ottima non nota, nel caso si tratti di minimizzazione (il rapporto inverso, in caso di massimizzazione). Naturalmente, noi siamo interessati ad algoritmi di approssimazione efficienti, ossia, al più, tempo-polynomiali. Ammesso che sia vero  $P \neq NP$ , possiamo così riassumere alcuni risultati finora ottenuti:

- per certi problemi, come quello del commesso viaggiatore (TSP) che abbiamo incontrato nel quarto capitolo, non è possibile alcun rapporto di approssimazione finito.
- Per il problema di *maximum independent vertex set*, il rapporto di approssimazione nel caso peggiore è, al più, di ordine  $n$ , essendo  $n$  il numero di nodi del grafo, e non può essere abbassato.
- Per il problema di *minimum set cover*, il rapporto di approssimazione nel caso peggiore è di ordine  $\ln(n)$ , essendo  $n$  il numero di elementi dell'insieme “universo”. Per la precisione, con l'algoritmo *greedy* del caso non pesato di cui abbiamo parlato, tale rapporto è di ordine  $\ln(m)$ , se  $m$  sono gli elementi del sottoinsieme più grande nella famiglia data, e la soluzione trovata sarà costituita da non più di  $\ln(n) - \ln(k) + 1$  insiemi della famiglia data, se  $k$  sono quelli di una soluzione ottima.
- Per alcuni problemi, un rapporto di approssimazione costante è possibile, ma ha un limite inferiore (maggiore di 1) difficile da individuare: le relative dimostrazioni di “non approssimabilità” sono annoverate tra i risultati più raffinati in questo settore dell’informatica. Abbiamo accennato, ad esempio, al caso del *minimum vertex cover*, per il quale si conoscono algoritmi con fattore 2, ma non inferiore, ed è stato provato che non si può scendere al di sotto di 1.36: potrebbe trattarsi effettivamente di un problema per cui è “difficile” approssimare meglio la soluzione!
- Per il problema di MAX-CUT è facile trovare un algoritmo con fattore 2, anche nel caso pesato (lo lascio al lettore come utile esercizio), ma abbiamo citato l’esistenza sia di algoritmi più precisi, sia di un limite inferiore.
- Infine, per i problemi “più fortunati” nella classe degli NP-ardui, l’approssimabilità non ha un limite inferiore maggiore di 1, in quanto esistono algoritmi con rapporto di approssimazione arbitrariamente prossimo a 1 (per eccesso). In questo gruppo di problemi ce n’è uno di cui abbiamo diffusamente parlato nel quarto capitolo: quello dello zaino (KNAPSACK).

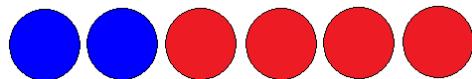
## 9. Giocando con le biglie.

Abbandonando i problemi “difficili”, in questo paragrafo propongo alcuni giochetti addirittura “indecidibili” – s’intende in generale e per via algoritmica – ma vi assicuro che qui una soluzione c’è: potrete trovarne una col ragionamento e poi confrontarla con quella che ho scritto subito appresso.

Iniziamo con questo: supponete di avere una riserva illimitata di biglie, sia rosse sia blu, e di metterne in fila un certo numero, a vostro piacere. Poi, per continuare, avete una sola regola, qui sotto illustrata: una sequenza di biglie come quella a sinistra (se è presente nella fila) sarà sostituita con la sequenza a destra (che, notate, è più lunga); dovete procedere finché sarà possibile fare qualche sostituzione.



Se partite da questa fila di biglie:



quale fila otterrete alla fine?

Esisterà qualche fila di biglie partendo dalla quale il procedimento non termina?

Se ora modifichiamo la nostra regola nella seguente (notate la “piccola” differenza):



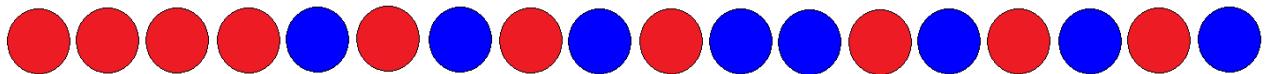
sapreste trovare una fila di biglie – possibilmente di lunghezza minima – partendo dalla quale il procedimento non termina?

Prima di passare alle risposte, accenniamo alle importanti nozioni di informatica nascoste sotto questo giochetto.

Un *sistema di riscrittura di stringhe* (storicamente *semi-sistema di Thue*, da Axel Thue; cfr. p. 91) è costituito da un insieme finito di simboli e da un insieme finito di regole che descrivono tutte le ammissibili trasformazioni di *sequenze finite di simboli*, le stringhe per l’appunto. Per fare un semplice esempio, di certo familiare a tutti, consideriamo le espressioni aritmetiche da “ridurre” a un singolo numero, proposte in tanti esercizi alle scuole medie. Ad ogni passo di riscrittura, una sotto-espressione è sostituita con un’altra – di solito però più breve! – ad essa equivalente, sicché il valore (ossia il significato) della intera espressione rimane lo stesso; e il processo continua, appunto, finché si può applicare qualche regola di riscrittura. (In verità, in questo esempio, sono coinvolte le operazioni aritmetiche, quindi funzioni anziché semplici simboli: si parla allora di sistemi di riscrittura di *termini*, anziché di stringhe, tanto importanti per lo sviluppo di linguaggi algebrici... ma anche i sistemi di riscrittura di stringhe hanno la stessa potenza delle macchine di Turing!)

In generale, dato un sistema di riscrittura, bisogna chiedersi se tale processo termini sempre, da qualsiasi sequenza si parta, e, in caso affermativo, se sia anche *confluente*, vale a dire se, quando a un passo di riscrittura sono permesse più sostituzioni, tutte le scelte possibili conducano infine alla stessa sequenza, non più riscrivibile: la cosiddetta “forma normale” della sequenza di partenza. Se il sistema di riscrittura è terminante e confluente, allora ogni sequenza ha una e una sola forma normale. Inoltre, se il sistema di riscrittura è terminante, allora la confluenza è decidibile (in modo effettivo, cioè per via algoritmica).

**Soluzioni.** Partendo dalla fila con due biglie blu seguite da quattro rosse, applicando la prima regola, in sei passi otterremo la sequenza

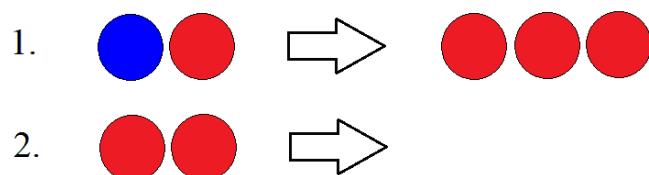


mentre applicando la seconda regola, ottenuta grazie a una “piccola” modifica della prima, il procedimento non termina, così come non termina partendo dalla sequenza (più breve)



Adottando la prima regola, invece, il procedimento termina sempre, nonostante tale regola (così come la seconda) produca una sequenza più lunga. Che cosa si potrà dire sulla confluenza?

Facciamo ora un esempio con *due* regole: ad ogni passo, si applichi o l’una o l’altra regola, finché sia possibile applicare almeno una delle due. (Si noti che la seconda stabilisce che una sequenza di due biglie rosse può essere eliminata.)



Anche qui chiediamoci: esiste qualche fila di biglie partendo dalla quale il processo non termina? E inoltre: *quali* sequenze possono essere ottenute a partire da quella qui sotto disegnata? (Questo, in effetti, è un esempio di sistema di riscrittura *non confluente!*)



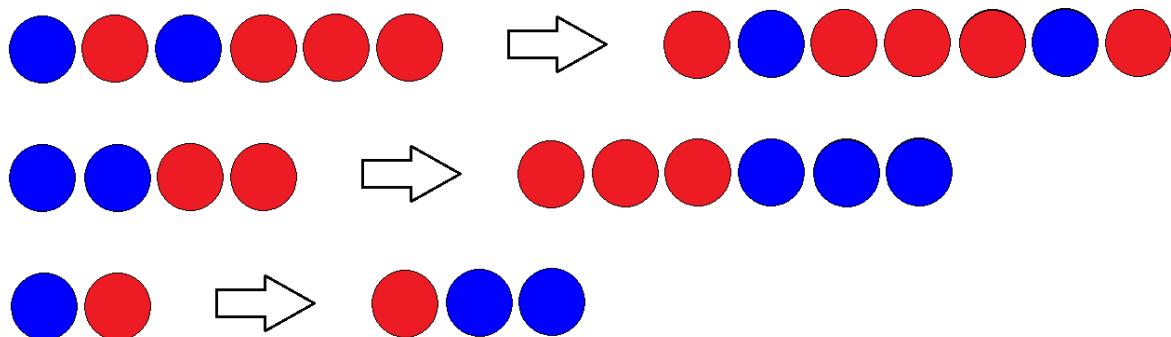
**Soluzioni.** Si può ottenere, alla fine, una di queste tre sequenze:



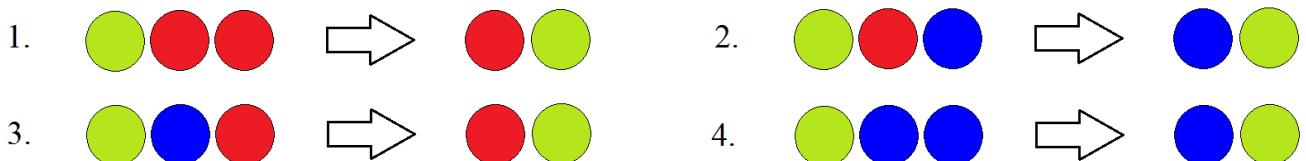
Una condizione certamente sufficiente per la terminazione è che ciascuna regola riduca la lunghezza della sequenza, ciò che non accade in questo caso, così come non accadeva nei precedenti. Qui, tuttavia, possiamo facilmente convincerci del fatto che il processo di riscrittura abbia sempre termine: le eventuali biglie blu in fondo alla sequenza iniziale non saranno coinvolte nel processo; le altre, se ci sono, non potranno diventare più numerose, mentre le biglie rosse potranno crescere in numero soltanto se vi è una biglia blu, che però scompare quando si applica la prima regola...

Il classico *problema della parola* (A. Thue, 1914: date le regole, la stringa iniziale e quella finale, stabilire se quest'ultima può essere ottenuta a partire dalla stringa iniziale) in generale è soltanto semidecidibile (E. L. Post, 1947; A. A. Markov, 1947; *cfr.* p. 91). Inoltre, è stato provato che, quando vi è *una sola regola*, la confluenza è decidibile (Celia Wrathall, 1990), ma sulla terminazione nulla si sa; comunque, in generale, entrambe le questioni sono indecidibili.

Vi lascio ancora tre esempi di sistema, con una sola regola ciascuno, su cui riflettere.

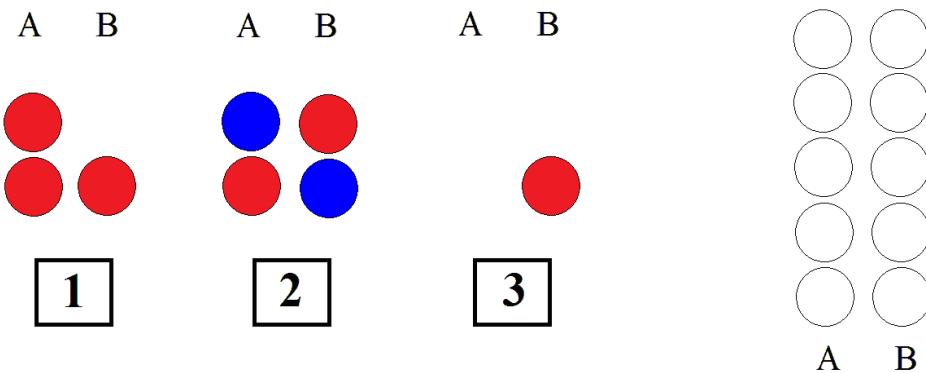


Il primo termina (e se togliessimo l'ultima biglia rossa, in fondo, sia a sinistra sia a destra?), il secondo termina (con complessità lineare) e il terzo pure (con complessità esponenziale). Le ultime due regole sono di un tipo particolare, studiato a fondo: da una parte della freccia stanno alcune biglie di un colore seguite da alcune biglie di colore diverso, dall'altra parte della freccia l'ordine dei due colori è invertito. Infine, ecco un sistema con quattro regole, terminante (in modo uniforme) e confluente, dall'aria piuttosto familiare (Ju. V. Matijasevič, 1967): che cosa ne pensate?



Passiamo a un altro gioco, che potremmo chiamare “cascate di biglie” e immaginare realizzato per mezzo di un software...

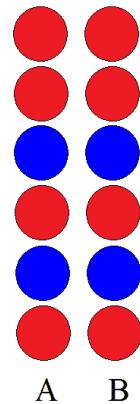
Leone sta provando un nuovo puzzle, che ha appena caricato sul proprio computer. Alcune coppie di colonne, formate da biglie di vari colori, compaiono sul display, a sinistra; ora sono uscite tre coppie, con biglie rosse e blu, e la colonna A della terza coppia è vuota (il bottone 3 lascerà cadere una sola biglia rossa nel cilindro B):



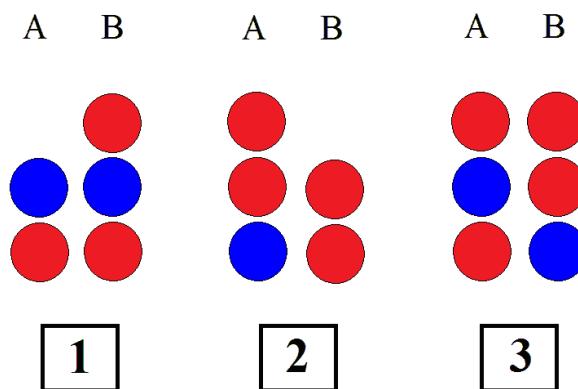
Ogni volta che Leone clicca su uno dei bottoni numerati, la corrispondente coppia di colonne cade, dall’alto, nei due cilindri A e B (all’inizio vuoti) a destra sul display. Il puzzle riesce quando nei due cilindri si ottengono due colonne *identiche*.

Non c’è limite al numero di *click* sui bottoni: assumiamo pure che le coppie di biglie di stesso colore, affiancate in fondo ai cilindri, scompaiano verso il basso, e quelle sopra scendano.

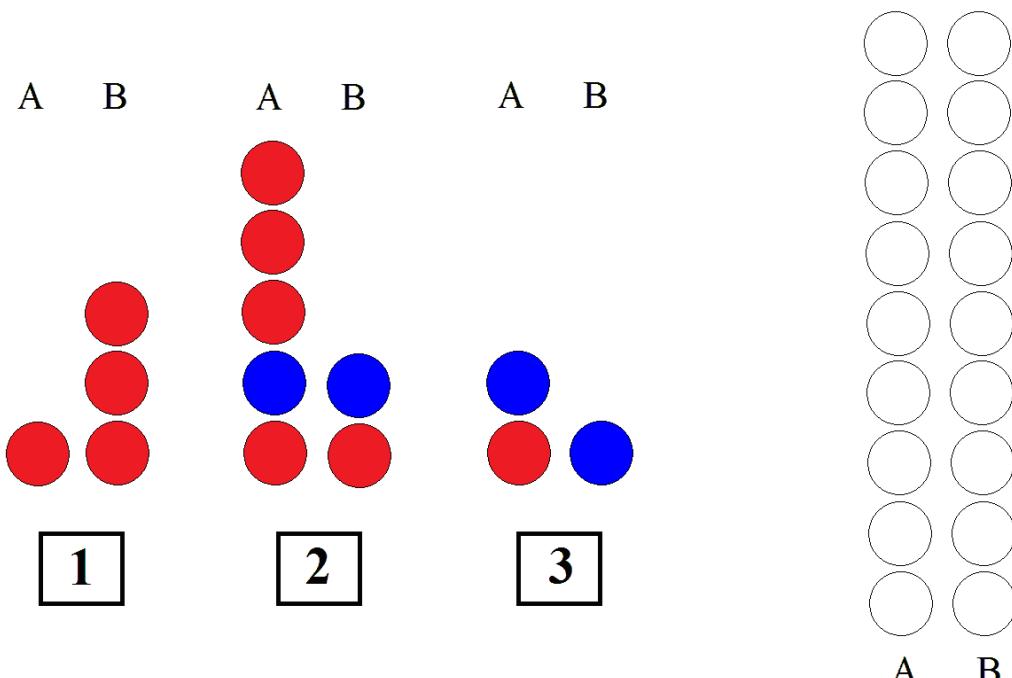
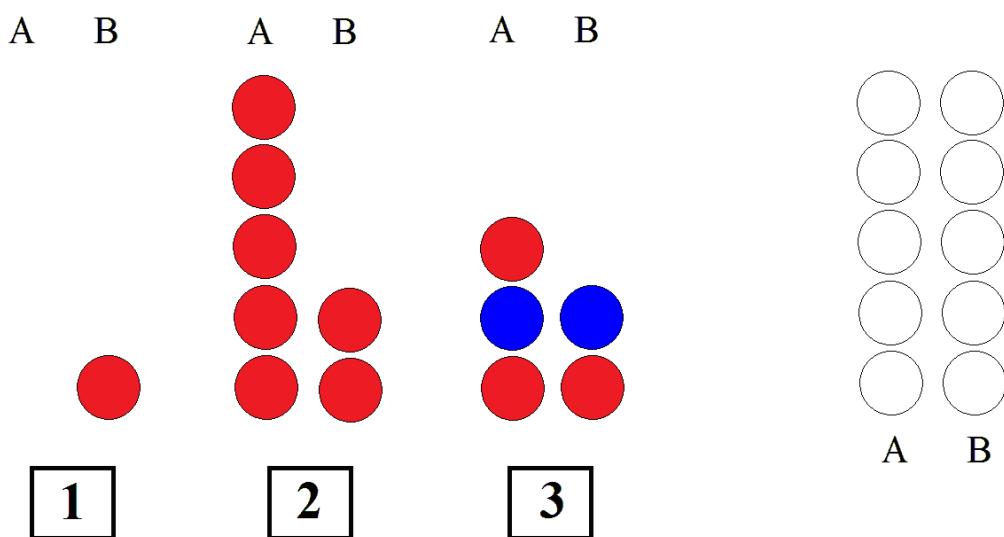
Pigiamo la sequenza [3, 2, 2, 1] si formerebbe la cascata qui a destra, e il puzzle sarebbe risolto. Notiamo che, in questo caso, ogni sequenza della forma  $[3, n \text{ volte } 2, 1]$ , con  $n \geq 0$ , è una soluzione, e così tutte quelle costituite dalle sue ripetizioni; sicché la soluzione più breve è [3, 1], che evita persino di pigiare il bottone 2!



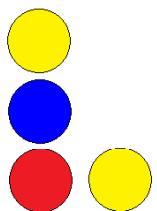
In generale, da ciascuna soluzione, se ne può costruire un’infinità di altre, ripetendo tale soluzione un numero arbitrario di volte. Certi casi, come quello ora illustrato, ammettono infinite soluzioni *non* ottenute per ripetizione. E, naturalmente, vi sono (infiniti) casi che non ammettono alcuna soluzione; eccone un esempio:



Propongo qui di seguito dei puzzle da risolvere, in ordine più o meno crescente di difficoltà: alcuni hanno quattro bottoni, il terzo anche biglie gialle, ma tutti sono risolubili. Buon divertimento... tuttavia non sperate di progettare un programma che possa preparare e poi risolvere puzzle casuali, pur limitati a pochi bottoni, poiché si tratta di istanze del *problema della corrispondenza* di Post, anch'esso *in generale* soltanto semidecidibile (E. L. Post, 1946). Precisiamo meglio: limitato a 2 coppie (nel nostro gioco, bottoni) tale problema è decidibile (A. Ehrenfeucht e altri, 1982), con 5 coppie non è decidibile (T. Neary, 2015), con 3 o 4 non si sa. A chi vuol saperne di più, e divertirsi a scoprire casi veramente difficili con 3 o 4 coppie, consiglio la pagina <http://webdocs.cs.ualberta.ca/~games/PCP/> (*PCP A nice problem*), da cui si può accedere alla tesi di Ling Zhao, del 2002, e anche risalire al dilettevole e stimolante sito ~games dell'Università di Alberta, Canada.

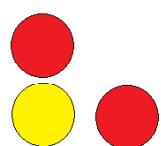


A B



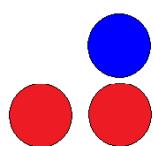
**1**

A B



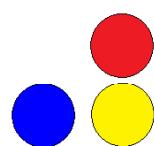
**2**

A B

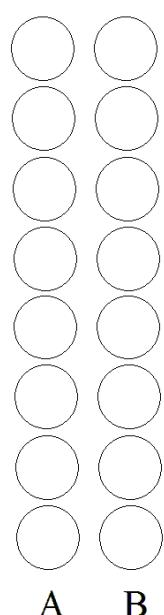


**3**

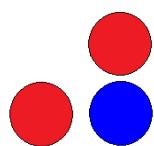
A B



**4**

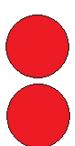


A B



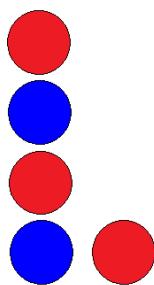
**1**

A B



**2**

A B

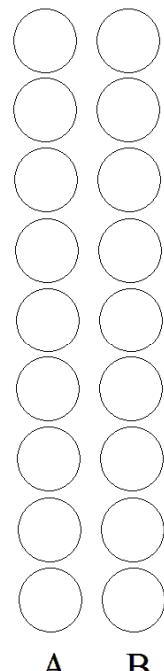


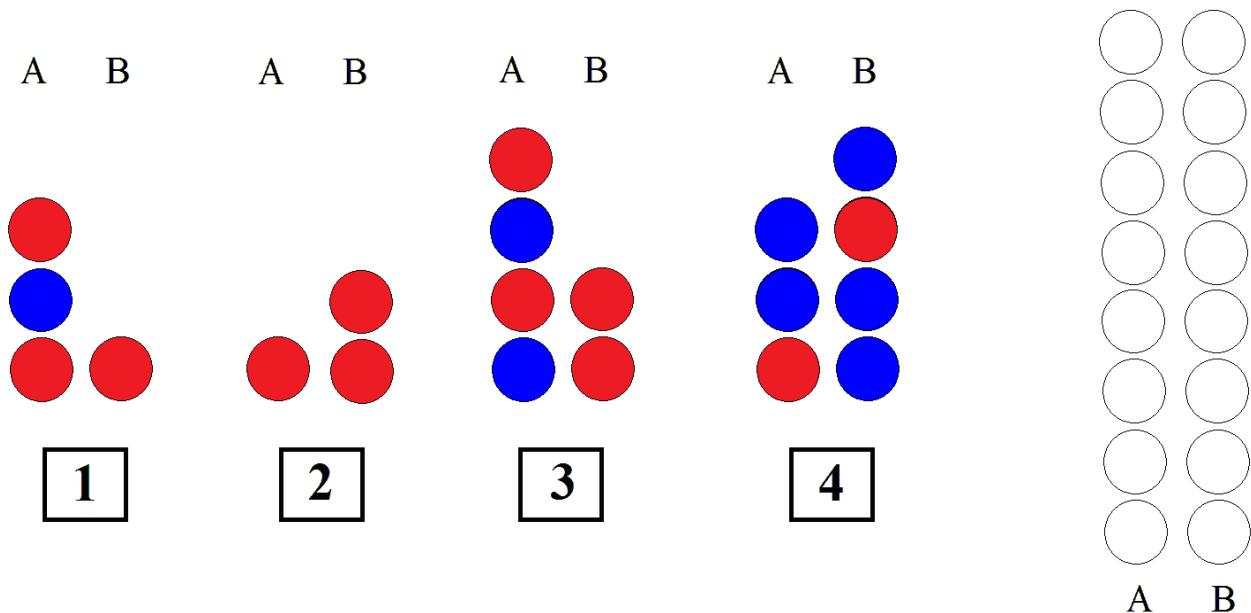
**3**

A B

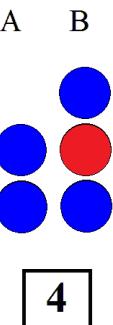


**4**





Se, in quest'ultimo caso, il bottone 4 si presentasse come nella figura qui a destra (notate che sono state tolte soltanto le due biglie alla base della precedente configurazione), allora il puzzle non sarebbe più risolvibile...



### Soluzioni. Elenco qui le soluzioni più brevi.

- Nel primo caso, il più semplice: 3, 1 (senza pigiare il bottone 2, altrimenti ogni soluzione è più lunga, ad esempio: una volta il bottone 2 e tre volte il bottone 1, in qualsiasi ordine);
- nel secondo caso: 2, 1, 1, 3;
- nel terzo caso: 3, 4, 2, 3, 1;
- nel quarto caso, dove i bottoni 2 e 4 lasciano cadere biglie in uno solo dei due cilindri: tre volte il bottone 2 e due volte il bottone 4, in qualsiasi ordine (senza utilizzare i bottoni 1 e 3; ci sono soluzioni più lunghe, che utilizzano anche questi, ad esempio: 2, 1, 1, 3, 2, 4);
- infine, nel quinto caso: 2, 4, 1 (senza pigiare il bottone 3: “modulo ripetizioni”, questa è l'unica soluzione).

Il matematico e logico statunitense, di origine polacca, Emil Leon Post (1897-1954) diede notevoli contributi all'informatica teorica. Già negli anni '20 del Novecento, egli ideò dei particolari sistemi di riscrittura, equivalenti alle macchine di Turing, ma pubblicò i risultati ottenuti soltanto nel 1943. Proprio questi *sistemi di Post* hanno ispirato il prossimo quesito, da me proposto per le gare Bebras nel 2016.

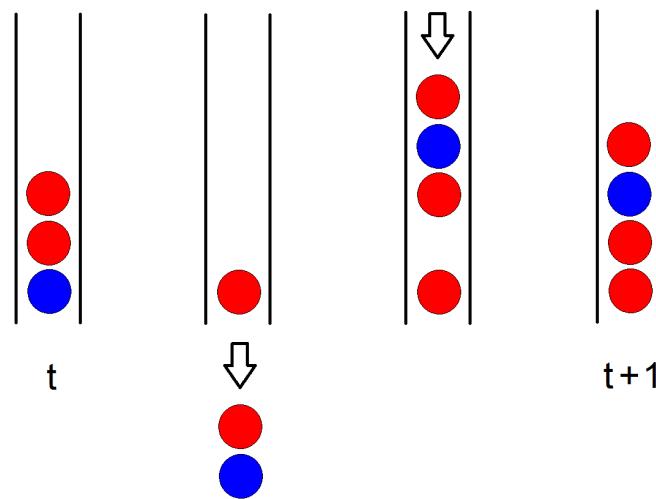
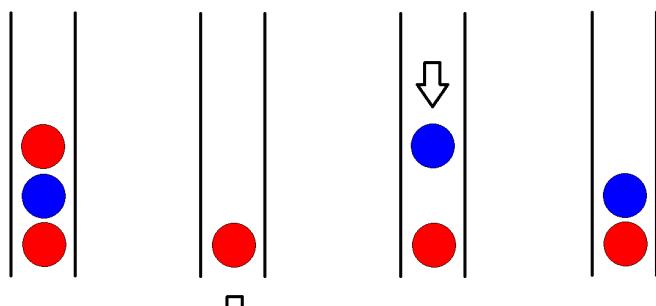
Leone ha ideato un altro gioco con biglie di due colori, e lo ha realizzato mediante un programma al computer che, attualmente, permette all'utente di predisporre, in un cilindro riprodotto sullo schermo, una colonna di almeno tre biglie, ciascuna delle quali può essere o rossa o blu. Dopodiché, ad ogni unità di tempo prefissata, le due biglie più in basso escono dal fondo del cilindro e, immediatamente, ne cade qualcun'altra dall'alto, secondo queste regole:

- se la biglia che esce per prima dal fondo è rossa, allora cade una nuova biglia blu in cima al cilindro;

- se invece la biglia che esce per prima dal fondo è blu, allora cadono tre nuove biglie in cima al cilindro: una rossa, una blu e una rossa, in quest'ordine.

Il processo termina soltanto se e quando rimangono meno di tre biglie nel cilindro.

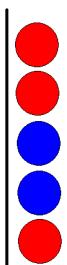
Le seguenti figure esemplificano le due regole; nel primo caso, il processo termina al tempo  $t+1$ , poiché restano due biglie.



E veniamo a un paio di domande interessanti.

Se il programma è fatto partire con la colonna raffigurata qui a destra, dopo quante unità di tempo si arresterà?

Sapreste trovare una colonna iniziale di sole tre biglie, partendo dalla quale il programma non si fermerà mai più?



Come abbiamo anticipato, questo gioco non è altro che un esempio di *sistema di produzioni di Post*, un modello di calcolo fondato ancora sulla riscrittura di stringhe, inteso come sequenze finite di simboli (qui due soli: biglia rossa e biglia blu, dunque l’alfabeto è binario).

In effetti, un sistema di Post, esattamente come una macchina di Turing, costituisce una possibile definizione di un ben preciso algoritmo, inteso come procedimento di calcolo per ottenere un “risultato” a partire da un “dato di input” rappresentato dalla stringa iniziale, ammesso che a tale ingresso corrisponda un’uscita.

Volendo essere più precisi, il sistema sul quale si basa il nostro gioco è un *2-tag system*, poiché due simboli (biglie) sono cancellati in fondo alla stringa (escono dal cilindro) ad ogni passo. L’insieme dei sistemi 2-tag è stato provato *Turing-completo*: come già è stato detto, ciò significa che esso ha la stessa capacità di calcolo di una macchina di Turing universale; da quando è stata ideata, nel 1936, si ritiene che quest’ultima – di cui i nostri computer costituiscono una realizzazione, seppur con memoria limitata – sia in grado di calcolare tutto quanto è, in linea di principio, calcolabile.

Più in generale, si può affermare che, per ogni  $m > 1$ , l’insieme dei sistemi  $m$ -tag è Turing-completo: per ogni data macchina di Turing T, esiste un sistema  $m$ -tag che emula T. In realtà, a dire il vero, si possono costruire diversi sistemi 2-tag (ma parecchio più complessi di quello qui presentato, che ha due sole regole!) ciascuno dei quali è in grado di emulare una macchina di Turing universale (H. Wang, 1963; J. Cocke e M. Minsky, 1964).

A chi desideri approfondire l’argomento, segnalo la tesi di Turlough Neary: *Small universal Turing machines* ([www.ini.uzh.ch/~tneary/tneary\\_Thesis.pdf](http://www.ini.uzh.ch/~tneary/tneary_Thesis.pdf)).

**Soluzioni.** Se parte con la colonna proposta, dopo 5 unità di tempo il programma si fermerà, perché resteranno due sole biglie blu. Infatti, indicando le biglie con i rispettivi colori, dall’alto verso il basso, si attuano le seguenti transizioni:

RRBBR → BRRB → RBRBR → BRBR → BBR → BB

Se parte invece con una colonna di tre biglie, delle quali quella in fondo sia di colore blu, il programma non terminerà. Innanzitutto, se la biglia in fondo fosse rossa, il programma terminerebbe trascorsa una sola unità di tempo, poiché nel cilindro si troverebbero soltanto due biglie. Se invece la biglia in fondo è blu, dopo, al più, cinque unità di tempo il cilindro conterrà la colonna RBRRBR (a partire dall’alto); infatti, essendo indifferente il colore della biglia centrale, i casi possibili sono due:

BXB → RBRB → RBRRB → RBRRBR;

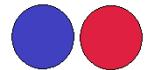
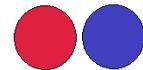
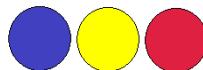
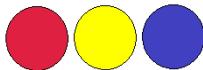
RXB → RBRR → BRB → RBRB → RBRRB → RBRRBR.

Così il programma è destinato a cadere in *loop*, perché dopo altre quattro unità di tempo la stessa configurazione RBRRBR si ripeterà:

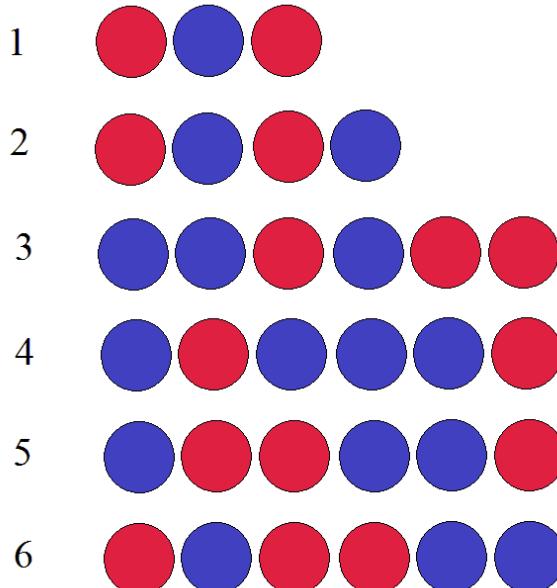
RBRRBR → BRBRR → BBRB → RBRBB → RBRRBR.

Concludo questo paragrafo ricordando un problema, che è invece decidibile, di cui abbiamo parlato nel terzo capitolo.

Si parte con una sola biglia gialla. Ad ogni passo, una biglia gialla può essere sostituita con una (a scelta) delle quattro sequenze mostrate in figura.



È possibile ottenere in qualche modo le sequenze sotto riportate?



Aggiungiamo la regola secondo la quale una biglia gialla può anche essere sostituita da *due* biglie gialle. Quali tra le sequenze sopra riportate si possono ora ottenere? E quali, fra queste, in più di un modo?

**Soluzioni.** Si possono ottenere soltanto le sequenze 2, 3 e 5, e in un unico modo.

Se una biglia gialla può anche essere sostituita da due biglie gialle, allora si può ottenere anche la sequenza 6. Se dicendo “in più di un modo” s’intende “passando attraverso sequenze diverse” (o anche “da alberi di derivazione diversi”, cfr. p. 82), allora ciascuna delle sequenze 2, 3, 5 e 6 può essere ottenuta in più modi: questa grammatica è *ambigua*!

In ogni caso, le sequenze ottenibili hanno tante biglie rosse quante blu, per cui le sequenze 1 e 4 sono escluse.

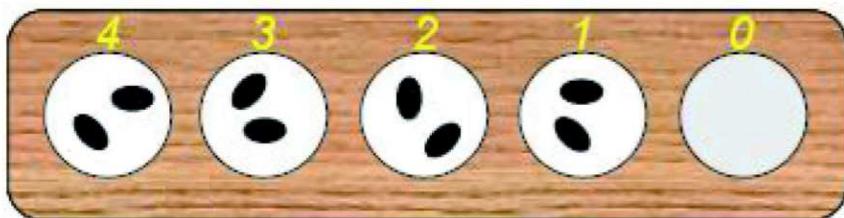
In questo quesito sono stati proposti due esempi di grammatiche libere da contesto, non equivalenti: abbiamo infatti constatato che non generano lo stesso linguaggio. Come già sappiamo dal terzo capitolo, per via algoritmica si può decidere se una data stringa appartiene al linguaggio generato da una data grammatica libera, mentre si può soltanto semidecidere se una data grammatica libera è ambigua, ma non si può neppure semidecidere se due date grammatiche libere sono equivalenti.

## 10. Tchouka e Tchoukaillon.

Nel decimo capitolo, abbiamo imparato una delle centinaia di varianti di una antichissima famiglia di giochi d'origine africana, detta degli Awélé o Mancala: sono giochi “di semina”, poiché i due contendenti depositano semi (o, in mancanza, sassolini) nelle buche che formano il tavoliere, poi prelevano tutti i semi da una buca e li ridistribuiscono uno dopo l'altro nelle buche successive, rispettando determinate regole. Ne esistono anche tante versioni da giocare “in solitario”, diffuse soprattutto in India, Malesia e Indonesia.

Il primo a riferire di *Tchouka* fu il matematico e storico francese, intendente militare Henri Auguste Delannoy, in una memoria pubblicata nel 1895, ove affermava che tale gioco gli era stato descritto in una lettera dal matematico Édouard Lucas, più volte citato in questo libro, e non soltanto per le sue più note “ricreazioni”, quali la torre di Hanoi e il congiungimento di punti (*pipopipette*, o *Dots-and-Boxes*, o *punti e linee*). Pare proprio che le regole riportate da Delannoy costituiscano una variante originale ideata da Lucas, poiché esaurienti ricerche etnografiche non hanno trovato un gioco identico in Indonesia. Lucas avrebbe sì potuto conoscere qualche altro gioco (magari a due contendenti, come l'indonesiano Dakon, o Dhakon) da cui trarre ispirazione, ma di certo adattò l'idea alle proprie intenzioni ricreative.

Nella moderna letteratura, questo solitario è noto come Tchuka Ruma (o Tchouka Rouma): nelle lingue della Malesia e dell'Indonesia *rumah* significa “casa”, mentre nel sud-est dell'India i semi del tamarindo sono spesso usati nei giochi “di semina”, e *chukra* in sanscrito indica l'aceto prodotto proprio con questi semi, da cui *chuka* in Malesia. La “casa” è rappresentata dalla buca più a destra di una fila di cinque; in ognuna delle altre quattro buche sono inizialmente depositi due semi.



L'obiettivo del gioco consiste nel portare tutti i semi nella *ruma*, la buca più a destra, per l'appunto.

Ad ogni passo, si devono prendere tutti i semi che si trovano in una delle quattro buche a sinistra (numerate da 1 a 4, nel disegno qui sopra) e, procedendo verso destra, seminarli uno per buca, *ruma* (buca 0) compresa; se si hanno ancora semi dopo averne posto uno nella *ruma*, si riprende a seminare dalla buca più a sinistra (buca 4). Se l'ultimo seme cade in una delle buche da 1 a 4 che già contiene almeno un seme, al passo successivo si devono prendere i semi di quella buca; se l'ultimo seme cade in una buca vuota, fra quelle da 1 a 4, il gioco termina e il solitario non riesce; se l'ultimo seme cade nella *ruma* e vi è almeno un'altra buca non vuota, si può scegliere da quale altra buca (non vuota) prendere i semi per il passo successivo.

Esiste una sola sequenza vincente, che consta di 10 passi, o semine: se non la trovate, leggete la soluzione alla fine del paragrafo.

Questo solitario può essere generalizzato in modo naturale, variando i suoi due parametri, che sono il numero  $k$  di semi depositati in ciascuna buca all'inizio del gioco e il numero  $n$  di buche oltre alla *ruma*.

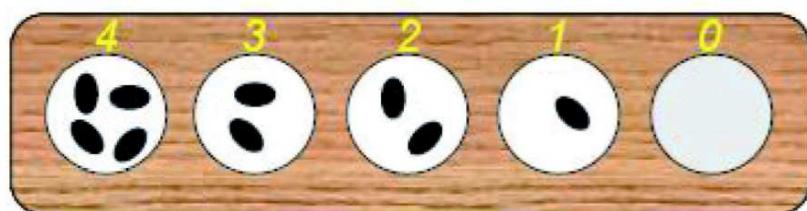
Con  $n = 4$  e  $k = 3$  (o 4 o 5) non vi sono sequenze vincenti; invece con  $k = 6$  sì.

Con  $n = 6$  e  $k = 3$  il solitario non può riuscire, mentre con  $k = 4$  (o 5) può riuscire.

Per saperne di più, consiglierei l'articolo di P. J. Campbell e D. P. Chavey del 1995, *Tchuka Ruma Solitaire*, The UMAP Journal, 16(4): 343-365, disponibile alla pagina <https://www.beloit.edu/computerscience/faculty/chavey/tchuka/>.

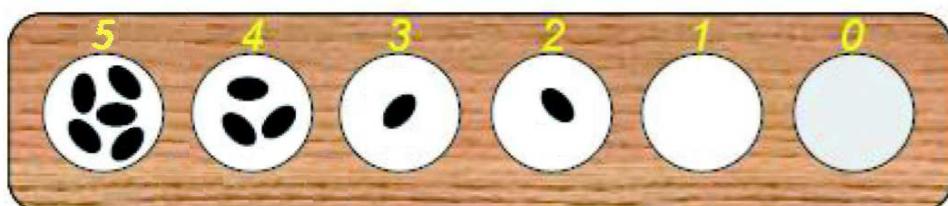
Nelle sue 23 pagine, l'articolo offre al lettore tanto un'analisi approfondita del gioco in generale, quanto la descrizione di alcune varianti e cenni sulla programmazione, nonché suggestive riflessioni di genere antropologico.

Una variante di Tchouka Ruma, detta *Tchoukaillon*, fu ideata dalla matematica francese Véronique Gautheron nel 1977, e da allora è stata studiata a fondo come gioco combinatorio. Estendiamo *ad libitum* il tavoliere verso sinistra, aggiungendovi le buche 5, 6, 7, ..., e prendiamo un certo numero di semi (nel seguito indicato con  $n$ ) che distribuiamo a nostro piacimento in varie buche, tranne che nella buca 0. Un passo del gioco consiste nel prelevare tutti i semi da una buca contrassegnata da un numero positivo e seminarli verso destra, uno in ciascuna delle buche contrassegnate da un numero inferiore; l'ultimo seme deve però cadere nella buca 0, altrimenti il solitario non riesce. Anche qui, naturalmente, l'obiettivo è portare tutti i semi nella buca 0. Dunque, affinché il solitario riesca, ad ogni passo deve esservi almeno una buca contrassegnata da un numero ( $> 0$ ) uguale alla quantità di semi in essa contenuti.



Ad esempio, usando lo stesso tavoliere di Tchouka con 5 buche, se vi disponiamo 9 semi come mostrato qui sopra, abbiamo la possibilità di vincere; dobbiamo però svuotare le buche precisamente nel seguente ordine: 1, 2, 1, 4, 1, 3, 1, 2, 1.

Se avessimo 10 semi, dovremmo usare un tavoliere con una buca in più e distribuirli inizialmente come nel seguente disegno, per far sì che il solitario possa riuscire.



La sola scelta che non porti a un'immediata sconfitta cade sulla buca 5: svuotandola, si giunge allo stesso stato del gioco precedente, con un seme già deposto nella buca 0. Notate che se volessimo partire con 11 semi, basterebbe aggiungere un seme nella buca 1 dell'ultima figura... e ovviamente iniziare a svuotare proprio la buca 1.

In effetti, è stato dimostrato che, per ogni numero naturale  $n$ , esiste una e una sola distribuzione di  $n$  semi (in un numero sufficiente di buche) che permette di vincere. Partendo da quella banale ( $n = 0$ ), ogni volta che si ripete questa procedura:

**se la buca numero 1 è vuota allora**

metti un seme nella buca numero 1

**altrimenti**

sia  $k (> 1)$  il numero della prima buca vuota, andando verso sinistra;

metti  $k$  semi nella buca numero  $k$ ;

togli un seme da ciascuna delle buche precedenti (da 1 a  $k - 1$ )

si ottiene la configurazione vincente con un seme in più!

Di seguito sono elencate le posizioni vincenti per  $n < 22$  (con 22 semi occorre ancora una buca in più): si vedano le sequenze A028931 e A028932 in OEIS.

<i>n</i>	<i>buche</i>						
	7	6	5	4	3	2	1
0							0
1							1
2				2	0		
3				2	1		
4			3	1	0		
5			3	1	1		
6		4	2	0	0		
7		4	2	0	1		
8		4	2	2	0		
9		4	2	2	1		
10		5	3	1	1	0	
11		5	3	1	1	1	
12	6	4	2	0	0	0	
13	6	4	2	0	0	1	
14	6	4	2	0	2	0	
15	6	4	2	0	2	1	
16	6	4	2	3	1	0	
17	6	4	2	3	1	1	
18	7	5	3	1	2	0	0
19	7	5	3	1	2	0	1
20	7	5	3	1	2	2	0
21	7	5	3	1	2	2	1

La sequenza di interi, indicanti passo dopo passo la prima buca vuota, è la seguente:

1, 2, 1, 3, 1, 4, 1, 2, 1, 5, 1, 6, 1, 2, 1, 3, 1, 7, 1, 2, 1, 8 ... (A028920 in OEIS)

(Si confronti con la sequenza risolutiva, *letta a ritroso*, dell'esempio a pagina 360.) Denotando con  $s(k)$  la posizione alla quale  $k$  occorre per la prima volta in questa sequenza, i valori di  $s(k)$  per  $k > 0$  formano un'altra sequenza:

$$1, 2, 4, 6, 10, 12, 18, 22 \dots \text{ (A002491 in OEIS)}$$

in cui diversi studiosi, tra i quali il grande matematico ungherese Paul Erdős, hanno scovato graziose proprietà, pur non direttamente collegate a Tchoukaillon;  $s(k)$  è il più piccolo numero di semi che richiede la presenza della buca  $k$  per poter vincere, ed è stato dimostrato che, al crescere di  $k$ ,  $s(k)$  tende asintoticamente a  $k^2 / \pi$  (D. M. Broline e D. E. Loeb, 1995): un risultato davvero sorprendente!

Dunque, tornando a considerazioni più immediate, la sola strategia vincente (*in avanti*) consiste nello svuotare, ad ogni passo, sempre osservando le buche da destra verso sinistra, la prima buca che abbia numero ( $> 0$ ) uguale alla quantità di semi in essa contenuti. Se, così facendo, il solitario non riesce, allora significa che dalla distribuzione di partenza è impossibile vincere.

Un'istanza di questo problema fu proposta dal gruppo Bebras italiano nel 2015.

L'autrice stessa di Tchoukaillon ne diede due varianti:

- *Monokalah*: si gioca su un tavoliere con un prefissato numero di buche; se, durante una semina, si hanno ancora semi in mano dopo averne deposto uno nella buca 0, si continua la semina dall'ultima buca a sinistra, proseguendo come al solito verso destra... ma se l'ultimo seme non cade nella buca 0, il solitario non riesce! Provate, ad esempio, con 5 buche, disponendovi inizialmente, da sinistra: 5, 3, 12, 1 semi.

- *Multitchouka*: come Monokalah... ma se l'ultimo seme cade in una buca ( $> 0$ ) non vuota, tutto il contenuto di quella buca si semina nuovamente, iniziando dalla buca adiacente a destra, e così via; se invece l'ultimo seme cade in una buca ( $> 0$ ) vuota, si perde. Provate a risolverlo con 4 buche, disponendovi da sinistra: 5, 1, 12 semi.

Infine, nel 2008, sulla stessa linea di Monokalah, Tom Bylander propose *Solitaire Mancala*: su un tavoliere con 7 buche (compresa la buca 0), l'ultimo seme non deve necessariamente cadere nella buca 0, ma quel che importa è terminare il gioco nel minor numero di passi.

**Soluzione.** La classica Tchouka Rouma illustrata a pagina 359 si risolve svuotando le buche nel seguente ordine: 2, 1, 3, 2, 1, 4, 1, 3, 2, 1.

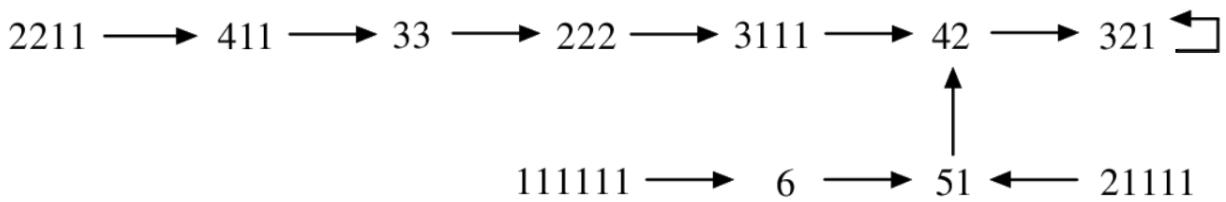
## 11. Il solitario bulgaro.

Se i giochi ai quali abbiamo accennato nel paragrafo precedente sono “di semina”, ve ne sono tanti altri “di semina inversa”, ossia “di mietitura”, “di raccolta”. Uno dei più noti e studiati, insieme con parecchie sue varianti, è il cosiddetto *solitario bulgaro*, della cui genesi è piacevole fare un breve racconto.

Verso il 1980, su un treno diretto a San Pietroburgo, un anonimo viaggiatore espose un fatto curioso a Konstantin Oskolkov, all’epoca professore presso l’Istituto di Matematica Steklov, a Mosca: si prendono 15 carte da gioco (o altrettante pedine) e si suddividono in diverse pile, alte a piacere; poi, ripetutamente, si preleva una carta da ciascuna pila e, con le carte raccolte, si forma una nuova pila. Comunque si ripartiscano all’inizio le 15 carte, dopo un certo numero di passi resteranno sul tavolo cinque pile, di 1, 2, 3, 4 e 5 carte, e questa situazione non muterà più dopo ulteriori passi. Tornato a Mosca, Oskolkov raccontò di questo incontro ai colleghi esperti di teoria dei numeri, che pensarono subito al problema generale con  $n$  carte.

Uno di loro, Anatolij Karacuba (famoso fra l’altro per l’*algoritmo di Karatsuba*, il primo metodo di moltiplicazione veloce), ne parlò durante una visita ai colleghi dell’Accademia delle Scienze in Bulgaria, nel maggio del 1980. In quello stesso anno il problema apparve su due riviste di matematica, in Bulgaria e in Russia, e nel 1981 furono pubblicate le prime soluzioni. Nel frattempo, anche un altro matematico in visita a Sofia portò la questione a conoscenza del suo amico Henrik Eriksson a Stoccolma, il quale pubblicò la propria soluzione e diede al gioco il nome di solitario bulgaro, pur riconoscendo che sia *bulgaro* sia *solitario* erano termini inappropriati! Dalla Svezia, questi la portò a sua volta negli Stati Uniti. Nel 1982 Donald Knuth iniziò un suo corso all’università di Stanford affrontando questo problema; l’anno successivo Martin Gardner ne scrisse su *Scientific American*, sicché da allora il gioco acquisì vasta popolarità, e continua ancor oggi a ispirare nuove ricerche nelle teorie dei giochi combinatori e delle probabilità, e a offrire spunti per la didattica.

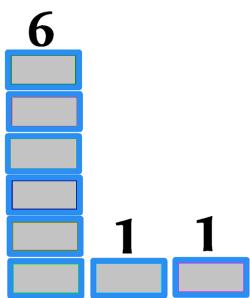
Vediamo quali sono stati i risultati più importanti ottenuti dall’analisi di questo “solitario”. Anzitutto, poiché il numero delle possibili *partizioni* dell’insieme delle  $n$  carte (per di più non distinguendo una carta dall’altra) è comunque finito, il solitario può essere riguardato come un *sistema dinamico discreto*, con un numero finito di stati (ad ogni partizione dell’insieme delle  $n$  carte corrisponde uno stato e viceversa), il cui comportamento è governato da una *funzione di transizione* da uno stato al successivo; tali stati possono essere rappresentati univocamente con sequenze finite di interi positivi, ordinate in modo non ascendente. Al sistema corrisponde quindi un grafo orientato, i cui nodi sono gli stati; e da ogni nodo esce esattamente un arco. Per qualsiasi  $n$ , e da qualunque nodo si parta, si tornerà prima o poi in un nodo già visitato, perché il numero di stati è finito; sicché il grafo presenterà almeno un *ciclo*. Ad esempio, alla pagina seguente, è disegnato il grafo completo per  $n = 6$ .



Noteate che il grafo ha un ciclo di lunghezza 1, cioè costituito da un solo nodo:  $(3, 2, 1)$  è dunque uno *stato stabile* del sistema, ossia un *punto fisso* della funzione di transizione. Gli stati  $(4, 2)$  e  $(5, 1)$  hanno due predecessori, mentre vi sono tre stati,  $(1, 1, 1, 1, 1, 1)$ ,  $(2, 1, 1, 1, 1)$  e  $(2, 2, 1, 1)$ , che non hanno alcun predecessore: questi sono i cosiddetti “giardini dell’Eden”, stati dai quali si può soltanto uscire per non tornarvi mai più!

In effetti, le prime “soluzioni” pubblicate dimostravano, in diversi modi, che quando  $n$  è un *numero triangolare*, cioè della forma  $1 + 2 + \dots + k = k(k+1)/2 = T_k$  per un qualche  $k$ , allora la funzione di transizione ha uno e un solo punto fisso, costituito dalla partizione  $(k, k-1, \dots, 2, 1)$ , a cui prima o poi si giunge partendo da qualsiasi altra. Noteate che 6 e 15 sono numeri triangolari (per  $k = 3$  e  $k = 5$ , rispettivamente). Successivamente, fu provato che il cammino più lungo che porta allo stato stabile parte dallo stato  $(k-1, k-1, k-2, \dots, 2, 1, 1)$ , ottenuto dallo stato stabile suddividendo la pila di  $k$  carte in due pile di  $k-1$  carte e una carta, e giunge allo stato stabile in un numero di passi dato dal prodotto  $k(k-1)$ .

Quando  $n$  non è un numero triangolare, si andrà a cadere in un ciclo certamente più lungo, cioè di almeno due stati. Nel grafo vi possono essere più cicli, che tendono a formarsi fra stati “quasi triangolari”: per la precisione, se  $T_{c-1}$  e  $T_c$  sono numeri triangolari e  $T_{c-1} < n < T_c$ , allora per ogni  $d$  che divide esattamente sia  $c$  sia  $n - T_{c-1}$  esiste un ciclo di lunghezza  $c/d$ , e non vi sono cicli di altre lunghezze: il ciclo più lungo ha dunque  $c$  stati (A. Grensjo, 2012).



Anche un’istanza del solitario bulgaro fu oggetto di un quesito nella gara Bebras del 2014, su proposta italiana: vi si chiedeva di riconoscere quale fosse, fra quelle presentate, una sequenza di passi ottenibile con  $n = 8$  pedine (anziché carte, ma di certo non fa differenza). Se provate a partire dalla situazione raffigurata qui a fianco, vi accorgerete che, dopo due passi, cadrete in un ciclo formato da due stati:  $(4, 2, 2)$  e  $(3, 3, 1, 1)$ .

Ma un fatto ancor più interessante – come sopra anticipato – è che, con 8 pedine, c’è un altro ciclo; se, anziché da  $(6, 1, 1)$ , partite ad esempio da  $(5, 2, 1)$ , subito dopo cadrete in un secondo ciclo, questa volta di quattro stati:  $(4, 3, 1)$ ,  $(3, 3, 2)$ ,  $(3, 2, 2, 1)$  e  $(4, 2, 1, 1)$ , in accordo con quanto poc’anzi affermato, poiché il primo numero triangolare maggiore di 8 è  $T_4 = 10$ . Un altro fatto degno di attenzione è che lo stato  $(6, 1, 1)$  ha un solo predecessore, precisamente lo stato  $(2, 2, 1, 1, 1, 1)$ , ma questo non ne ha alcuno: è un giardino dell’Eden.

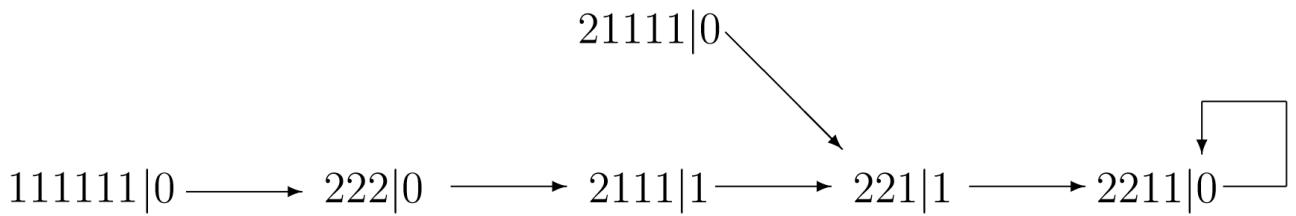
Se disegnate il grafo orientato delle transizioni tra gli stati (che sono ben 22), potrete verificare queste affermazioni, constatando pure che, essendovi due cicli, il grafo ha due *componenti connesse* (ossia è suddiviso in due parti “scollegate”); inoltre, vi sono sette giardini dell’Eden, due dei quali portano al ciclo di lunghezza 2, gli altri cinque a quello di lunghezza 4.

Il numero di partizioni, ossia di stati, cresce assai velocemente all’aumentare di  $n$ ; inoltre, per qualsiasi  $n$ , ognuno dei cicli può essere raggiunto a partire da un giardino dell’Eden. Per ogni  $n$ , è possibile sapere quanti sono i giardini dell’Eden; in effetti, essi sono caratterizzati dal fatto che la pila più alta ha un numero di carte inferiore al numero di pile decrementato di un’unità.

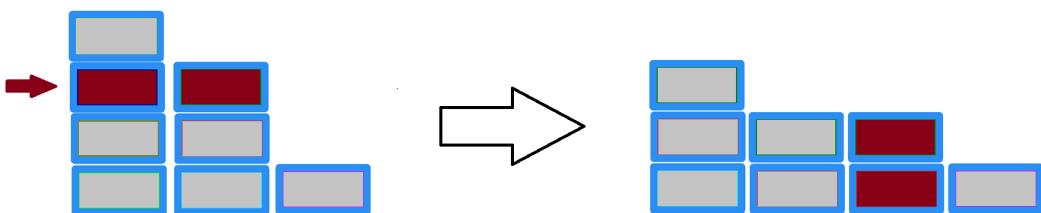
Molte varianti del solitario bulgaro sono state inventate e studiate a fondo dalla metà degli anni ’80; oltre a versioni “stocastiche” anziché deterministiche (introducendo delle probabilità di prelievo relative sia alle pile sia alle singole carte), oppure in tre dimensioni anziché in due, oppure ancora per due o più giocatori, menzioniamo le seguenti, rimandando il lettore interessato a un recente articolo di Vesselin Drensky (*The Bulgarian solitaire and the mathematics around it*, che si trova all’indirizzo <https://arxiv.org/pdf/1503.00885>) per bibliografia e indicazioni più precise:

- il *gioco duale*: mentre il gioco originale è basato sul principio di prendere una carta da ciascuna pila per formare un’altra pila (che può essere interpretato come prelevare una quota fissa da ciascuna società per formarne una nuova), nel gioco duale accade l’opposto, cioè si scomponе la pila più alta, aggiungendo una carta a ciascuna altra pila, e con le eventuali carte che avanzano si formano altrettante nuove pile di una sola carta ciascuna. È questo, in fondo, il principio di Robin Hood: prendere al ricco per dare al povero! Il gioco duale e quello originale sono isomorfi.

- Il *solitario austriaco*: ispirato dalla teoria del capitale scaturita dalla scuola austriaca (una scuola di pensiero economico eterodosso della seconda metà del XIX secolo), pone un limite all’altezza delle pile e aggiunge una “riserva”, o banca. Inizialmente si suddividono le  $n$  carte in pile di altezza  $\leq L$  ( $< n$ ) e la banca è vuota. Ad ogni passo si preleva una carta da ciascuna pila, si aggiungono alla banca le carte prelevate, e infine, finché è possibile, si prelevano nuove pile di altezza  $L$  dalla banca; quindi, alla fine di ogni passo, nella banca restano meno di  $L$  carte, eventualmente nessuna. In questo gioco non è affatto semplice stabilire quanti siano gli stati possibili, ossia i nodi del grafo. In cima alla pagina successiva è mostrato, a titolo d’esempio, il grafo per  $n = 6$  e  $L = 2$ ; le partizioni coinvolgono anche la banca (il cui deposito è indicato dopo la barra verticale), che però può essere vuota. Gli autori hanno congetturato che per ogni  $n$  e  $L$  vi sia sempre un unico ciclo (E. Akin e M. Davis, 1985).



- Una *versione per due giocatori* proposta da Brian Hopkins nel 2012: ad ogni mossa, il giocatore di turno sceglie a suo piacimento un'altezza (compresa tra 1 e l'altezza della pila più alta) alla quale prelevare una carta (o, meglio, una pedina) da ciascuna pila, per formarne una nuova. Ad esempio, con  $n = 8$ , nello stato  $(4, 3, 1)$ , se il giocatore di turno sceglie l'altezza 3, lo stato successivo sarà  $(3, 2, 2, 1)$ , come appresso illustrato.



Stesso risultato se sceglie l'altezza 2; se invece sceglie l'altezza 4, lo stato successivo sarà  $(3, 3, 1, 1)$ , mentre con l'altezza 1 (proprio come nel classico solitario bulgaro) si passa nello stato  $(3, 3, 2)$ . Si parte con una sola pila di  $n$  pedine; perde chi per primo ritorna in uno stato già visitato, ossia crea una partizione già vista nel corso della partita. Si tratta dunque di un *gioco combinatorio imparziale*, a informazione perfetta (cfr. l'undicesimo capitolo del libro): quale dei due giocatori ha la strategia vincente, e in che cosa consiste questa strategia? Le risposte a queste domande dipendono dal numero di pedine? In generale, sono ancora questioni aperte!

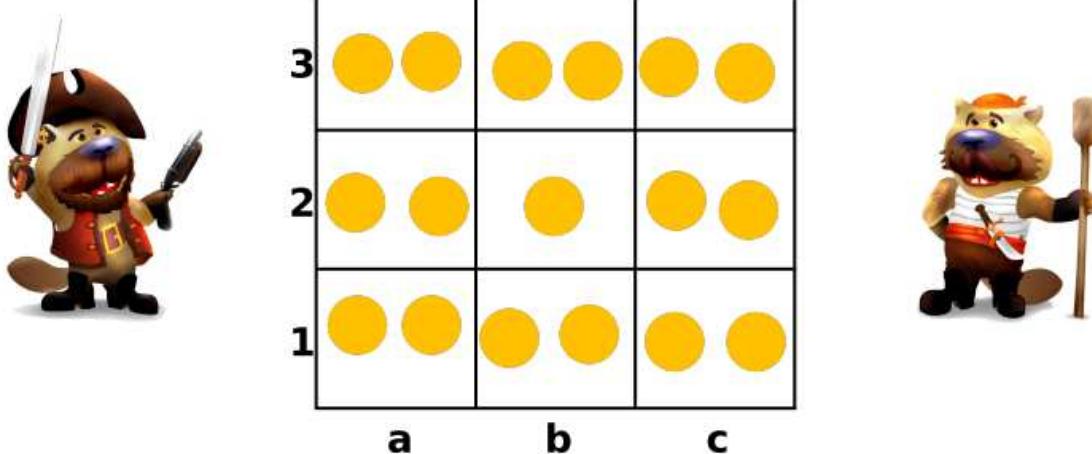
## 12. Ancora qualcosa sui giochi di strategia.

Nel 2014 il gruppo Bebras dell'Ucraina propose un facile ma interessante gioco di strategia, da noi ripreso per la finale Kangourou 2015. Eccolo, con i miei commenti.

I pirati Dente e Coda disputano una partita al seguente gioco, in cui vince chi riesce a prendere più monete d'oro.

Dapprima Dente occupa una casella a sua scelta e prende le monete che vi si trovano; poi Coda fa la stessa cosa con un'altra casella.

Dopodiché, per tre volte, a turno, sempre prima Dente e poi Coda, si muovono su una casella adiacente, in orizzontale o in verticale, prendendo le monete che vi si trovano. Ciascuno di loro non può andare sulla casella occupata dall'avversario.



Chi dei due vince la partita? E come procede?

**Soluzione.** Vince Coda: per ottenere la vittoria, deve forzare Dente a spostarsi in una casella vuota, e ciò è certamente possibile se Coda “ha l’opposizione”, per usare il gergo scacchistico...

Vale la pena, tuttavia, di analizzare in dettaglio tutti e tre i casi possibili.

a) Se Dente occupa una *casella d’angolo*, ad esempio a3, Coda gli si pone a fianco, in a2 o in b3; supponiamo in b3. Allora Dente deve spostarsi in a2, e Coda va in b2; Dente va in a1 e Coda in b1. Ora Dente è costretto a tornare in una casella vuota, mentre Coda va in c1 e vince 7 a 6.

*Nota.* In alternativa, se Dente occupa a3, Coda può anche occupare b1 (o c2) avendo ugualmente una strategia vincente. Infatti, supponendo che Dente occupi a3 e Coda b1, le varianti principali sono tre:

1. Dente b3, Coda c1, Dente c3, Coda c2, Dente b3, Coda b2, e Coda vince 7 a 6;
2. Dente b3, Coda c1, Dente b2, Coda c2, Dente a2, Coda c3, e Coda vince 8 a 7;
3. Dente a2, Coda b2, Dente a1, Coda c2, Dente ..., Coda c1, e Coda vince 7 a 6.

b) Se Dente occupa la *casella centrale*, Coda deve necessariamente occupare una casella al centro di un lato, supponiamo b3. Se Dente scende a occupare b1, Coda si può spostare indifferentemente in a3 o in c3... e alla fine potrà vincere 8 a 7 oppure 6 a 5 (dipende se Dente si muove dalla parte opposta o dalla stessa parte di Coda), oppure si può spostare nella casella centrale b2, assicurandosi la possibilità di vincere 6 a 5; altrimenti, se Dente va ad occupare a2 o c2, Coda si deve spostare dalla parte opposta, ossia in c3 o a3, rispettivamente, e questa volta potrà vincere 8 a 7.

Ecco le varianti principali, dopo che Dente occupa b2 e Coda b3:

1. Dente b1, Coda a3, Dente a1, Coda a2, Dente ..., Coda ..., e Coda vince 6 a 5;
2. Dente b1, Coda a3, Dente c1, Coda a2, Dente c2, Coda a1, e Coda vince 8 a 7;
3. Dente b1, Coda b2, Dente a1, Coda a2, Dente b1, Coda a3, e Coda vince 6 a 5;
4. Dente a2, Coda c3, Dente a1, Coda c2, Dente b1, Coda c1, e Coda vince 8 a 7.

c) Se Dente occupa una *casella al centro di un lato*, ad esempio b3, Coda si deve piazzare al centro. Dente è quindi costretto ad andare in un angolo, supponiamo in a3, e allora Coda va in a2. Di nuovo, Dente deve tornare in una casella vuota, b3, mentre Coda può scendere in a1; l'ultima mossa frutterà a ciascuno altre due monete, sicché la partita è ancora vinta da Coda 7 a 6.

I giochi di strategia sono stati argomento degli ultimi tre capitoli del presente libro. Ivi è stato appena nominato un gioco, che ha avuto discreto successo presso studenti e ricercatori, parecchi dei quali hanno divulgato in rete le proprie tesi sull'argomento: Forza 4. Inventato da Howard Wexler e Ned Strongin, e in commercio negli Stati Uniti dal 1974 col nome Connect Four, è un gioco di allineamento che ricorda il filetto; il tavoliere, che nella versione classica ha 6 righe e 7 colonne, è però disposto verticalmente, in modo che le pedine cadano nelle colonne fermandosi nel posto libero più in basso. A turno, dunque, ciascuno dei due giocatori fa cadere una pedina del proprio colore in una colonna (non piena) a sua scelta; vince chi riesce a disporre in fila (orizzontale, verticale o diagonale) quattro pedine del proprio colore.

Gli stati possibili sono oltre 4500 miliardi; ciononostante, nel 1988 Forza 4 fu risolto, sebbene in senso debole, da James D. Allen (che nel 2010 ha pubblicato, presso la Sterling Publishing Company, *The Complete Book of Connect 4: History, Strategy, Puzzles*) e, indipendentemente, da Louis Victor Allis (del *team* di ricercatori olandesi che studiò l'Aware): il primo giocatore può vincere, iniziando dalla colonna centrale. Secondo Allis, le mosse dovrebbero essere guidate da un obiettivo identificato in anticipo, che qui è infine l'allineamento di quattro pedine, ma è tutt'altro che banale valutare l'efficacia di un tentativo; nella sua tesi, disponibile in rete, egli presenta un programma, basato su nove regole strategiche, delle quali dimostra la correttezza.

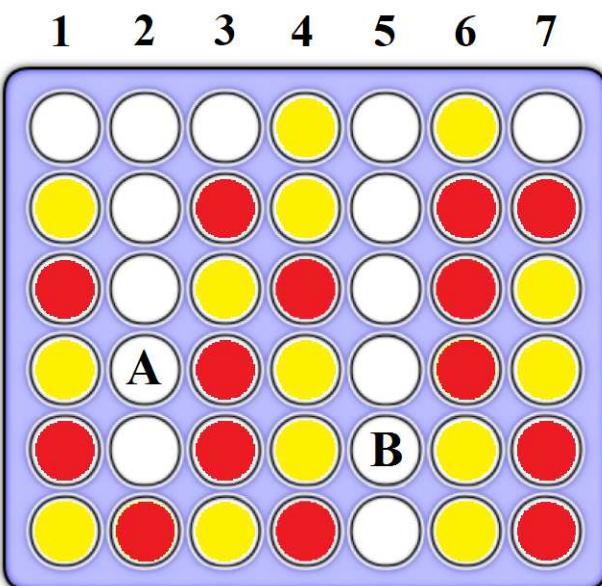
<http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf>

La risoluzione in senso forte ebbe inizio col lavoro di un altro olandese, Johannes T. Tromp, che nel 1995 approntò un database contenente tutte le posizioni legali a otto

semimosse in cui nessuno ha ancora vinto e la successiva mossa non è forzata. Per quanto si sa, se chi inizia fa cadere la pedina in colonna 3 o 5 la partita è teoricamente patta, ma se sceglie una delle colonne laterali (1, 2, 6 e 7) allora regala la strategia vincente al secondo giocatore – purché, s'intende, il gioco proceda poi in maniera perfetta da entrambe le parti.

Un semplice programma di “intelligenza artificiale”, sviluppato ad esempio in un linguaggio a oggetti, come C++ o Java, oppure in Prolog, analizza l’albero di gioco fino a un certo livello, sottponendo gli stati oltre i quali non scende a una funzione di valutazione euristica, che ne possa stimare al meglio l’utilità attesa, e “potando” l’albero, durante la visita in profondità, tutte le volte che è possibile.

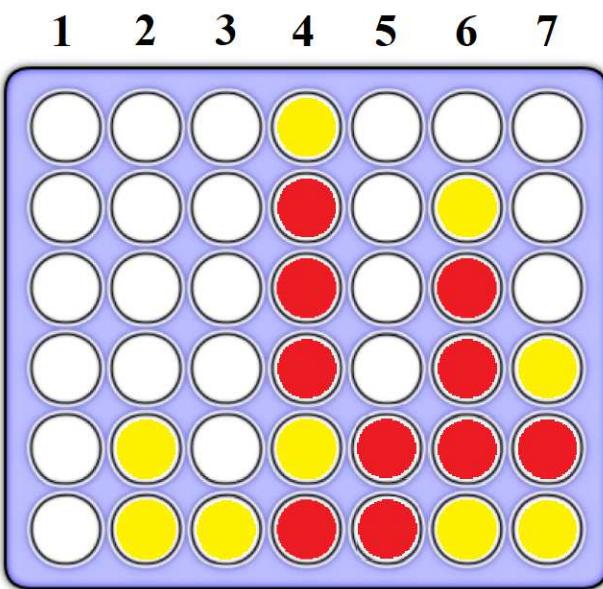
Come sappiamo, l’aspetto più delicato è proprio la messa a punto di una efficace funzione di valutazione: nel caso di Forza 4, essa dovrà considerare, e pesare in modo appropriato, l’allineamento di più pedine di stesso colore lungo le diverse direzioni, le mosse forzate e le minacce (eventualmente doppie). Senza scendere in dettaglio, una *minaccia* è un posto libero che, qualora venisse occupato dall’avversario, questi completerebbe un quartetto di sue pedine e quindi vincerebbe; se per qualche motivo tale evento fosse impossibile, la minaccia sarebbe inutile. Chiaramente, una minaccia in verticale sta sopra tre pedine, e dunque o è subito resa vana dall’avversario (che si trova ad avere una mossa forzata) o chi l’ha attuata vince, completando il quartetto. Una minaccia in orizzontale o in diagonale, che si trovi subito sopra una minaccia avversaria, è inutile: o riesce quella che sta sotto o sono entrambe neutralizzabili da parte avversaria. Per il giocatore che inizia la partita, le minacce migliori sono quelle che hanno sotto di sé un numero pari di posti; per il secondo giocatore, invece, un numero dispari di posti. Vediamo un esempio; chi ha iniziato ha le pedine rosse.



Entrambi i giocatori hanno una buona minaccia, in diagonale, su differenti colonne: A per il Rosso (con due posti sotto), B per il Giallo (con un posto sotto); se fossero sulla stessa colonna, vincerebbe il giocatore che attua la minaccia inferiore. Tuttavia qui il Rosso, dovendo muovere, può vincere entro 6 mosse: come deve giocare?

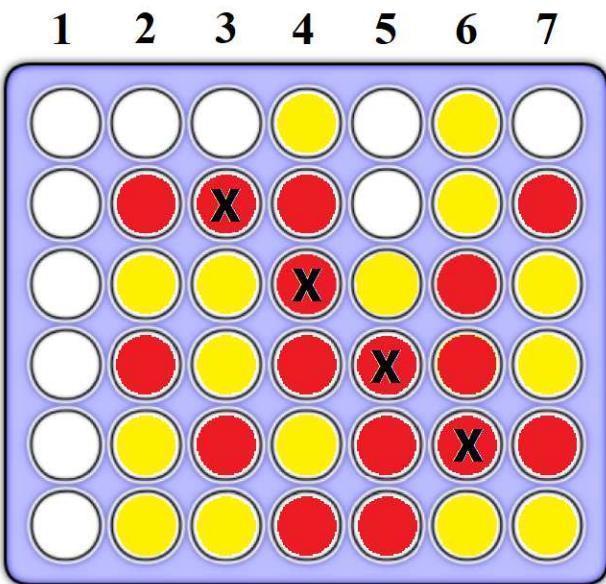
Ovviamente, non deve scegliere la colonna 5, altrimenti perde subito; ma non deve nemmeno calare la propria pedina nella colonna 2, perché anche il Giallo opterebbe per la stessa colonna, annullando in tal modo la minaccia del Rosso, e vincerebbe in 5 mosse. Il Rosso deve fare una mossa d'attesa, completando una delle colonne 1, 3 o 7, indifferentemente. Ora tocca al Giallo, il quale, del tutto analogamente, se cala una pedina nella colonna 2 permette al Rosso di vincere subito, mentre se sceglie la colonna 5 dà al Rosso la possibilità di occupare il posto B. Così anche il Giallo completerà una delle altre due colonne rimaste, e il Rosso l'ultima. A questo punto, per ritardare ancora la sconfitta, il Giallo opterà per la colonna 5, che anche il Rosso sceglierà per vanificare la minaccia del Giallo...

Ora propongo un problema. Nello schema sottostante deve muovere il Giallo, che però è destinato a perdere entro 6 mosse: come potrà evolvere la partita, se il Giallo agirà in modo da procrastinare il più possibile la propria sconfitta?



**Soluzione.** Per brevità, indicheremo le mosse con l'iniziale del giocatore e il numero della colonna in cui farà cadere una delle proprie pedine. Il Rosso dispone di una minaccia immediata, per cui il Giallo ha una mossa forzata: **G7**. Il Rosso ha due mosse ugualmente ottime: R2 e R3. Supponiamo che scelga **R2** (a R3 seguirebbe G2 o G3 o G7); il Giallo può rispondere con G6 o G7: assumiamo **G6**. Dopodiché: **R3** (con una minaccia; in alternativa R7), **G3** (in alternativa G7), **R7** (in alternativa R3), **G2** (con una minaccia; in alternativa G1 o G7), **R5** (con minaccia immediata in colonna 5 e minaccia "buona" in colonna 3), **G5** (mossa forzata), **R2** (con minaccia immediata, ancora in colonna 3, che raddoppia la precedente). A questo punto, il Giallo è costretto a contrastare la minaccia inferiore in colonna 3, ma non può far nulla per neutralizzare la superiore; così la partita si conclude, dopo **G3** e **R3**, nello stato raffigurato alla pagina successiva.

Potete divertirvi a giocare in due, o contro un programma "perfetto", accettando magari i suoi suggerimenti, tramite il sito <https://connect4.gamesolver.org/>.



Un gioco che riscuote sempre successo, e mette talvolta a dura prova le capacità deduttive di chi compete in gare di abilità mentale, è il Mastermind. In commercio dal 1970 nelle eleganti confezioni con pioli di tanti colori, risale in realtà ad almeno un secolo prima, conosciuto con nomi diversi e praticato semplicemente con carta e matita. Non solo: verso la fine degli anni '60 erano già stati sviluppati programmi al calcolatore in grado di sostenere il ruolo del solutore, sebbene all'epoca non fosse ancora nota una strategia ottimale.

Le regole sono note: il codificatore stabilisce un codice segreto, ossia una sequenza di 4 o 5 colori (o cifre), scelti tra 6 o 8 o 10 diversi, con o senza possibilità di ripetizioni. Il solutore (decodificatore) deve scoprire tale codice facendo dei tentativi, per ciascuno dei quali il codificatore dirà quanti colori (o cifre) sono presenti al posto giusto e quanti altri al posto sbagliato. Nella classica versione con sei cifre (da 1 a 6) e codice di quattro cifre con possibili ripetizioni, i codici sono  $6^4 = 1296$ ; ad esempio, se il codice segreto è 1535, al tentativo 3523 il codificatore risponde: una cifra al posto giusto (“un nero”) e una al posto sbagliato (“un bianco”). Infatti, 5 è in seconda posizione in entrambe le sequenze, mentre 3 è al terzo posto nel codice segreto e quindi *una* delle cifre 3 del tentativo è corretta, tuttavia nessuna delle due si trova al posto giusto.

Un ovvio programma che sostenga la parte del solutore potrebbe iniziare con un tentativo scelto a caso tra i 1296; acquisita la risposta del codificatore, elimina tutti i codici incompatibili con la risposta ricevuta e, tra quelli rimasti, ne sceglie uno a caso; e così via, finché non ottiene la risposta “quattro neri”.

Si possono tuttavia adottare strategie appositamente studiate nel corso degli anni, per tener conto di vari parametri statistici o semplicemente per minimizzare il numero massimo di tentativi: quest’ultimo obiettivo fu oggetto di un’analisi compiuta dal già più volte citato D. E. Knuth (*The computer as master mind*, Journal of Recreational Mathematics, Vol. 9, No. 1, 1976/1977, pp. 1-6). Knuth fu il primo a dimostrare che talora può convenire un tentativo sicuramente diverso dalla soluzione, che quindi non otterrà la risposta “quattro neri”, ma lascerà il minor numero di ulteriori possibilità.

Knuth consiglia di iniziare con due cifre uguali seguite da altre due cifre uguali; senza perdita di generalità, supponiamo dunque che il primo tentativo sia 1122. Per il decodificatore, la risposta migliore – dopo “quattro neri” – è “quattro bianchi”: in tal caso il codice segreto non può che essere 2211, e quindi la soluzione sarà trovata al secondo passo. Per ognuna delle altre possibili risposte a 1122 ( $n$  = neri,  $b$  = bianchi), la seguente tabella riporta: il numero dei codici compatibili con la risposta (cc1), il secondo tentativo che lascerà il minor numero di ulteriori possibilità (t2) e, appunto, quanti codici al massimo potranno ancora esservi dopo il secondo tentativo, nel peggiore dei casi (cc2).

n b	cc1	t2	cc2
0 3	16	1213 *	4
0 2	96	2344 *	18
0 1	256	2344	44
0 0	256	3345	46
1 2	36	1213	7
1 1	208	1134 *	38
1 0	256	1344	44
2 2	4	1213 *	1
2 1	32	1223	6
2 0	114	1234 *	20
3 0	20	1223 *	5

Naturalmente, nella colonna t2, le cifre 3, 4 e 5 stanno per tre cifre indifferentemente scelte nell’insieme  $\{3, 4, 5, 6\}$ .

Si noti che, oltre a 3 neri e 1 bianco che è sempre impossibile, il tentativo 1122 non ammette neppure la risposta 1 nero e 3 bianchi; ma, soprattutto, si noti che i tentativi marcati con \* sicuramente non coincidono con la soluzione, e tuttavia ci permettono di sfoltire il più possibile i rimanenti codici coerenti con le risposte ricevute.

Analizziamo, ad esempio, il caso più semplice: supponiamo di ricevere la risposta 2 neri e 2 bianchi. Rimangono quattro codici possibili: 1221, 1212, 2121, 2112; allora:

se il codice segreto è: 1221      1212      2121      2112

e il secondo tentativo è:		riceveremo la risposta:		
1221		4n	2n, 2b	2n, 2b
1212		2n, 2b	4n	4b
2121		2n, 2b	4b	2n, 2b
2112		4b	2n, 2b	2n, 2b

e dunque, per ognuno dei quattro possibili tentativi “corretti”, la risposta (2n, 2b) non ci permette di concludere al prossimo passo, ciò che invece farà il tentativo “sbagliato” 1213 (con cifra finale né 1 né 2), ricevendo quattro risposte diverse:

1213      |      2n, 1b      3n      3b      1n, 2b.

Così, in base alla risposta ricevuta, individueremo subito il codice segreto! Dalla tabella alla pagina precedente, si evince che le risposte peggiori che possiamo aspettarci dopo il primo tentativo sono: un solo nero, o un solo bianco, o nessun nero e nessun bianco; tutte e tre lasciano 256 codici coerenti, e il secondo miglior tentativo sarà questa volta uno di essi.

L'analisi completa fatta da Knuth porta alla conclusione che si può sempre giungere alla soluzione in, al più, 5 passi, stabilendo man mano i tentativi in modo opportuno. Tuttavia, com'egli stesso osserva, la sua strategia non minimizza il numero *medio* di passi...

Anche per il classico Mastermind vi propongo un problema. Supponete di aver già fatto tre tentativi, con i seguenti esiti:

1234	2b
5562	1n
3322	1b

(Si noti che il secondo tentativo non è coerente con l'esito del primo, quindi non poteva di certo essere la soluzione.)

- a) Quanti sono i codici segreti compatibili con le risposte ricevute?
- b) Sapreste individuare, tra questi, un quarto tentativo che, se non riceve risposta 4n, garantisca di determinare il codice segreto al passo successivo?
- c) Supponiamo che il terzo tentativo, anziché 3322, sia 2422 con esito 1n. Allora i codici segreti compatibili con le tre risposte sarebbero nove, ma tra questi non vi è ora alcun tentativo che, se non è la soluzione, può garantire di individuarla al passo successivo. Esiste però un *unico* tentativo (sicuramente diverso dalla soluzione) che permette di concludere (altrettanto sicuramente) al passo successivo: sapreste dire qual è? (Questa domanda è veramente diabolica!)

**Soluzioni.** a) Sono sette: 4463, 4543, 4663, 5113, 5443, 6163, 6463.

b) Tra questi, soltanto 4663, proposto come quarto tentativo, riceverà sette risposte diverse, che permetteranno quindi di concludere, come si evince dalla tabella sotto riportata, analoga alla precedente:

4463	4543	4663	5113	5443	6163	6463
4463   4n	2n, 1b	3n	1n	2n, 1b	2n	3n
4543   2n, 1b	4n	2n	1n, 1b	2n, 2b	1n	1n, 1b
4663   3n	2n	4n	1n	1n, 1b	2n, 1b	2n, 2b
5113   1n	1n, 1b	1n	4n	2n	2n	1n
5443   2n, 1b	2n, 2b	1n, 1b	2n	4n	1n	2n
6163   2n	1n	2n, 1b	2n	1n	4n	3n
6463   3n	1n, 1b	2n, 2b	1n	2n	3n	4n

(Per la precisione, ci sono altri 12 codici che, pur essendo “sbagliati”, garantiscono di non fallire al quinto passo.)

Pur non avendo adottato la strategia di Knuth, con i primi tre tentativi siamo stati fortunati: ci hanno permesso infatti di trovare la soluzione, al più, al quinto passo.

Notate che le matrici che abbiamo costruito sono simmetriche: la risposta ricevuta dipende dai due codici, non importa quale sia il tentativo e quale il codice segreto.

c) Se il terzo tentativo fosse 2422 con esito 1n, allora i codici segreti compatibili con i tre esiti finora avuti sarebbero questi nove:

3463, 3466, 4461, 4463, 5411, 5441, 5443, 6461, 6463

ma al quarto passo dovremmo proporre al codificatore 6343, non compreso tra essi, in modo da ottenere nove diverse risposte; nei rispettivi casi:

1n 3b, 3b, 2b, 1n 2b, 1b, 1n, 2n, 1n 1b, 2n 1b.

Notate che soltanto tre codici fra i nove scritti sopra sono anche fra i sette elencati al punto a): il codice segreto sarà allora uno di questi tre!

Un parente stretto del Mastermind è giocato con i bit, informando il decodificatore del solo numero di “neri”, ossia di cifre binarie azzeccate al posto giusto. Supponete di giocare con sequenze di otto bit, di aver fatto cinque tentativi e di aver ricevuto, per ognuno di essi, la risposta scritta a fianco.

10110111	5
00010110	2
10100001	6
11100101	4
10101011	6

Al prossimo tentativo dovreste andare a colpo sicuro sulla soluzione: qual è?

Proviamo con un'altra partita: il primo tentativo è lo stesso e riceve la stessa risposta, ma i successivi sono cambiati.

10110111	5
10001111	4
10111100	6
00111110	4
11111101	6

Anche in questo caso non resta che un codice coerente con le informazioni ricevute: quale?

Siamo stati fortunati ad avere la certezza di trovare la soluzione al sesto passo? C'è modo di scoprire il codice segreto in non più di 9 passi ovvero, generalizzando, in non più di  $n + 1$  passi, quando si gioca con sequenze di  $n$  bit?

**Soluzioni.** In entrambe le partite proposte il codice segreto è 10111001.

Supponiamo, per brevità, di giocare con sequenze di 5 bit.

Per scoprire il codice segreto entro 6 passi, prevediamo i seguenti tentativi, indicando con  $r_i$  la risposta all' $i$ -esimo tentativo ( $i = 1, \dots, 5$ ) e supponendo  $r_i < 5$ , altrimenti il codice è svelato:

00000	$r_1$ = numero di bit 0 nel codice segreto (c.s.)
10000	se $r_2 > r_1$ , il primo bit nel c.s. è 1, altrimenti è 0
11000	se $r_3 > r_2$ , il secondo bit nel c.s. è 1, altrimenti è 0
11100	se $r_4 > r_3$ , il terzo bit nel c.s. è 1, altrimenti è 0
11110	se $r_5 > r_4$ , il quarto bit nel c.s. è 1, altrimenti è 0

e a questo punto, conoscendo i primi quattro bit del c.s., si contano quanti tra questi sono 0: se tale numero è uguale a  $r_1$ , il quinto bit nel c.s. è 1, altrimenti è 0.

Notate che, dalla seconda in poi, ciascuna risposta differisce di un'unità (in più o in meno) dalla precedente, poiché è cambiato un solo bit rispetto al tentativo precedente. Dunque si può anche partire con un tentativo a caso e poi cambiare un bit alla volta in ognuno dei successivi. Questo procedimento impiega una strategia *decrease-and-conquer* (“diminisci e conquista”), che tuttavia, almeno nel caso del nostro gioco, non è ottima per ogni  $n$ . Ad esempio, proprio per  $n = 5$ , Dennis E. Shasha ha provato che qualsiasi codice di 5 bit può essere individuato dopo quattro tentativi al massimo, precisamente questi: 00000, 11100, 01110, 00101, poiché codici diversi determinano sequenze (di quattro risposte) diverse. Dal suo eccellente *Puzzles for Programmers and Pros* (Wrox Press, 2007) ho tratto questo esempio con  $n = 5$ , ma non è nota alcuna formula generale che dia il risultato ottimo per un  $n$  arbitrario.

Il gruppo Bebras del Portogallo propose nel 2016 una variante più difficile, ancora con  $n = 8$ : al decodificatore è comunicata la risposta soltanto quando azzecca 4 o 8 bit, negli altri casi la risposta è nulla. Come si può procedere? Se il primo tentativo (casuale) ha risposta nulla, invertiamo il primo bit, poi il secondo, e così via come sopra, finché non otteniamo la risposta 4 (naturalmente, se siamo fortunati, arriva prima la risposta 8, ma come al solito ci mettiamo nel caso peggiore). Consideriamo ora il codice X che ha dato risposta 4, in cui bisogna individuare i quattro bit sbagliati; se in X invertiamo sia il primo sia il secondo bit, i casi sono due: o la risposta è nulla, e allora i primi due bit di X sono o entrambi giusti o entrambi sbagliati, o la risposta è di nuovo 4, e allora uno dei primi due bit di X è giusto e l'altro è sbagliato.

Se poi procediamo allo stesso modo con il primo e il terzo bit di X, il primo e il quarto, e così via, possiamo scoprire e invertire i bit che non si accordano col primo. Infine, giungeremo ad avere una sequenza di 8 bit o completamente giusta o completamente sbagliata, sicché individueremo il codice segreto.

### 13. Chi andrà a giocare in cortile? Ovvero, la programmazione logica.

In questo paragrafo cercherò di esemplificare le principali idee sulle quali si fonda il paradigma della programmazione logica, riprendendo il discorso appena accennato alle pagine 56-57.

Iniziamo con un semplice quesito, ispirato da uno ancor più semplice proposto dal gruppo Bebras di Taiwan nel 2015.

Alla scuola di un paesino vi sono undici alunni di varie età: Amy, Bill, Charlie, Dan, Ellen, Fanny, Gary, Hilary, Ian, Jimmy e Kate.

La maestra non si fida a lasciar giocare da soli in cortile gli alunni più piccoli, per cui decide che:

- 1) se possono andare in cortile Kate e Bill, allora vi può andare anche Jimmy;
- 2) se può andare in cortile Hilary, allora vi può andare anche Ellen;
- 3) se possono andare in cortile Charlie e Dan, allora vi può andare anche Kate;
- 4) se può andare in cortile Amy, allora vi può andare anche Dan.

Per l'intervallo di stamane, la maestra dà il permesso di andare a giocare in cortile ad Amy, Charlie, Gary e Hilary. In base alle regole stabilite dalla maestra, chi potrà certamente unirsi a questi alunni e andare a giocare con loro in cortile?

**Soluzione.** Qui l'*universo* (cioè l'insieme degli elementi noti, di cui si vogliono esprimere certe proprietà) è costituito dagli undici alunni. Oltre al *predicato* che stabilisce chi è un alunno, il solo altro che qui interessa (anch'esso di arità 1, poiché riguarda ogni singolo alunno) esprime la possibilità di andare in cortile. Scriviamo allora un breve programma in PROLOG, dando il nome `alunno` al primo predicato e `va` al secondo: `alunno(X)` è vero se e soltanto se `X` “è un alunno” (o, meglio, “è il nome di un alunno”), `va(X)` è vero se e soltanto se `X` “può andare in cortile”.

```
alunno(amy).  
alunno(bill).  
alunno(charlie).  
alunno(dan).  
alunno(ellen).  
alunno(fanny).  
alunno(gary).  
alunno(hilary).  
alunno(ian).  
alunno(jimmy).  
alunno(kate).  
  
va(amy).  
va(charlie).  
va(gary).  
va(hilary).  
  
va(jimmy) :- va(kate), va(bill).
```

```

va(ellen) :- va(hilary).
va(kate)  :- va(charlie), va(dan).
va(dan)   :- va(amy).

```

Le prime quindici righe esprimono *fatti*, le altre quattro *regole*. Ad esempio, il primo fatto asserisce che Amy è un alunno (o alunna!), mentre la prima regola stabilisce che Jimmy può andare in cortile se può andarvi Kate e può andarvi Bill. La domanda che poniamo al sistema, il cosiddetto *goal*, è:

```
?- va(X).
```

L'intento è sapere con quali nomi di alunni può essere sostituita la variabile X (i nomi di variabile iniziano con una lettera maiuscola) per ottenere una *conseguenza logica* (una formula, nella fattispecie atomica e senza variabili, vera in tutti i modelli) della *teoria* i cui assiomi sono i fatti e le regole del nostro programma. Otteniamo:

```

X = amy ;
X = charlie ;
X = gary ;
X = hilary ;
X = ellen ;
X = kate ;
X = dan.

```

In questo caso, il sistema è in grado di fornire tutte le risposte che possiamo aspettarci: infatti, ai primi quattro alunni, si possono aggiungere Dan, Ellen e Kate.

Per la seconda regola, poiché *va(hilary)* è vero, deve pur esserlo *va(ellen)*.

Per la quarta regola, poiché *va(amy)* è vero, deve essere vero anche *va(dan)*.

Allora, per la terza regola, poiché *va(charlie)* e *va(dan)* sono veri, deve essere vero anche *va(kate)*.

Altro non si può dedurre. Qui la cosiddetta *base di Herbrand*<sup>2</sup> è formata da ventidue *atomi* (senza variabili):

```
alunno(amy), ..., alunno(kate), va(amy), ..., va(kate),
```

vale a dire, tutto ciò che *può essere vero*.

Il *minimo modello* (di Herbrand) è costituito da tutte le *conseguenze logiche* di fatti e regole, e cioè da tutto e soltanto ciò che, date le premesse, *deve essere vero necessariamente*; nel nostro caso, diciotto di quei ventidue atomi...

E di Bill, Fanny, Ian e Jimmy che cosa si può dire, oltre che sono alunni? Se potessimo asserire la falsità del predicato *va* su ciascuno di loro, concluderemmo che non possono andare in cortile. In realtà, un modello di questa teoria è anche l'intera base di Herbrand, per cui pur essi potrebbero andare in cortile; fra l'altro, la maestra non ha menzionato né Fanny né Ian.

---

<sup>2</sup> Dal nome del matematico francese Jacques Herbrand, che nel 1931 scrisse un importante studio sulla teoria della dimostrazione e delle funzioni ricorsive, pur coerente con l'indecidibilità di Gödel.

Un’importante osservazione: questa teoria ha almeno un modello (l’intera base di Herbrand), ossia è *consistente*.

Come funziona un sistema di deduzione automatica che opera su teorie costituite da un insieme finito di fatti e regole di questo tipo, ma in generale con variabili? Assiomi siffatti sono *clausole di Horn positive* (o *definite*; Alfred Horn, 1951), ciascuna delle quali è la *disgiunzione logica* (*or* inclusivo) di atomi, di cui uno soltanto non è sotto negazione, e tutte le variabili che vi occorrono sono quantificate universalmente all’esterno della disgiunzione. I fatti sono precisamente quelle clausole con un atomo soltanto, che quindi non è sotto negazione. Vediamo di capire bene quanto ora detto, tramite alcuni esempi.

Sia P1 il programma costituito dal fatto

```
uomo(socrate).
```

col significato “il predicato uomo è vero sulla costante socrate”, e dalla regola

```
mortale(X) :- uomo(X).
```

che corrisponde alla clausola  $\forall X (\sim \text{uomo}(X) \vee \text{mortale}(X))$ , logicamente equivalente a  $\forall X (\text{uomo}(X) \rightarrow \text{mortale}(X))$ , col significato “per ogni  $X$ , se il predicato uomo è vero su  $X$ , allora anche il predicato mortale è vero su  $X$ ”.

La *variabile logica*  $X$  può assumere valore nell’*universo di Herbrand* (l’insieme dei *termini senza variabili*, che – in assenza di identità fra di essi – è in corrispondenza biunivoca con l’insieme degli oggetti di cui si sta parlando) costituito in questo caso dalla sola costante socrate; la *base di Herbrand* è invece formata da tutti gli *atomi senza variabili*, ottenibili applicando i predicati dati agli elementi di tale universo: qui sono due soltanto, uomo(socrate) e mortale(socrate).

Sia P2 il programma costituito dai fatti

```
p(a,b).
```

col significato “il predicato (questa volta di arità 2) p è vero sulla coppia (ordinata) di costanti (a, b)”, e

```
q(X,X).
```

che corrisponde alla clausola  $\forall X q(X, X)$ , col significato “per ogni  $X$ , il predicato (di arità 2) q è vero sulla coppia (X, X)”, e dalla regola (attenzione: *ricorsiva*)

```
q(X,Y) :- p(X,Z), q(Z,Y).
```

che corrisponde alla clausola  $\forall X \forall Y \forall Z (\sim p(X, Z) \vee \sim q(Z, Y) \vee q(X, Y))$ , che è una clausola di Horn positiva (l’ultimo atomo, infatti, non è sotto negazione) logicamente equivalente a  $\forall X \forall Y \forall Z (p(X, Z) \wedge q(Z, Y) \rightarrow q(X, Y))$  e anche a  $\forall X \forall Y (\exists Z (p(X, Z) \wedge q(Z, Y)) \rightarrow q(X, Y))$ , col significato “per ogni  $X$  e per ogni  $Y$ , se esiste  $Z$  tale che il predicato p è vero sulla coppia (X, Z) e il predicato q è vero sulla coppia (Z, Y), allora il predicato q è vero anche sulla coppia (X, Y)”.

Le variabili logiche (anche *z*, che è *locale al corpo* della regola) non hanno tipo, o meglio sono tutte di un unico tipo, e ciascuna di esse può essere sostituita – oltre che con una nuova variabile “fresca” – con un elemento nell’universo di Herbrand, che in questo caso consiste di due termini costanti: *a* e *b*; la base di Herbrand è formata dagli atomi senza variabili, che sono:

*p(a, a), p(a, b), p(b, a), p(b, b), q(a, a), q(a, b), q(b, a), q(b, b)*.

Più in generale, un fatto è costituito da un *singolo atomo*, anche con variabili che s’intendono tutte quantificate universalmente, mentre una regola è costituita da una *testa* (o *conclusione* o *conseguente*: un singolo atomo, con variabili) e un *corpo* (o *condizione* o *antecedente*: un atomo o una *congiunzione logica, and*, di atomi, con variabili) e corrisponde alla clausola che può essere così descritta: tutte le variabili che compaiono nella testa sono quantificate universalmente all’esterno, e il corpo (preceduto dalle eventuali altre variabili, quelle locali, quantificate esistenzialmente) implica logicamente la testa. Un fatto è dunque una clausola (positiva) che ha soltanto la testa; infatti il suo corpo è vuoto. Conveniamo che tutti e soli i nomi di variabili inizino con una lettera maiuscola; il loro campo d’azione lessicale è comunque limitato a una sola clausola.

Insieme con un programma, i sistemi di deduzione accettano un *goal*, costituito da una *clausola di Horn negativa non vuota* (in altre parole, una regola priva di testa o, meglio, col *falso* come testa) che è la *negazione logica* della formula da dimostrare (o altrimenti, possibilmente, da refutare) nella teoria espressa dal programma.

Ad esempio, col programma P1 si può dare da dimostrare la formula atomica senza variabili *mortale(socrate)*, che può essere dedotta per *modus ponens* dopo aver sostituito con la costante *socrate* la variabile logica che compare nella regola.

In effetti, alla domanda

```
?- mortale(socrate).
```

un sistema PROLOG risponde yes, perché l’atomo *mortale(socrate)* “unifica” (“fa match”: cfr. p. 41) con la testa della regola, e quindi la dimostrazione si riduce a quella dell’atomo *uomo(socrate)*, che è un fatto, sicché si giunge alla clausola *vuota*: non si ha più nulla da provare; analogamente, alla domanda

```
?- mortale(V).
```

il sistema risponderà soltanto *V = socrate*, perché tentando di provare *uomo(V)* non ha alcun’altra possibilità di *istanziare* la variabile *V* (cioè di sostituirla con una costante) per giungere nuovamente alla clausola vuota, mentre alla domanda

```
?- mortale(io).
```

risponderà no, perché può “accorgersi” che l’atomo *mortale(io)* non è deducibile nella teoria espressa dal programma: *uomo(io)* non unifica con alcun fatto o testa di regola; tra l’altro, la costante *io* neppure appartiene all’universo di Herbrand del programma, *goal* escluso.

Vediamo un esempio meno banale, col programma P2, ove il predicato  $q$  altro non è che la *chiusura riflessiva e transitiva* del predicato  $p$ . Se poniamo la domanda

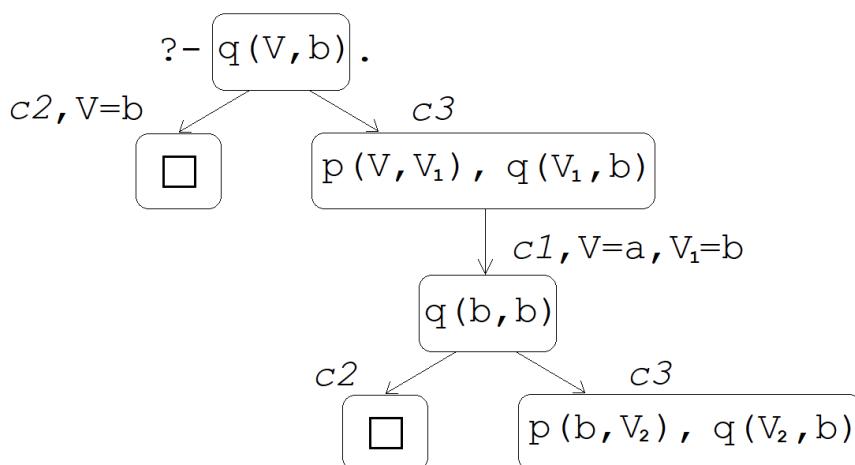
?-  $q(V, b)$ .

che corrisponde a chiedere se, nella teoria espressa dal programma, è possibile dimostrare (costruttivamente) la formula  $\exists V q(V, b)$  (producendone, in caso di riuscita, almeno un *testimone*), la risposta che ci possiamo attendere è affermativa; due infatti sono le possibilità di istanziare la variabile logica  $V$  per provare *vera* (in tutti i modelli del programma) la formula da dimostrare:  $V = b$  e  $V = a$ . Si osservi che la negazione della formula da dimostrare (il nostro *goal*) è  $\sim \exists V q(V, b)$ , equivalente a  $\forall V \sim q(V, b)$ , che è una clausola di Horn negativa.

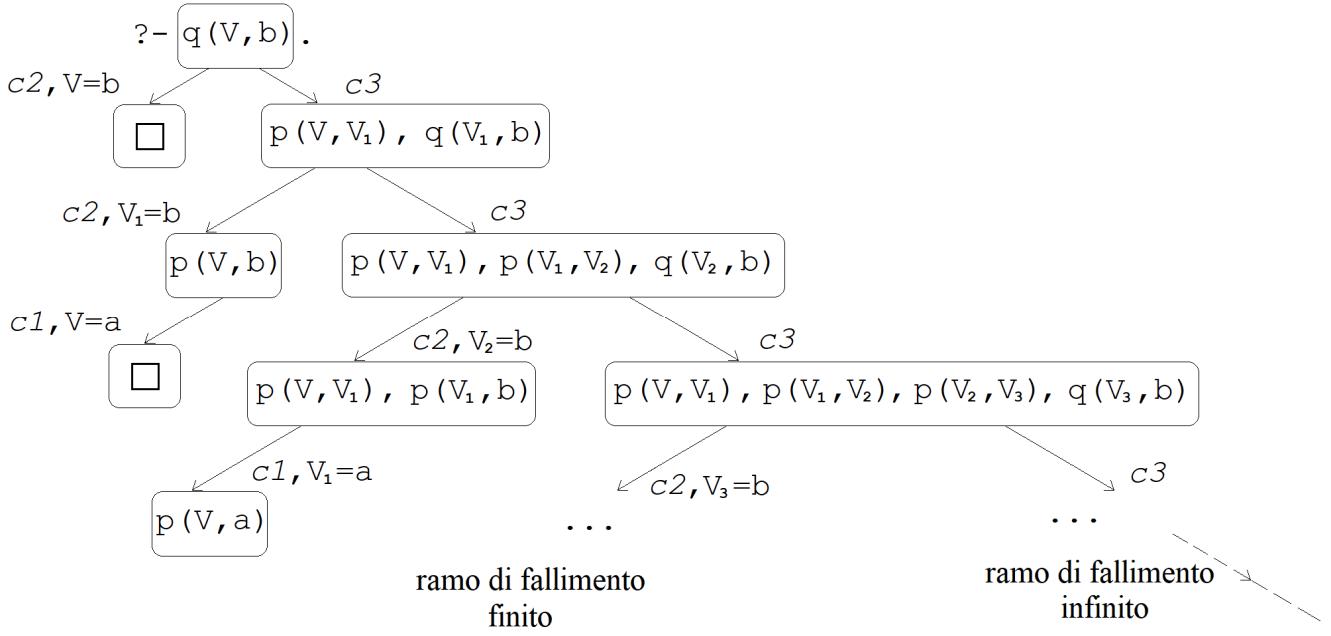
Più in generale, per cercare di raggiungere l'obiettivo, un sistema PROLOG procede applicando ripetutamente due meccanismi:

- unificazione* (il più possibile generale, per non perdere alcuna opportunità) di un atomo che compare nel *goal* corrente (atomo che è detto *subgoal* ed è scelto mediante una *regola di selezione*) con un fatto o con una testa di regola (da scegliere tra i possibili mediante una *regola di ricerca*),
- successiva *sostituzione* nel *goal* corrente.

Il processo del calcolo si riduce così a una particolare forma di procedimento dimostrativo (costruttivo) di teoremi, basato sull'esplorazione di un *albero*, da ogni nodo del quale si dipartono tanti archi quante sono le possibili unificazioni dell'atomo selezionato (*subgoal*) con un fatto o una testa di regola; se per tale *subgoal* non vi sono unificazioni allora il nodo è una *foglia di fallimento*, alla estremità di un *ramo di fallimento di lunghezza finita*. A seconda della regola di selezione adottata, si possono ottenere alberi *diversi* (anche prescindendo dall'ordine dei rami): nel nostro esempio, se l'atomo scelto è il primo a sinistra si ottiene l'albero disegnato qui sotto, mentre se l'atomo scelto è il primo a destra si ottiene l'albero alla pagina successiva.<sup>3</sup>



<sup>3</sup> Si osservi che, ogni volta che si applica o si usa una clausola, questa è istanziata con variabili “fresche”. Ciascuna clausola è indicata con  $c$  e numero d'ordine nell'elenco delle clausole.



Quest'ultimo albero ha un *ramo di fallimento infinito* (e infiniti rami di fallimento di lunghezza finita): in effetti, la regola di selezione influenza soltanto il *numero* dei rami di fallimento, includendo quelli infiniti.

Si tratta di un metodo *corretto* di ragionamento, poiché sono deducibili soltanto fatti veri in tutti i modelli della teoria espressa dal programma. Tuttavia, usualmente, i sistemi PROLOG adottano una regola di selezione *leftmost* (ossia, il primo atomo a sinistra) e una regola di ricerca *depth-first* (ossia, la prima clausola “in profondità”) secondo l’ordinamento delle clausole stabilito dalla sequenza in cui esse sono scritte nel programma: in tal modo, è certo che *non si possa ottenere la completezza!*

Infatti, la *regola di ricerca* determina il *criterio di visita* degli alberi al fine di cercare i *nodi-foglia di successo*,<sup>4</sup> corrispondenti alla *clausola vuota* (che rappresenta il *falso*).<sup>5</sup> Così, ad esempio, nell’albero raffigurato qui sopra, se si forza il sistema alla ricerca di ulteriori nodi di successo dopo i primi due trovati, esso cade in *loop* (e fin qui poco male, visto che altre risposte non si possono dare); ma se, maldestramente, avessimo scritto *per prima* la terza clausola nella forma

$q(X, Y) :- q(Z, Y), p(X, Z).$

(si noti che il significato logico del programma non cambia affatto) allora il sistema

<sup>4</sup> Per inciso, se il numero di tali nodi è finito, allora è lo stesso in tutti gli alberi, indipendentemente dalla regola di selezione adottata. Risalendo il ramo da uno qualsiasi di questi nodi di successo fino alla radice dell’albero, si ottiene una dimostrazione costruttiva (*backward*).

<sup>5</sup> Mettendo insieme le clausole del programma (che almeno un modello ce l’hanno, quello in cui tutti gli atomi appartenenti alla base di Herbrand del programma sono veri) con il *goal*, si ottiene il *falso*: dunque la teoria che consta degli assiomi del programma più il *goal* non ha modelli, dunque la negazione del *goal* (cioè la formula da dimostrare) è vera in tutti i modelli del programma. Infatti, poiché ammettere che la formula da dimostrare sia falsa conduce all’assurdo (clausola vuota), il principio aristotelico del terzo escluso permette di concludere che è vera: e proprio per questo il procedimento è detto “di refutazione”.

non troverebbe alcuna risposta, perché sarebbe percorso subito il ramo di fallimento infinito. E non si pensi di riuscire a cavarsela comunque, grazie a un particolare ordinamento delle clausole e/o degli atomi nei corpi delle regole! A tal proposito, ecco un classico esempio di semplice teoria, nella quale tuttavia non riusciamo a ottenere dimostrazioni di teoremi piuttosto elementari.

Programma P3:

```
p(a,b).
p(c,b).
p(X,Y) :- p(Y,X).
p(X,Y) :- p(X,Z), p(Z,Y).
```

*Goal:*

```
?- p(a,c).
```

Le due regole ricorsive (terza e quarta clausola) stabiliscono, rispettivamente, le proprietà di *simmetria* e di *transitività* per il predicato *p*. La dimostrazione della formula atomica (senza variabili) *p(a,c)* è breve: dalla seconda e dalla terza clausola si ottiene il teorema *p(b,c)*; da questo, dalla prima e dalla quarta clausola si ottiene il teorema *p(a,c)*, ciò che si voleva dimostrare. Tuttavia, con una regola di ricerca *in profondità* basata su un ordine comunque *fissato* per le quattro clausole, il sistema non è in grado di raggiungere il successo; e ciò vale anche se si scambiano i due atomi nel corpo della quarta clausola scritta sopra.

Stabilire sia una regola di selezione sia una regola di ricerca determina una *procedura di refutazione* (possibilmente con *backtracking*). Se, ad esempio, la regola di ricerca adottata è *breadth-first*, secondo la quale gli alberi sono *visitati per livelli* anziché in profondità (e quindi senza bisogno di “tornare indietro”), allora la procedura è *completa*, nel senso che se un nodo di successo esiste prima o poi è raggiunto. In altre parole: se la formula da dimostrare è *conseguenza logica* del programma (ossia, è *vera in tutti i modelli* del programma, cioè in tutte le strutture nelle quali tutte le proprietà espresse dalle clausole del programma valgono), allora la procedura ne riesce a costruire una dimostrazione. Ciò porta a un risultato di *completezza* – che si aggiunge alla correttezza – per questa tecnica di risoluzione.

Di solito, tuttavia, per motivi di efficienza, i sistemi PROLOG non adottano il criterio di visita per livelli, bensì – ribadiamo – quello in profondità, perdendo così la completezza.

### 13.1. Il problema della negazione.

Procediamo affrontando un’altra questione delicatissima: la negazione. Riprendendo il quesito iniziale, supponiamo che la maestra decida così: se possono andare in cortile almeno due alunni, allora vi può andare anche Fanny. Come può essere scritta la corrispondente regola sotto forma di clausola? Di certo sbagliamo se scriviamo:

```
va(fanny) :- va(X), va(Y).
```

perché nulla vieta di sostituire le due variabili con uno stesso alunno – ma non solo!

Dunque, se il linguaggio che usiamo permette l'applicazione di un predicato `not` a un atomo, potremmo introdurre un predicato di “identità” e scrivere:

```
stesso(X,X).  
va(fanny) :- va(X), va(Y), not(stesso(fanny,X)),  
           not(stesso(fanny,Y)), not(stesso(X,Y)).
```

A meno che non si espliciti per tutte le coppie di alunni, per esprimere la diversità occorre il `not`... ma quest'ultima non è una clausola di Horn positiva! Infatti, quando sono scritte come implicazioni logiche, le clausole di Horn positive non contengono alcuna negazione. Come vedremo, la soluzione è corretta soltanto se le variabili sono sostituite con termini *costanti* quando sono valutati gli atomi sotto negazione.

È comunque opportuno riflettere su un aspetto: quando si rappresentano informazioni sotto forma di fatti *senza variabili* in PROLOG (o di righe di tabelle in una base di dati), tali informazioni sono sempre “positive”. Ad esempio, postulando il fatto `p(a,b)` col significato “il predicato `p` è vero sulla coppia di costanti `(a,b)`”, si fa una affermazione comunque “in positivo”, indipendentemente dal significato che si vuole attribuire al predicato stesso: potrebbe significare tanto che l’oggetto rappresentato dalla costante `a` “è posato sopra” l’oggetto rappresentato dalla costante `b` (oggetto che è certamente distinto dal precedente!), quanto che il primo “non è posato sopra” il secondo. Si può allora, e semmai come, dedurre informazione “negativa” rispetto a quella “positiva” rappresentata?

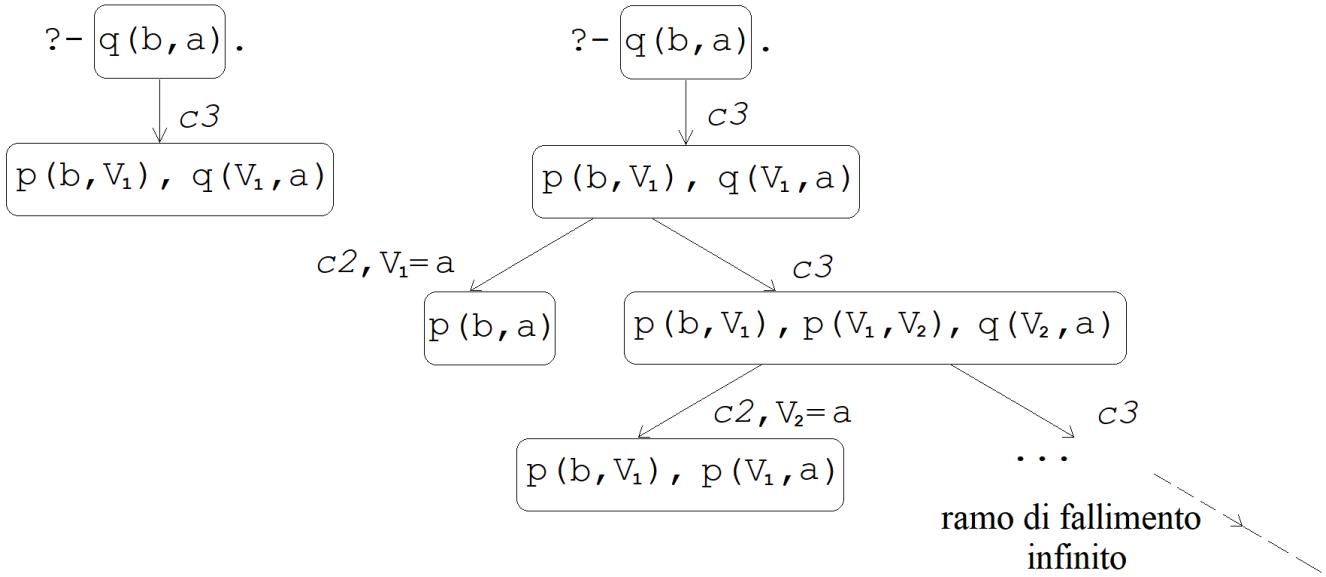
Nel 1978, in una stessa raccolta di lavori su logica e basi di dati, furono pubblicate due idee, entrambe formalizzabili come regole di inferenza, ma non equivalenti poiché possono dar luogo a esiti diversi. Una, nota come “ipotesi di mondo chiuso” (*Closed World Assumption*, o in breve CWA), fu proposta da Raymond Reiter nel contesto dei *databases deduttivi*, quale criterio per caratterizzare appunto le informazioni “negative” rispetto a tutto ciò che è esplicitamente asserito e registrato in una base di dati; nell’ambito della programmazione logica, la corrispondente regola di inferenza è questa: se un atomo, appartenente alla base di Herbrand di un programma, non può essere provato nella teoria espressa dal programma stesso con alcuna procedura di refutazione, allora la negazione di questo atomo è ritenuta vera (in tutti i modelli del programma). Sfortunatamente, a questa definizione non corrisponde un metodo di ragionamento *effettivo*: ciò vuol dire che, in generale, l’insieme di detti atomi non è *ricorsivamente enumerabile* (cfr. nota a piè di p. 39). L’altra idea, che conduce a una forma di “non provabilità” che può tuttavia rivelarsi più restrittiva, è detta “negazione come fallimento (sottinteso finito)” (*Negation as Failure*, o più semplicemente NF) e fu introdotta da Keith L. Clark quale strumento *effettivo* per i programmi logici: se per un atomo, appartenente alla base di Herbrand di un programma, esiste un albero, secondo una qualche regola di selezione, che abbia in radice il *goal* costituito soltanto da tale atomo e che sia *finitamente fallito*, allora la negazione di questo atomo è ritenuta vera (in tutti i modelli del programma).

Un albero si dice finitamente fallito se è finito e privo di nodi di successo. In altre parole: è un albero che ha *tutti* i rami di lunghezza finita e di fallimento. Invece, un *albero di successo* (finito o infinito che sia) ha almeno un nodo (foglia) di successo. Clark propose quindi la nozione di “completamento” (*completion*) del programma logico considerato (che, come si è detto, è costituito esclusivamente da clausole positive), quale descrizione del suo significato.

Riprendiamo il programma P2, questa volta con la domanda

?–  $q(b, a)$ .

Se l’atomo scelto è il primo a sinistra si ottiene l’albero disegnato sotto a sinistra, che ha un unico ramo che è di fallimento e di lunghezza finita, mentre se l’atomo scelto è il primo a destra si ottiene l’albero disegnato a destra, anch’esso giustamente privo di nodi di successo, che ha infiniti rami di fallimento di lunghezza finita e un ramo di fallimento infinito: l’esplorazione dell’albero alla ricerca di un eventuale nodo di successo *non termina*, qualunque sia il criterio di visita adottato! Siccome però c’è un albero (quello a sinistra) finitamente fallito, allora la negazione dell’atomo  $q(b, a)$  è considerata vera secondo la NF, e quindi anche secondo la CWA. La regola di selezione dell’atomo da unificare – che è importante dal punto di vista del fallimento dei *goal*, poiché, come si è visto, da essa può dipendere il numero dei rami di fallimento, compresi quelli infiniti – assume quindi una grande rilevanza ai fini della negazione; torneremo tra breve su questo punto.



Riassumendo: nel PROLOG puro (senza negazione, al quale finora abbiamo fatto riferimento), la teoria espressa da un qualsiasi programma è *consistente*, cioè ha almeno un modello, quello in cui tutti gli atomi appartenenti alla base di Herbrand del programma sono veri. L’*insieme di successo* di un programma, costituito dagli atomi appartenenti alla sua base di Herbrand che possono essere provati con una qualche procedura di refutazione, è uguale al suo *minimo modello di Herbrand*, definito come l’insieme di tutti gli atomi senza variabili che sono conseguenza logica

del programma stesso.<sup>6</sup> Si dimostra facilmente che tale modello minimo esiste ed è unico. Seri problemi si pongono tuttavia se si vuole introdurre la *negazione logica*, anche semplicemente di una formula atomica (nel *goal* o nel corpo di una regola del programma) che vorremmo poter scrivere come argomento di un predicato *not*.

Il problema della negazione in programmazione logica è sorto principalmente dalla ricerca sulle basi di dati deduttive. Vediamo un piccolo esempio, considerando il programma P4 costituito dai seguenti quattro fatti, ai quali si vuole attribuire l'ovvio significato:

```
pesce(luccio).  
pesce(trota).  
mammifero(topo).  
mammifero(elefante).
```

Non è possibile dimostrare la *negazione* dell'atomo *pesce (topo)*: infatti, esiste un modello del programma in cui il topo, oltre a essere un mammifero, è anche un pesce. Non è possibile comunque dimostrare l'atomo *pesce (topo)*, cioè che il topo è un pesce: infatti, esiste un modello del programma (quello “naturale” che, erroneamente, si sarebbe indotti a credere l'unico!) in cui non lo è.

Se si considera il programma P4 come una *base di dati*, allora può essere conveniente riuscire a dimostrare che il topo non è un pesce, e così pure l'elefante, e che il luccio e la trota non sono mammiferi. Vale a dire, in generale: *tutto ciò che non è asserito esplicitamente, né può essere desunto dai dati, è da ritenere falso*.

Nel caso d'esempio, vorremmo che risultassero vere le seguenti negazioni di atomi:

```
not(pesce(topo))  
not(pesce(elefante))  
not(mammifero(luccio))  
not(mammifero(trota))
```

sicché il modello “ovvio” sarebbe davvero l'unico! Ma questo, in generale, significa usare la CWA: se A (simbolo che qui usiamo per indicare un generico atomo *senza variabili*) non è una conseguenza logica del programma, allora è possibile inferire *not (A)*.<sup>7</sup> Purtroppo, ribadiamo, se A non è conseguenza logica del programma, allora il procedimento di risoluzione con il *goal* iniziale  $?- A$ . può non terminare, e

---

<sup>6</sup> “Minimo” poiché nessun suo sottoinsieme proprio è un modello della teoria espressa dal programma considerato, mentre ogni altro modello lo contiene in senso stretto; si può anche dire che è il modello di *minima verità*, in quanto in esso è vero soltanto ciò che necessariamente deve essere vero in ciascun modello della teoria considerata.

Si noti che, essendo unico il tipo, l'insieme *supporto* di ciascun modello avrà tanti elementi quanti sono quelli dell'universo di Herbrand del programma: infatti, in assenza di identità, nomi di costanti diversi si devono intendere riferiti a oggetti distinti.

<sup>7</sup> Contrapposta alla CWA è la OWA (dove O sta per *Open*): tutto ciò che non è conseguenza logica degli assiomi è da ritenersi *sconosciuto*. In tal modo il ragionamento preserva la monotonicità, poiché l'aggiunta di nuova informazione non può falsificare una precedente conclusione.

pertanto, a livello operativo, la CWA non riesce a dire nulla! Nella pratica, l'applicazione della CWA è ristretta agli atomi la cui prova fallisce in tempo finito. Quindi, per provare  $\text{not}(A)$ , si controlla se il *goal*  $?- A$ . *fallisce finitamente*; ma nel far questo, come si è intuito, entra in gioco in modo determinante la regola di selezione: a livello operativo, basta seguire una regola “imparziale” (*fair*), che garantisca che ciascun possibile *subgoal*, ossia atomo che compare (eventualmente in congiunzione logica con altri atomi) in un qualsiasi nodo (non foglia di fallimento), sia prima o poi selezionato. Ovvero si può decidere di provare, ad ogni passo, *tutte* le possibili scelte “in parallelo”: se una sequenza di scelte porta a un albero finitamente fallito, allora prima o poi si riuscirà a concludere!<sup>8</sup>

Questo criterio è giustificato rispetto alla teoria in cui vi sono *equivalenze logiche* al posto delle implicazioni (*completamento* del programma); ad esempio, nel caso del programma P4:

```
pesce(X) ↔ (X = luccio) ∨ (X = trota)
mammifero(X) ↔ (X = topo) ∨ (X = elefante)
```

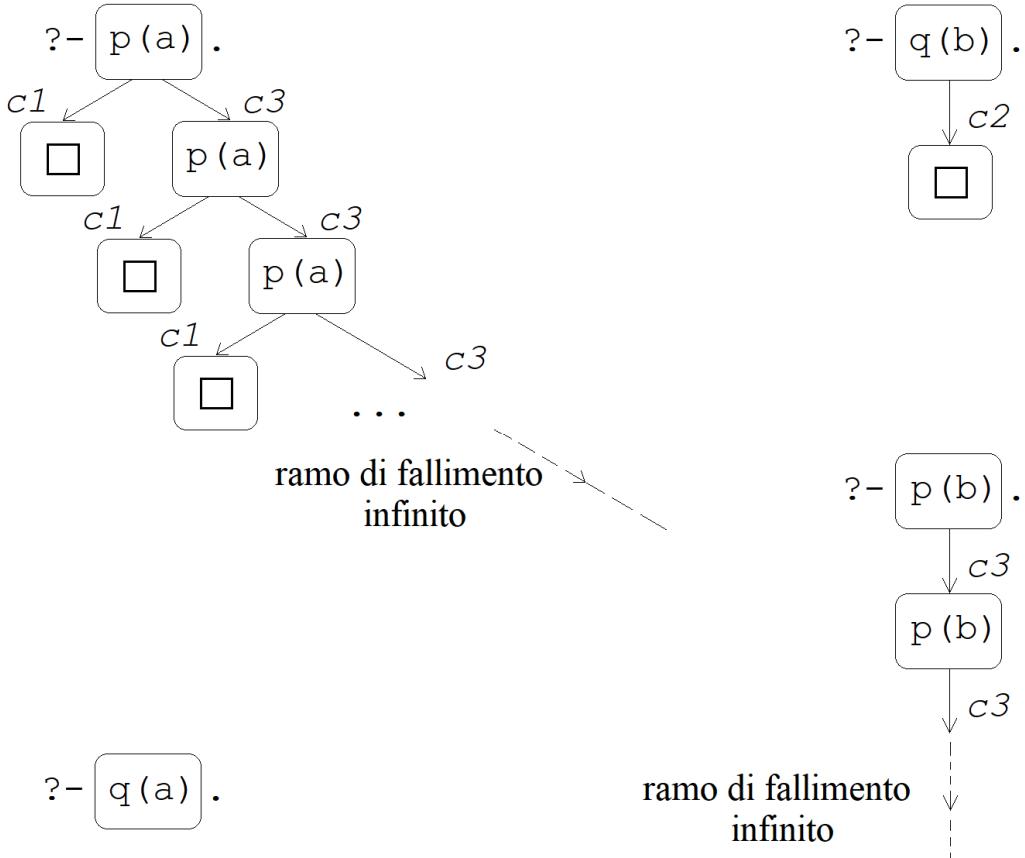
dove il predicato di uguaglianza deve essere interpretato come la relazione d'identità sull'universo di Herbrand (cfr. stesso introdotto a pagina 383):  $X = \text{luccio}$  è vero se e soltanto se la variabile  $X$  è sostituita con la costante  $\text{luccio}$ . Di solito, completando un programma, i modelli diminuiscono; certamente non aumentano! Vediamo ancora un esempio, assai semplice ma concettualmente significativo. Sia P5 il programma costituito dalle seguenti tre clausole:

```
p(a).
q(b).
p(X) :- p(X).
```

Allora, osservando i quattro casi raffigurati alla pagina successiva, consideriamo i *goal* in cui compare un singolo atomo, appartenente alla base di Herbrand di questo programma: i *goal*  $?- p(a)$ . e  $?- q(b)$ . hanno successo (l'albero che ha in radice  $?- p(a)$ . ha anche un ramo infinito: attenzione dunque al criterio di visita!); il *goal*  $?- q(a)$ . fallisce finitamente (per cui si può ritenere vera la negazione di  $q(a)$ ); anche il *goal*  $?- p(b)$ . ha un unico albero, che però è infinito e con un solo ramo... Secondo la CWA si può quindi ritenere vera la negazione di  $p(b)$ , tuttavia un metodo operativo non può concludere: ecco un caso, piuttosto banale, catturato dalla CWA ma non dalla NF!

---

<sup>8</sup> Una semplice regola *fair* fu data da John W. Lloyd: è selezionato l'atomo più a sinistra che si trova alla destra della lista (eventualmente vuota) di atomi introdotta al passo di derivazione precedente, se tale atomo esiste; altrimenti è selezionato l'atomo più a sinistra. Si noti che, in generale, anche con una regola di selezione *fair*, una ricerca *in profondità* può non concludere (correttamente)... e quindi non dovrebbe essere comunque adottata! Ad esempio, si veda ancora l'albero a pagina 381, relativo al programma P2: applicando due volte la terza clausola, prima all'unico *subgoal* e dopo all'atomo a destra, si ottiene un *goal* ove – quand'anche sia poi selezionato l'atomo a sinistra o quello centrale – ormai, continuando a scendere, non vi è più la possibilità di trovare la clausola vuota!



La regola NF è realizzabile abbastanza facilmente ed è quella adottata, ad esempio, nei sistemi PROLOG, ma con i limiti che solitamente questi presentano riguardo a regole di selezione e ricerca, ad esclusivo favore dell'efficienza.

Comunque, una risoluzione, che procede con la regola di ricerca *breadth-first* e con una regola di selezione *fair*, è una realizzazione *corretta e completa* della nozione di “fallimento finito” per gli atomi senza variabili le cui negazioni sono *conseguenza logica del completamento* del programma considerato.

Per risolvere *goal generali*, cioè che possono contenere anche *negazioni* di atomi (eventualmente con variabili), il classico procedimento di risoluzione è stato combinato con la regola di negazione come fallimento finito (Clark, 1978).

Anzitutto, si adotta una regola di selezione “sicura” (*safe*): si sceglie una negazione di atomo soltanto quando l’atomo è completamente istanziato (ossia senza variabili); poi, se dal *goal* G è scelto *not (A)* (con A senza variabili), si tenta di costruire a parte un albero finitamente fallito con il *goal*  $?- A$ . prima di continuare con il resto della computazione. Quindi:

i) se si trova un albero finitamente fallito con il *goal*  $?- A$ . (e, come abbiamo visto, se un tale albero esiste è effettivamente possibile costruirlo), allora *not (A)* si considera dimostrato con successo (e dunque vero), e perciò al successivo passo di derivazione si ha un nuovo *goal*, ottenuto da G eliminandovi *not (A)*;<sup>9</sup>

<sup>9</sup> Si noti, *senza applicare alcuna sostituzione*: il soddisfacimento di *not (A)* (pur con variabili) non può provocare istanziazioni di variabili, perché è conseguenza del fallimento (finito) del *goal*  $?- A$ .

*ii)* se il *goal*  $?- A$ . ha successo (e, come abbiamo visto, se un nodo di successo esiste, è effettivamente possibile raggiungerlo), allora la dimostrazione di  $\text{not}(A)$  fallisce e quindi, con essa, fallisce tutto il *goal* G (dato che la componente selezionata  $\text{not}(A)$ , se non è l'unica, è in congiunzione logica con altre componenti, cioè con atomi o negazioni di atomi);

*iii)* c'è tuttavia la possibilità di non terminazione: quando il *goal*  $?- A$ . non ha successo, ma per tale *goal* non esiste alcun albero finitamente fallito.

La "sicurezza", ossia la scelta di negazioni di atomi soltanto quando non vi sono variabili, è necessaria per garantire anche la sola *correttezza* del procedimento di risoluzione rispetto a *goal* generali. Purtroppo, per le note ragioni di efficienza, di solito nei sistemi PROLOG la selezione è *leftmost* (regola *non fair*), pur nel caso in cui più a sinistra si trovi la negazione di un atomo con variabili non istanziate (regola *non safe*): e questa è certamente una realizzazione *non corretta* della procedura! Vediamo, a questo proposito, un paio di esempi.

Programma P6:

```
capitale(roma).
capoluogo(genova).
citta(X) :- capitale(X).
citta(X) :- capoluogo(X).
```

*Goal G* (vogliamo sapere se è nota qualche città che non sia capitale...):

```
?- not(capitale(X)), citta(X).
```

Se la regola di selezione è *leftmost*, dal *goal* G è scelta la negazione dell'atomo *capitale(X)*; ma il *goal*  $?- \text{capitale}(X)$ . ha successo con la sostituzione calcolata  $X = \text{roma}$ , e quindi il *goal* G fallisce! Se ci si fermasse qui, come di solito accade con i sistemi PROLOG, si perderebbe la completezza: infatti G corrisponde alla negazione della formula  $\exists X (\sim \text{capitale}(X) \wedge \text{citta}(X))$ , per la quale esiste l'istanziazione  $X = \text{genova}$  tale che  $\sim \text{capitale}(\text{genova})$  è derivabile da P6 tramite NF e *citta(genova)* è conseguenza logica di P6; si ottiene infatti una refutazione per il *goal*  $?- \text{not}(\text{capitale}(\text{genova}))$ , *citta(genova)* ..

Il problema nasce ancora dal fatto che, quando è selezionata la negazione di un atomo con variabili, la quantificazione non è correttamente interpretata: scegliendo *not(capitale(X))* si tenta di verificare a parte il fallimento finito del *goal*  $?- \text{capitale}(X)$ ., ciò che porterebbe a concludere  $\sim \exists X \text{ capitale}(X)$  ossia  $\forall X \sim \text{capitale}(X)$ .

Con la regola di selezione *leftmost*, la sostituzione  $X = \text{genova}$  sarebbe in effetti calcolata come risposta alla domanda  $?- \text{citta}(X)$ , *not(capitale(X))*.: la variabile che compare nell'atomo sotto negazione sarebbe infatti istanziata dalla valutazione di un altro atomo.

Sia ora P4' il programma ottenuto da P4 aggiungendovi i seguenti fatti:

```
mammifero(gatto).  
mangia(gatto,topo).  
mangia(luccio,trota).
```

Vorremmo sapere se esiste qualche animale che non è mangiato dal gatto. Tuttavia, alla domanda `?- not(mangia(gatto,X))`. un sistema PROLOG risponde no, perché il *goal* `?- mangia(gatto,X)`. ha successo; in realtà le negazioni di atomi

```
~mangia(gatto,luccio)  
~mangia(gatto,trota)  
~mangia(gatto,elefante)  
~mangia(gatto,gatto)
```

sono conseguenze logiche del completamento del programma P4'.

Assumendo la regola di selezione *leftmost*, alle seguenti domande il sistema risponde come a fianco riportato:

<code>?- mangia(gatto,luccio).</code>	no
<code>?- not(mangia(gatto,luccio)).</code>	yes
<code>?- mangia(gatto,X).</code>	<code>X = topo</code>
<code>?- not(mangia(gatto,X)), pesce(X).</code>	no (conclusione errata! Infatti...)
<code>?- pesce(X), not(mangia(gatto,X)).</code>	<code>X = luccio o X = trota</code>
<code>?- mammifero(X), not(mangia(gatto,X)).</code>	<code>X = elefante o X = gatto</code>

Anche qui, per rispondere alle ultime due domande, la variabile che compare nell'atomo sotto negazione è istanziata dalla valutazione di un altro atomo, a causa della regola di selezione *leftmost*, e quindi la risposta è corretta.

Infine, alla domanda `?- not(mangia(X,gatto))`. il sistema risponde semplicemente yes, che è corretto, senza però produrre alcun testimone (*cfr.* nota a piè di pagina 387).

È doveroso precisare che un termine, in PROLOG puro, può sì essere una costante o una variabile, ma può anche essere costruito con un *funtore* di arità  $n \geq 1$  applicato a  $n$  termini (*cfr.* p. 57), ciò che rende il linguaggio computazionalmente completo.<sup>10</sup>

Un buon ambiente per sviluppare programmi logici, e provare ciò che abbiamo visto in questo paragrafo, è SWI-Prolog, realizzato presso l'Università di Amsterdam e liberamente accessibile al sito <http://www.swi-prolog.org/>.

Passiamo finalmente a considerare il caso più generale in cui anche le regole di un programma possano avere negazioni di atomi (con variabili) nel loro corpo. Interpretando sempre la negazione come fallimento finito e ponendo opportune restrizioni sulla forma di programma e *goal*, si ottengono anche in questo caso dei risultati di correttezza e completezza del calcolo.

---

<sup>10</sup> Nei sistemi PROLOG comunemente disponibili, il linguaggio (non più puro) comprende anche: usuale aritmetica, operazioni di confronto, predicati metalogici, direttiva *cut* e predicati extra-logici, nonché operazioni di input/output e tante altre cose di utilità per l'utente.

## 13.2. Programmi logici generali e *database deduttivi*.

Colgo quest'occasione per accennare anche alle *basi di dati deduttive* e a Datalog, un linguaggio (sostanzialmente, un sottoinsieme del PROLOG non puro, con certe restrizioni) che, mediante regole, permette di definire “tabelle derivate” ed esprimere interrogazioni su *database (relazionali)*, perciò detti “logici” o “deduttivi”. Le origini di questo linguaggio risalgono agli albori della programmazione logica; si delineò come area di ricerca a sé un po’ più tardi, intorno al 1977, quando fu organizzato in Francia un *workshop* su logica e basi di dati: fu proprio in quell’occasione che Reiter e Clark presentarono le loro proposte circa il problema della negazione. Il nome *Datalog* fu poi coniato verso la metà degli anni ’80 da un gruppo di ricercatori nel campo della teoria dei database; da allora questo linguaggio ha riscosso un notevole interesse da parte della comunità scientifica, ma non ha avuto altrettanto successo nei sistemi di gestione di database commerciali.

Ovviamente, qui è impossibile trattare questo tema, per cui ci limitiamo a delinearne soltanto i principî di fondo, allo scopo di poter ancora illustrare un paio di istruttivi esempi. Il linguaggio Datalog non permette l’uso di simboli di funzione (*funtori*) per costruire termini (composti), importante possibilità del PROLOG puro, che abbiamo qui trascurato; pertanto, in Datalog, sono ammesse soltanto costanti e variabili quali argomenti di predicati, proprio come in tutti gli esempi finora illustrati. Come nelle usuali realizzazioni del PROLOG, sono disponibili funzioni aritmetiche predefinite e operazioni di confronto; a differenza di quelle, tuttavia, qui l’ordine sia delle clausole sia degli atomi (anche sotto negazione) nei corpi delle regole è irrilevante, grazie a un diverso approccio alla valutazione di quei predicati (detti *intensionali*) definiti appunto per mezzo di regole.

Dunque, i predicati, che corrispondono alle *relazioni* dei database, si suddividono in:

*i) predicati estensionali*, definiti esclusivamente da fatti privi di variabili (cioè dalla loro *estensione*): corrispondono alle tavole “permanenti” o “di base”, che risiedono stabilmente in memoria (e sono soggette a modifiche, magari frequenti);

*ii) predicati intensionali*, specificati soltanto attraverso regole logiche (che ne pongono il significato in modo essenziale, secondo l’*intento* del programmatore): corrispondono alle tavole “derivate” o “calcolate” o “virtuali”, le cosiddette “viste”, che saranno “materializzate” se e quando occorrerà.

Infine, bisogna rispettare alcune “condizioni di sicurezza” (*safety conditions*): sostanzialmente, si tratta di vincoli sulla scrittura delle regole che limitano il potere espressivo del linguaggio, allo scopo di preservare l’*integrità semantica* della base di dati e di ottenere qualche forma di completezza, oltre che di correttezza. Possono essere così sintetizzate:

*i) i (simboli di) predicati estensionali possono comparire, oltre che nei fatti, soltanto nei corpi delle regole;*

*ii)* ciascuna variabile che occorre in (un qualsiasi punto di) una regola deve occorrere in almeno un atomo, *non* sotto negazione e *non* “aritmetico” (operazioni di confronto incluse), che si trova nel *corpo* della regola stessa (la quale regola, in tal caso, si dice *safe*);

*iii)* non ci deve essere alcun ciclo di dipendenze tra predicati che coinvolga una negazione (in tal caso, il programma non presenta “negazioni ricorsive” e si dice *stratificato*).

La prima condizione serve ad escludere ogni tentativo di ridefinire le relazioni “di base”; le altre sono comunque essenziali per una corretta terminazione del calcolo dei predicati intensionali.

Senza scendere in dettaglio, bisogna però capire come, in generale, il meccanismo di applicazione delle regole non sia lo stesso del PROLOG: qui si devono “costruire tabelle”, ossia, per ogni regola, trovare *tutte* le *n-uple* di costanti che ne rendono vera la testa. La valutazione dei predicati intensionali parte dai fatti e procede “in avanti”, secondo un approccio *forward chaining* guidato dai dati, lo stesso che ha ispirato i cosiddetti sistemi “esperti” o “basati sulla conoscenza”. Vediamo un metodo *naïf* che ci permetta di comprendere l’idea; esistono tuttavia algoritmi assai più efficienti.

*i)* Dati i predicati estensionali, espressi mediante i fatti che costituiscono le tabelle di base, si parte con tutte le tabelle virtuali vuote.

*ii)* Si istanziano (con costanti) le variabili di tutte le regole, in tutti i modi possibili; se il corpo è vero (cioè se tutti gli atomi che vi compaiono sotto negazione sono falsi – ossia non si trovano nel *database* – e tutti gli altri sono veri), allora si deduce che la testa è vera e quindi questa va aggiunta al *database*, nella appropriata tabella virtuale.

*iii)* Si ripete ciclicamente il passo precedente, finendo quando nessun nuovo “fatto” (relativo a un predicato intensionale, e dunque corrispondente a una riga di una tabella virtuale) può essere dedotto.<sup>11</sup>

Il limite di questa valutazione costituisce il *minimo punto fisso del programma*, certamente raggiunto dopo un numero finito di passi se le condizioni di sicurezza sono rispettate (e visto che le estensioni sono tutte finite e non vi sono termini composti con funtori tra gli argomenti dei predicati). In assenza di negazioni, è uguale al minimo modello di Herbrand del programma;<sup>12</sup> ammettendo però la presenza di negazioni di atomi nei corpi delle regole, non è detto che vi sia un unico modello minimo.

---

<sup>11</sup> In alternativa, per ciascuna regola, invece di considerare tutte le possibili sostituzioni con termini costanti delle variabili che vi occorrono, si possono considerare soltanto tutte le possibili assegnazioni di *n-uple* (dalle rispettive tabelle) agli atomi nel corpo della regola che non siano sotto negazione né “aritmetici” (operazioni di confronto incluse).

<sup>12</sup> Vale la pena osservare che, nel caso particolare del programma P5, procedendo in questo modo si ottiene proprio il risultato della CWA!

Tipicamente, una base di dati è soggetta a frequenti mutamenti: infatti, sulle tabelle di base possono essere eseguite operazioni di inserimento o di modifica o di cancellazione di “righe” (*n-uple*); dunque, al momento dell’uso, le tabelle “virtuali” dovranno essere ricalcolate o aggiornate in modo appropriato; l’aggiornamento è un compito non facile, al quale in questa sede neppure accenniamo.

È chiaro a questo punto il motivo della seconda delle condizioni di sicurezza sopra elencate, da ricercare nella presenza di operatori di confronto aritmetico, ma pure della negazione stessa: ad esempio, se fosse data la regola  $p(X) :- q(Y), X > 0.$ , allora un’infinità di istanze per la variabile  $X$  potrebbe rendere vero l’atomo  $X > 0$ , anche se  $q$  è una relazione finita; analogamente nei casi di regole quali  $p(X) :- q(Y), X > Y$ . oppure  $p(X) :- \text{not}(q(Y)) ..$

Come al solito, anche nello scrivere regole *safe*, occorre sempre fare attenzione al significato che esse rivestono, specialmente quando compare qualche negazione nel loro corpo, pure in assenza di ricorsione (diretta o indiretta). A maggior ragione, l’interazione tra negazione e ricorsione crea problemi circa il significato di un programma: il risultato di una valutazione può dipendere infatti dal procedimento seguito e non sempre è intuitivo... Ed è questo il motivo della terza condizione di sicurezza, sulla quale sorvoliamo: la “stratificazione” del programma, che tuttavia non assicura l’unicità del modello minimo.

Trovare corrispondenze di fatti con i corpi delle regole permette di inferire (per *modus ponens*) nuove asserzioni, che vanno ad aggiungersi alla base di conoscenza; ad ogni passo, si generano dunque nuove conclusioni, che possono a loro volta dare luogo a nuove inferenze. Lo stesso accade quando al database è aggiunto un nuovo fatto, e si vogliono generare le sue conseguenze; ma quando un fatto decade, anche se non vi sono negazioni di sorta, la base di conoscenza può contrarsi... L’approccio *forward* è giusto, in generale, quando a priori ci si può attendere un grande numero di risposte o non si hanno conoscenze sufficientemente precise sulle forme di tali risposte; ci siamo già imbattuti in situazioni del genere: le configurazioni successive di un automa, le uscite di un sistema di produzioni, l’analisi di un gioco.

Al contrario, l’approccio *backward* guidato dall’obiettivo, attuato dal PROLOG, è ragionevole quando si sa, almeno con una buona approssimazione, che cosa si sta cercando di provare ed è atteso un numero limitato di risposte: l’identificazione di un sistema o certe applicazioni in campo diagnostico ne sono esempi. Il corrispettivo logico è il *modus tollens*: negando il conseguente, si giunge a negare l’antecedente.

Passiamo ad illustrare un paio di casi di database deduttivi, semplici ma significativi, continuando ad usare, per comodità, la sintassi del PROLOG che ci è fin qui servita. Supponiamo di avere due predicati estensionali: *persona*, di arità 1, e *sposataCon*, di arità 2; ad esempio, si possono avere i fatti *persona(beppe)* e *persona(lea)* nella tabella di base *persona*; *sposataCon(lea, beppe)* nella tabella di base *sposataCon*. Vogliamo definire (mediante regole) un predicato intensionale *nonConiugata*, di arità 1.

Tentiamo di esprimere la seguente definizione: “una persona  $X$  è non coniugata se e soltanto se non esiste una persona  $Y$  tale che  $X$  è sposata con  $Y$ ”. Si potrebbe scrivere:

```
nonConiugata(X) :- persona(X), not(sposataCon(X,Y)).
```

ma questa regola non è né *safe* (la variabile  $Y$  compare soltanto nell’atomo sotto negazione), né corretta: supponendo, ad esempio, che aldo, lea e beppe siano persone e che  $sposataCon(lea, beppe)$  sia vero, le istanziazioni  $X = lea$  e  $Y = aldo$  rendono vero il corpo di questa regola, permettendoci, erroneamente, di dedurre  $nonConiugata(lea)$ : abbiamo dimenticato di asserire che  $Y$  deve essere una persona, ma ciò non basta per ottenere la correttezza. Quella che segue è una versione “sicura” della regola, ma ancora non corretta, che giustifica in modo più chiaro la suddetta interpretazione:

```
nonConiugata(X) :- persona(X), persona(Y), not(sposataCon(X,Y)).
```

Infine, diamo una versione corretta, oltre che *safe*, che usa anche un altro predicato intensionale, *coniugata*, di arità 1:

```
coniugata(X) :- sposataCon(X,Y).  
coniugata(X) :- sposataCon(Y,X).  
nonConiugata(X) :- persona(X), not(coniugata(X)).
```

Si noti che, in Datalog, non è permessa la seguente regola, che esprime la simmetria del predicato *sposataCon*

```
sposataCon(X,Y) :- sposataCon(Y,X).
```

in sostituzione di una delle due regole relative al predicato *coniugata*, in quanto *sposataCon* è un predicato *estensionale*, che non può comparire nella testa di una regola. In effetti, questo esempio tocca un altro problema, ulteriore motivo di inaccettabilità delle prime due versioni di *nonConiugata* qualora *sposataCon* sia asserito su una coppia di costanti ma non sulla simmetrica. Tuttavia, quando si progetta un database, di solito si fa attenzione a evitare ridondanze nei dati esplicitamente memorizzati: nel nostro caso, la regola che esprime la simmetria di *sposataCon* farebbe risparmiare mezza tabella...

Per concludere, illustro un database deduttivo ispirato da un classico esempio fatto da Jeffrey D. Ullman, autore di testi fondamentali su diversi temi dell’informatica e ora professore emerito della Stanford University: il *database dei bevitori di birra* (*Principles of database and knowledge-base systems*, Volume I, Computer Science Press, 1988). Le tabelle di base (ovvero i predicati estensionali) sono sei: tre per gli insiemi di entità (i bevitori, i bar e le birre) e tre per le associazioni fra ciascuna coppia di tali insiemi (quali bevitori frequentano quali bar, quali bar servono quali birre, e quali birre piacciono a quali bevitori). Tali associazioni sono indipendenti l’una dall’altra, infatti convogliano informazioni diverse.

Avremo dunque sei predicati estensionali: tre di arit  1 (bevitore, bar, birra) e tre di arit  2 (frequenta, serve, piaceA). Esempi di fatti che li definiscono sono:

```
bevitore(ugo).
bar(sport).
birra(orzetti).
frequenta(ugo, sport).
serve(sport, orzetti).
piaceA(orzetti, ugo).
```

I fatti che definiscono frequenta citeranno tutte e sole le coppie (bevitore, bar da lui frequentato), che di solito non sono tutte le coppie (bevitore, bar), e analogamente per serve e piaceA.   Invece obbligatorio che, se un bevitore   citato in un fatto riguardante frequenta e/o piaceA, allora deve figurare anche in un fatto relativo a bevitore (ossia, deve comparire nell'elenco di tutti i bevitori), ma non   obbligatorio il viceversa, e analogamente per i bar e le birre: nei database relazionali, questi sono i vincoli di *integrit  referenziale*.

Un esempio di (utile) predicato intensionale serve a determinare le coppie (bar, birra che piace ad almeno un bevitore che lo frequenta):

```
puoSentiRsChiedere(Bar,Birra) :- piaceA(Birra,Bevitore),
                                         frequenta(Bevitore,Bar).
```

Ora potremmo costruire la differenza tra gli insiemi di coppie (bar, birra) definiti dal predicato estensionale serve e dal predicato intensionale puoSentiRsChiedere, e anche viceversa, ottenendo interessanti informazioni; infatti, se scriviamo:

```
puoNonRifornisiDi(Bar,Birra) :-  
    serve(Bar,Birra),  
    not(puoSentirsiChiedere(Bar,Birra)).  
  
dovrebbeRifornisiDi(Bar,Birra) :-  
    puoSentiRsChiedere(Bar,Birra),  
    not(serve(Bar,Birra)).
```

definiamo altri due predicati intensionali, il cui significato dovrebbe essere chiaro: il primo determina le coppie (bar, birra servita in quel bar ma che non piace a nessuno dei bevitori che lo frequentano), mentre il secondo individua le coppie (bar, birra non servita in quel bar ma che piace a qualcuno dei bevitori che lo frequentano), ed entrambi gli insiemi di coppie costituiscono informazione utile ai gestori dei bar!

Tutto funziona bene anche in PROLOG, poich , con la regola di selezione *leftmost*, quando l'atomo sotto negazione   valutato, le sue variabili sono gi  istanziate.

All'intersezione dei due insiemi caratterizzati da serve e puoSentiRsChiedere appartengono le coppie (bar, birra servita in quel bar e che piace a qualche bevitore che lo frequenta); nessuna di queste coppie appartiene a puoNonRifornisiDi, n  a dovrebbeRifornisiDi. Tale intersezione   caratterizzata dal predicato:

```

serveAQualcheAvventore(Bar,Birra) :-  

    serve(Bar,Birra),  

    puoSensitarsiChiedere(Bar,Birra).

```

Provate a definire altri predicati intensionali per determinare:

1. i bar che servono almeno una birra che piace a Ugo;
2. le birre ciascuna delle quali è servita in almeno uno dei bar frequentati da Ugo;
3. le coppie (bevitore, bar che serve almeno una birra che a lui piace);
4. chi frequenta almeno un bar che serve qualche birra che gli piace;
5. chi non frequenta alcun bar che serve qualche birra che gli piace;
6. a chi piace almeno una delle birre servite in qualche bar;
7. a chi non piace nessuna delle birre servite in qualche bar;
8. chi frequenta almeno un bar che non serve alcuna birra che gli piace;
9. chi frequenta esclusivamente bar che servono qualche birra che gli piace;
10. chi non frequenta almeno uno dei bar che servono qualche birra che gli piace;
11. chi frequenta tutti i bar che servono qualche birra che gli piace;
12. chi frequenta esclusivamente bar che servono soltanto birre che a lui piacciono.

Come per i precedenti, non c'è bisogno della ricorsione!

## Soluzioni.

1. `serveBirraChePiaceAUgo(Bar) :-  
 serve(Bar,Birra), piaceA(Birra,ugo).`

Se a Ugo non piace alcuna birra, il risultato è vuoto.

2. `servitaInUnBarFrequentatoDaUgo(Birra) :-  
 serve(Bar,Birra), frequenta(ugo,Bar).`

Se Ugo non frequenta alcun bar, il risultato è vuoto.

3. `puoEssereSoddisfattoDa(Bevitore,Bar) :-  
 serve(Bar,Birra), piaceA(Birra,Bevitore).`

Si confronti con la risposta 1.

4. `bevitoreSaggio(Bevitore) :- frequenta(Bevitore,Bar),  
 puoEssereSoddisfattoDa(Bevitore,Bar).`

Non è l'unica soluzione, ma avendo già la risposta 3. è la più immediata.

5. `bevitoreSciocco(Bevitore) :- bevitore(Bevitore),  
 not(bevitoreSaggio(Bevitore)).`

Definisce l'insieme complementare del precedente rispetto all'universo dei bevitori; si noti l'importanza fondamentale del primo atomo nel corpo della regola.

6. `puoEssereSoddisfatto(Bevitore) :-  
          puoEssereSoddisfattoDa(Bevitore,_) .`

È semplicemente la proiezione del primo argomento della risposta 3.

Si noti l'uso della *variabile anonima* `_` (al suo posto può stare una variabile fresca, qui diversa da `Bevitore`, non legata a nessun'altra sua occorrenza).

7. `insoddisfattibile(Bevitore) :- bevitore(Bevitore),  
                              not(puoEssereSoddisfatto(Bevitore)) .`

Definisce l'insieme complementare del precedente rispetto all'universo dei bevitori...

8. `nonSelettivo(Bevitore) :- frequenta(Bevitore,Bar),  
                              not(puoEssereSoddisfattoDa(Bevitore,Bar)) .`

9. `selettivo(Bevitore) :-  
                              bevitore(Bevitore), not(nonSelettivo(Bevitore)) .`

Definisce l'insieme complementare del precedente rispetto all'universo dei bevitori... Si noti che gli eventuali bevitori che non frequentano alcun bar risultano correttamente “selettivi”: infatti, per non essere tali, dovrebbero frequentare almeno un bar che non serva birre che a loro piacciono; se si vogliono escludere tuttavia, basta sostituire il primo atomo nel corpo della regola con `frequenta(Bevitore,_)` ... E se invece si sostituisse con `puoEssereSoddisfatto(Bevitore)` quale richiesta sarebbe esaudita?

10. `bevitorePigro(Bevitore) :-  
                              puoEssereSoddisfattoDa(Bevitore,Bar),  
                              not(frequenta(Bevitore,Bar)) .`

11. `bevitoreIngordo(Bevitore) :- bevitore(Bevitore),  
                              not(bevitorePigro(Bevitore)) .`

Definisce l'insieme complementare del precedente rispetto all'universo dei bevitori...

12. Bisogna dapprima definire un altro predicato:

`tollerAAltreBirre(Bevitore) :- frequenta(Bevitore,Bar),  
                              serve(Bar,Birra), not(piaceA(Birra,Bevitore)) .`

dopodiché la richiesta è soddisfatta dal predicato:

`nonTollerAAltreBirre(Bevitore) :- bevitore(Bevitore),  
                              not(tollerAAltreBirre(Bevitore)) .`

Giunti così al termine di questo impegnativo paragrafo, passiamo a un altro quesito, ben più leggero, che riguarda ancora le basi di dati relazionali...

## 14. Il database dell'agenzia di viaggi.

Vi presento – qui soltanto un poco modificato – un quesito proposto dal gruppo Bebras del Giappone nel 2015.

L'Agenzia di Viaggi dei Castori propone ai propri clienti le dieci offerte illustrate in tabella.

Tipo di viaggio	Paese di destinazione	Tipo di alloggio	Mezzo di trasporto	Vitto incluso
Viaggio di affari	Spagna	Albergo	Aereo	Sì
Fine settimana	Canada	Bungalow	Corriera	Sì
Esplorazione	Malesia	Albergo	Corriera	Sì
Luna di miele	Sud Africa	Bungalow	Aereo	No
Viaggio di affari	Francia	Albergo	Aereo	No
Viaggio di affari	Spagna	Appartamento	Aereo	Sì
Esplorazione	Malesia	Albergo	Corriera	No
Luna di miele	Sud Africa	Albergo	Corriera	Sì
Fine settimana	Canada	Appartamento	Aereo	No
Fine settimana	Francia	Albergo	Corriera	No

Come vedete, a ciascuna offerta corrisponde una riga della tabella, costituita da una “quintupla” di valori: il tipo di viaggio, il paese di destinazione, il tipo di alloggio, il mezzo di trasporto utilizzato e l'inclusione o meno del vitto.

a) Riuscite a trovare un insieme costituito dal minor numero di “voci”, i cui valori permettano di individuare non più di una riga della tabella? (Ad esempio, l'insieme costituito da tipo di viaggio e tipo di alloggio non va bene, perché due righe diverse, la prima e la quinta, hanno entrambe i valori *Viaggio di affari* e *Albergo*.) In tal modo, ogni volta che arriverà un cliente, l'impiegato dell'agenzia, Gim Castoro, potrà sapere col minor numero di domande qual è (se c'è) l'*unica* offerta in grado di soddisfare quel cliente.

b) Si può individuare una coppia di colonne tale che ciascuno dei valori presenti nella prima determini un solo valore nella seconda? (Ad esempio, tipo di viaggio e paese di destinazione non vanno bene, perché a *Viaggio di affari* corrispondono due possibili paesi: Spagna e Francia; né va bene la coppia rovesciata, cioè paese di destinazione e tipo di viaggio, perché a *Francia* corrispondono *Viaggio di affari* e *Fine settimana*.)

**Soluzioni.** a) Vi è un unico insieme di tre “voci” (o *attributi*, per usare un termine più appropriato) che soddisfa la richiesta: tipo di viaggio, tipo di alloggio e inclusione o meno del vitto.

Non vi sono insiemi di due attributi i cui valori permettono di individuare, al più, una riga della tabella, ma ce n’è uno di quattro attributi che non include banalmente tutti e tre quelli succitati: paese, tipo di alloggio, mezzo di trasporto e inclusione o meno del vitto.

Secondo la teoria delle basi di dati, *stando al contenuto attuale della tabella*, questi due insiemi di attributi:

- tipo di viaggio, tipo di alloggio e inclusione o meno del vitto;
- paese, tipo di alloggio, mezzo di trasporto e inclusione o meno del vitto

costituiscono perciò due diverse *chiavi* (*candidate*).

Una chiave è un insieme di attributi tale che, quando si fissa un valore per ciascuno di essi, al più una riga della tabella presenta proprio quei valori per quegli attributi; e ciò non è più vero, in generale, se si toglie anche un solo attributo da tale insieme. Si dice infatti che una chiave ha le proprietà di *univocità* e di *minimalità*.

Le chiavi sono importanti perché permettono appunto di individuare precisamente le righe di una tabella; di solito sono preferibili le chiavi formate da pochi attributi (nel caso in esame, probabilmente sceglieremmo la prima).

Ma supponiamo, ad esempio, di dover aggiungere una riga alla tabella dell’agenzia, contenente i seguenti valori:

Luna di miele, Spagna, Albergo, Aereo, Sì.

A questo punto, l’unica chiave possibile è costituita da tutti e cinque gli attributi!

Vorrei subito precisare che non è metodologicamente corretto dedurre a posteriori le chiavi *candidate*, tra le quali – se ve n’è più d’una – va scelta poi la chiave *primaria*. Nella fase di progettazione di una base di dati, bisogna stabilire quali sono le cosiddette *dipendenze funzionali*; intuitivamente, ci si deve chiedere: ogni volta che si fissano dei valori per gli attributi di un certo insieme, è ammesso, al più, un valore per qualche altro attributo?

Poi, sulla base di tali dipendenze, si determinano le chiavi candidate e, se è il caso, si ripartiscono i dati in più tabelle con diverse colonne (ovvero attributi)... Darò un’idea più chiara mediante un esempio, subito dopo aver risposto alla seconda richiesta.

b) No: in base ai dati contenuti nella tabella, non vi è alcuna dipendenza funzionale di un attributo da un unico altro attributo. (La verifica è semplice!)

Vediamo però di capire se può esservi dipendenza di un attributo da una *coppia* di altri attributi. Ad esempio, sarebbe legittimo chiedere a Gim Castoro se sussiste la dipendenza

$$\{ \text{Tipo di viaggio, Tipo di alloggio} \} \rightarrow \{ \text{Mezzo di trasporto} \},$$

in quanto con la coppia di valori (Viaggio di affari, Albergo) compare entrambe le volte Aereo, e con la coppia di valori (Esplorazione, Albergo) compare entrambe le volte Corriera. Se da una risposta affermativa ottenessimo conferma del nostro sospetto, allora dovremmo concludere che il progettista del database dell'agenzia ha lavorato assai male! E non è difficile scoprirne il motivo. Assumendo la chiave { Tipo di viaggio, Tipo di alloggio, Vitto incluso }, si avrebbe una *dipendenza parziale* dalla chiave: un attributo che non fa parte della chiave (Mezzo di trasporto) dipende da un sottoinsieme (proprio) della chiave, e questo fatto porta a diversi inconvenienti:

- inutili ripetizioni di valori: il mezzo di trasporto determinato da una possibile coppia di valori per il primo e il terzo attributo dovrebbe essere scritto *una sola volta*;
- impossibilità di inserire una nuova riga finché non si conosce il mezzo di trasporto;
- obbligo di ribadire un'informazione precedentemente acquisita quando si inserisce una riga con una coppia già nota di valori per il primo e il terzo attributo;
- obbligo di aggiornare tutte le righe ove occorre, qualora cambiasse il mezzo di trasporto determinato da una certa coppia di valori per il primo e il terzo attributo;
- rischio di perdita di informazione: ad esempio, se fosse cancellata l'ultima riga in tabella, ci si dimenticherebbe che per i week-end in albergo si andava in corriera...

Per la prossima stagione di vacanze, l'Agenzia di Viaggi dei Castori vuol proporre delle nuove offerte speciali. Il gestore del database predispone così un'altra tabella, chiamiamola T, sempre di cinque colonne; per comodità, indichiamo i corrispondenti cinque attributi con le prime lettere dell'alfabeto:

- A. tipo di vacanza (valori possibili: rilassante, avventurosa, culturale, ...);
- B. ambiente (mare, montagna, lago, città d'arte, ...);
- C. sistemazione (in bungalow, in albergo, in barca a vela, ...);
- D. escursioni previste (nessuna, in grotta, in mongolfiera, in un'isola, ...);
- E. presenza di una guida (nessuna, per le escursioni, per l'intera vacanza, ...).

Tuttavia, Gim Castoro ci dice che non tutte le combinazioni di valori sono previste; anzi, sussistono ben cinque dipendenze funzionali:

$$\{A\} \rightarrow \{B, D\}, \quad \{B, E\} \rightarrow \{C\}, \quad \{A, B\} \rightarrow \{C\}, \quad \{D\} \rightarrow \{E\}, \quad \{C\} \rightarrow \{A, D\}.$$

Per chiarezza: la prima significa che, fissato un valore per A, ad esso corrisponde, al più, una coppia di valori per B e D; la seconda significa che, fissata una coppia di valori per B ed E, ad essa corrisponde, al più, un valore per C, e così via.

Una domanda a questo punto si pone: chi ha pensato a metter tutto in un'unica tabella avrà fatto bene?

**Soluzione.** No, ha fatto malissimo! Potevamo sospettarlo, per via delle dipendenze piuttosto “intricate”; ma la conferma ci viene dalle chiavi candidate: come possiamo calcolarle? Nessuno dei cinque attributi è indipendente dagli altri: tutti compaiono a

destra in almeno una delle dipendenze funzionali. Dobbiamo allora partire dagli insiemi più piccoli, con un solo attributo, e controllare se qualcuno di essi determina transitivamente tutti gli altri; scopriamo così che:

- fissando A, si determinano B e D; ma A e B determinano C, mentre D determina E, e quindi tutti sono determinati;
- fissando C, si determinano A e D, e dunque nuovamente, avendo A, tutti gli altri.

Partendo invece da B o da D o da E non si determinano tutti gli altri. Vi sono allora due chiavi costituite da un solo attributo: {A} e {C}, e quindi insiemi di due o più attributi contenenti {A} o {C} non possono essere chiavi candidate, per il requisito della minimalità (sarebbero super-chiavi).

Possiamo così a controllare gli insiemi {B, D}, {B, E} e {D, E}. È facile verificare che i primi due sono chiavi candidate:

- fissando B e D, si determina E; con B ed E si determina C, e con questo A;
- fissando B ed E, si determina C, e dunque gli altri.

Non vi possono essere altre chiavi candidate: non esiste alcun insieme di tre attributi che non contenga una delle chiavi già individuate!

Considerando, ad esempio, la chiave {B, D} e la dipendenza di E da D, si vede che c'è una dipendenza parziale...

Ma supponiamo che Gim Castoro abbia detto qualcosa di troppo, e che sussistano in realtà soltanto le prime quattro dipendenze funzionali:

$$\{A\} \rightarrow \{B, D\}, \quad \{B, E\} \rightarrow \{C\}, \quad \{A, B\} \rightarrow \{C\}, \quad \{D\} \rightarrow \{E\}.$$

Possiamo così accettare la tabella T?

**Soluzione.** No, non ancora, sebbene la situazione sia cambiata in modo rilevante. Vi è infatti una sola chiave candidata, costituita dal solo attributo A, che non compare più a destra in nessuna delle dipendenze: dunque, essendo indipendente dagli altri attributi, deve appartenere a qualsiasi chiave. D'altra parte, come già constatato, esso da solo basta a determinare gli altri, e quindi costituisce una chiave; aggiungendovi anche uno solo degli altri attributi si formerebbe una super-chiave.

Dipendenze parziali dalla chiave non ve ne possono più essere, essendo la chiave di un solo attributo; purtroppo, però, continuano a presentarsi i già noti inconvenienti, a causa, ad esempio, della dipendenza di E da D, vale a dire fra due attributi che non fanno parte della chiave.

Convinciamo Gim Castoro a essere più permissivo, mantenendo soltanto le prime due dipendenze funzionali:

$$\{A\} \rightarrow \{B, D\}, \quad \{B, E\} \rightarrow \{C\}.$$

L'unica chiave è {A, E}, diversa dalle precedenti, ma si riscontra subito la presenza di dipendenze parziali (di B e D). Per porre definitivo rimedio agli inconvenienti che qui parimenti si ripropongono, dobbiamo prevedere ben tre tabelle in sostituzione di T:

- T1 con attributi (colonne) A, B, D, e chiave {A};
- T2 con B, E, C, e chiave {B, E};
- T3 con A, E, e chiave {A, E}.

La terza tabella è necessaria affinché non sia perduta informazione possibilmente contenuta nella tabella originaria T. Ricordiamo l'ordine dei cinque attributi in T:

Tipo di vacanza, Ambiente, Sistemazione, Escursioni previste, Presenza di una guida e supponiamo che in T vi fossero le tre quintuple:

- (rilassante, mare, in albergo, nessuna, per l'intera vacanza)
- (rilassante, mare, in bungalow, nessuna, nessuna)
- (avventurosa, mare, in bungalow, in un'isola, nessuna)

(si noti che le due dipendenze funzionali sono rispettate, come d'obbligo); allora in T3 vi sarebbero le tre coppie:

- (rilassante, per l'intera vacanza)
- (rilassante, nessuna)
- (avventurosa, nessuna)

senza le quali non si avrebbe modo di ricordare che in T *non* era presente la riga (avventurosa, mare, in albergo, in un'isola, per l'intera vacanza). La tabella T3 ha dunque la funzione di preservare tutta e sola l'informazione originaria.

Non resta che imporre ancora un paio di vincoli:

- se un valore dell'attributo A compare nella prima colonna di T3, esso deve anche comparire nella prima colonna di T1 (A costituisce la chiave in T1, quindi in T3 dovrà essere *chiave esterna* verso T1);
- se un valore dell'attributo E compare nella seconda colonna di T3, esso deve anche comparire nella seconda colonna di T2.

Bisogna dire che, quando si progetta una base di dati, le dipendenze tra insiemi di attributi devono emergere dall'analisi della realtà che si intende rappresentare; ma, prima ancora, è necessario individuare bene le entità, le associazioni fra queste, e le eventuali gerarchie tra entità e tra associazioni.

Seguendo le metodologie ormai consolidate, non dovrebbero sorgere problemi del genere incontrato in questo paragrafo, tranne che in casi del tutto eccezionali, come fra poco vedremo. Prima, però, servendoci di un altro esempio (ripreso da Ullman) di tabella palesemente mal costruita, cerchiamo di chiarire meglio le ultime frasi scritte qui sopra.

Abbiamo pensato di mettere in una tabella i dati che riguardano i gusti dei bevitori di birra. Gli attributi sono di nuovo cinque:

- A. nome del bevitore;
- B. indirizzo del bevitore;
- C. nome di una birra che piace al bevitore;
- D. nome del produttore di tale birra;
- E. nome della birra preferita dal bevitore.

Si escludano casi di omonimia, sia per i bevitori sia per le birre, sicché l'attributo A identifica un bevitore e l'attributo C una birra. Dovranno quindi valere le seguenti dipendenze funzionali:

$$\{A\} \rightarrow \{B, E\}, \quad \{C\} \rightarrow \{D\}.$$

Se cambia l'indirizzo di un bevitore o la sua birra preferita, devono essere aggiornate tante righe della tabella quante sono le birre che piacciono a quel bevitore. Se a un certo bevitore non piace più una certa birra, e quindi cancelliamo la corrispondente riga della tabella, può accadere che perdiamo qualche informazione, ad esempio il nome del produttore, qualora non vi siano altre righe che presentino lo stesso nome di birra in terza colonna.

La soluzione corretta prevede tre tavole:

- la prima, con i dati dei bevitori: A, B, E, e chiave {A};
- la seconda, con i dati delle birre: C, D, e chiave {C};
- la terza, con i dati dell'associazione tra bevitori e birre: A, C, e chiave {A, C}.

La terza tabella rappresenta infatti una ben precisa associazione tra bevitori e birre: un certo bevitore e una certa birra sono associati oppure no, e in caso affermativo il significato inteso è che a quel certo bevitore piace quella certa birra; inoltre, a un bevitore possono piacere più birre, e una birra può piacere a più bevitori (si dice che tale associazione è *molti-a-molti*). Si noti che l'attributo A è chiave esterna verso la prima tabella, mentre l'attributo C è chiave esterna verso la seconda tabella.

La prima e la seconda tabella rappresentano le entità dei bevitori e delle birre, con i relativi attributi. Si noti tuttavia la presenza di una chiave esterna anche nella prima tabella; l'attributo E rimanda infatti alla chiave C della seconda tabella: se, ad esempio, Orzetti è la birra preferita da Ugo, ci aspettiamo di ritrovare Orzetti in una riga della seconda tabella per sapere quale sia il suo produttore. Dunque l'attributo E rappresenta un'altra associazione tra bevitori e birre: questa volta, però, a un bevitore non è associata più di una birra, ed è questo il motivo per cui una chiave esterna è sufficiente a modellare una tale associazione. Rimane da sistemare una questione: poiché quest'ultima associazione implica la prima (se Orzetti è la birra preferita da Ugo, allora senza dubbio a Ugo piace la birra Orzetti), dobbiamo decidere se inserire d'ufficio la coppia (Ugo, Orzetti) nella terza tabella, o viceversa sottintenderla, senza inserirvela esplicitamente; in ogni caso, occorrerà fare attenzione quando si modifica la birra preferita da un certo bevitore o si vuol sapere quali birre gli piacciono...

Passiamo all'ultimo esempio, piccolo ma perfido, tratto ancora da Ullman. Negli Stati Uniti, il codice postale ( $Z = \text{zip code}$ ) determina univocamente la città ( $C = \text{city}$ ), ma per determinare il codice postale non basta la città: occorre anche la via ( $S = \text{street}$ ). Quindi devono essere rispettate due dipendenze funzionali:

$$\{S, C\} \rightarrow \{Z\}, \quad \{Z\} \rightarrow \{C\}.$$

Quali problemi sorgono?

**Soluzione.** Le chiavi candidate sono due:  $\{S, C\}$  ( $S$  e  $C$  determinano  $Z$ ) e  $\{S, Z\}$  (dove  $Z$  determina  $C$ ). Con la seconda, c'è una dipendenza parziale; scomponendo in due tabelle:

la prima, con attributi  $Z$  e  $C$ , e chiave  $\{Z\}$ ;

la seconda, con attributi  $S$  e  $Z$ , chiave  $\{S, Z\}$ , e chiave esterna  $\{Z\}$  verso la prima,

la dipendenza  $\{S, C\} \rightarrow \{Z\}$  non è preservata. Ad esempio, queste due tabelle sono corrette:

prima tabella	seconda tabella
(02138, Cambridge)	(545 Tech Sq., 02138)
(02139, Cambridge)	(545 Tech Sq., 02139)

ma, mettendo insieme i dati in esse contenuti, concluderemmo erroneamente che alla stessa coppia (545 Tech Sq., Cambridge) corrispondono diversi codici  $Z$ .

Qual è allora la soluzione? Non scomporre la tabella, assumere  $\{S, C\}$  come chiave, e imporre che a uno stesso valore di  $Z$  non possano corrispondere più valori di  $C$ ...

In Italia non abbiamo questo problema (bensì altri di enorme portata, ma il discorso esula dal nostro contesto): assumendo che  $C$  sia il comune, ciascuno dei tre attributi non dipende né da uno solo degli altri due, né dall'insieme formato dagli altri due! Inoltre, il nome del comune non è sufficiente per stabilire univocamente quale sia il comune di cui si sta parlando: ci sono sette coppie di comuni con lo stesso preciso nome (e in province diverse); ma per fortuna ciascun comune ha un codice unico (una lettera seguita da tre cifre) e dunque anche per i comuni italiani un identificatore standard esiste.

Volendo raccogliere i dati dei comuni italiani (codice del comune, nome del comune, provincia, regione), prevedere una tabella con quattro colonne non è una buona idea: la regione dipende dalla provincia, oltre che dal codice del comune; quindi, ad esempio, in tutte le righe in cui appare la provincia di Lucca, la regione non può che essere la Toscana. Dobbiamo allora scomporre i dati in due tabelle:

(provincia, regione), con chiave {provincia};

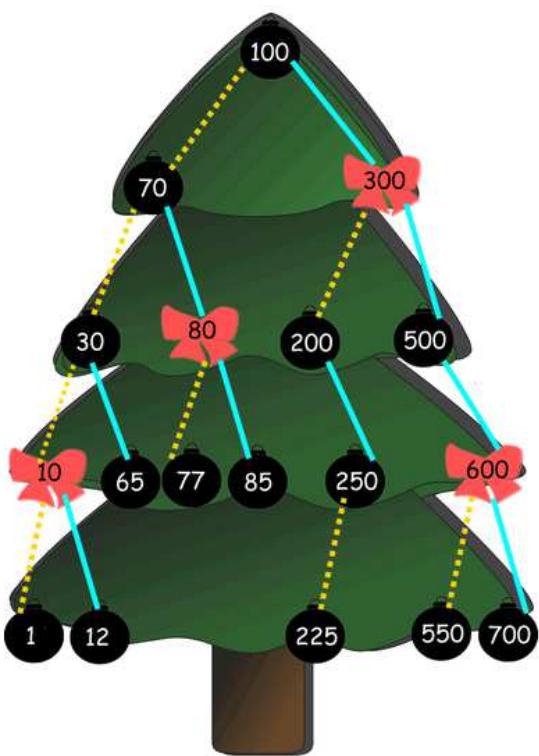
(codice del comune, nome del comune, provincia), con chiave {codice del comune} e chiave esterna {provincia} che rimanda alla tabella precedente.

E così, col pretesto del quesito sull'agenzia di viaggi, abbiamo toccato alcuni aspetti della buona progettazione di un database!

## 15. L'albero di Natale e gli alberi rosso-neri.

I sistemi di gestione delle basi di dati mantengono opportune strutture per agevolare le ricerche (anche non per chiave), sia interne (in memoria principale) sia esterne (in supporti di memoria permanente). Nel corso dei decenni sono state studiate e messe a punto tecniche sempre più sofisticate e adatte alle diverse esigenze che si possono presentare. Per mantenere ordinato un insieme di chiavi può rivelarsi utile un albero binario. Iniziamo dunque anche questo paragrafo con un quesito, proposto dal gruppo Bebras del Canada nel 2016.

Si avvicina il Natale, e Lucia decora il suo albero con fiocchi rossi e palline nere, distinguendo con un numero ciascuna decorazione.



Da ciascuna decorazione possono scendere un filo di lucine gialle a sinistra o un nastro azzurro a destra, a mo' di collegamento fino a un'altra decorazione.

Ecco qui un'immagine dell'albero di Lucia. Quest'anno la rivista “Feste eleganti” dice che un albero di Natale è *magnifico* se, *per ogni decorazione*:

- tutte le decorazioni che si trovano sotto a sinistra (seguendo il filo di lucine gialle e poi qualsiasi altro collegamento) portano un numero minore;
- tutte le decorazioni che si trovano sotto a destra (seguendo il nastro azzurro e poi qualsiasi altro collegamento) portano un numero maggiore.

Un *ramo* parte dalla decorazione in cima all'albero e giunge a una decorazione dalla quale non scendono collegamenti, né filo di lucine gialle né nastro azzurro.

Per “Feste eleganti” un albero è *vivace* se è magnifico e, inoltre, la decorazione più in alto è una pallina, ogni fiocco ha due palline immediatamente sotto di sé, e ogni ramo passa per quattro palline.

L'albero di Lucia è magnifico? È anche vivace?

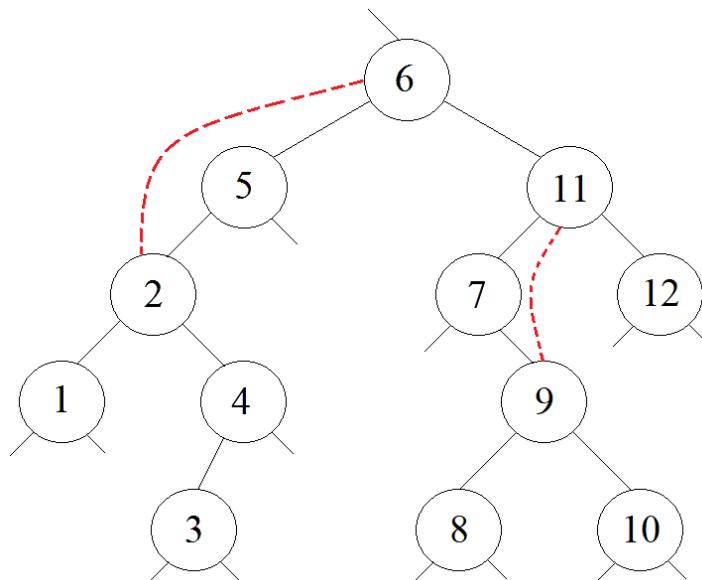
**Soluzione.** L'albero di Lucia è magnifico, ma non vivace.

È magnifico in quanto è vero che, considerando ciascuna singola decorazione, risultano soddisfatti entrambi i requisiti specificati nel testo, mentre non è vivace poiché non è rispettato l'ultimo requisito (“ogni ramo passa per quattro palline”): infatti, soltanto la metà dei rami lo soddisfa, mentre gli altri passano per tre palline.

L’albero di Natale decorato da Lucia è un... *albero binario di ricerca* (*binary search tree*, BST, in inglese); così sono dette in informatica quelle strutture ad albero binario che presentano questa caratteristica proprietà, *per ciascuno dei nodi* (fiocco o pallina che sia): tutti i nodi del sottoalbero sinistro (la parte di albero che si trova sotto il filo di lucine gialle che scende a sinistra del nodo considerato) portano un numero minore di quello scritto nel nodo considerato, mentre tutti i nodi del sottoalbero destro (la parte sotto il nastro azzurro) portano un numero maggiore. Ciò corrisponde proprio alla definizione di albero “magnifico” data nel testo del quesito.

Queste strutture servono per rappresentare insiemi di chiavi (nell’esempio, l’insieme dei numeri che etichettano le decorazioni), mantenendoli ordinati. Per stabilire se una certa chiave appartiene all’insieme, si parte dalla cima (la “radice” dell’albero, che come ben sappiamo si trova in alto!) e si confronta tale chiave con quella scritta nel nodo; se sono uguali la ricerca termina con successo, se la chiave cercata precede quella scritta nel nodo si scende a sinistra, altrimenti a destra, e così via; se non si può scendere (perché manca il filo o nastro da seguire), allora la ricerca termina con insuccesso e, nell’eventualità che la chiave cercata debba essere aggiunta all’insieme, si può scrivere in un nuovo nodo (foglia) che dovrà essere collegato proprio laddove la ricerca avrebbe dovuto proseguire.

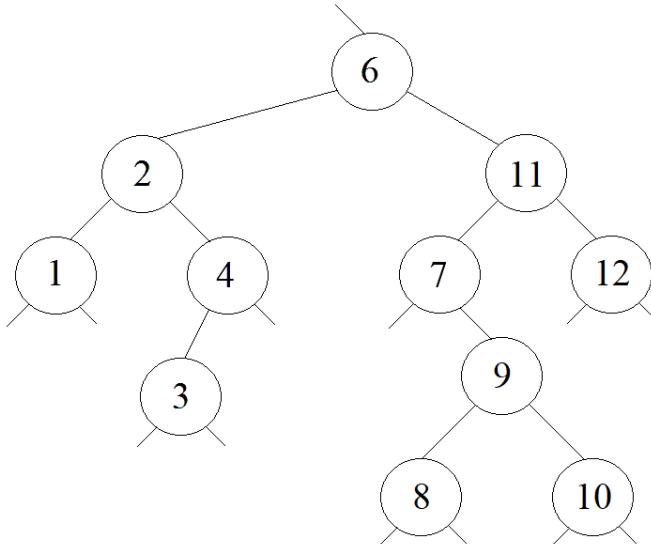
Per garantire la massima efficienza nell’operazione di ricerca, l’albero deve essere il più possibile “bilanciato” (vale a dire che possono mancare nodi soltanto sull’ultimo livello, quello alla base dell’albero); in tal caso, per una ricerca si faranno, al più,  $\text{floor}(\log_2 N) + 1$  confronti fra chiavi, se l’albero ha  $N$  nodi (cfr. p. 36). Per inciso, l’albero di Lucia non è perfettamente bilanciato, perché il penultimo livello è incompleto: vi mancano infatti due nodi.



Qui sopra è disegnato un BST; gli archi (*link*) si intendono distinti in sinistro e destro, con le frecce rivolte comunque verso il basso; sono messi in evidenza anche il *link* alla radice e i *link nulli* che denotano *sottoalberi vuoti*.

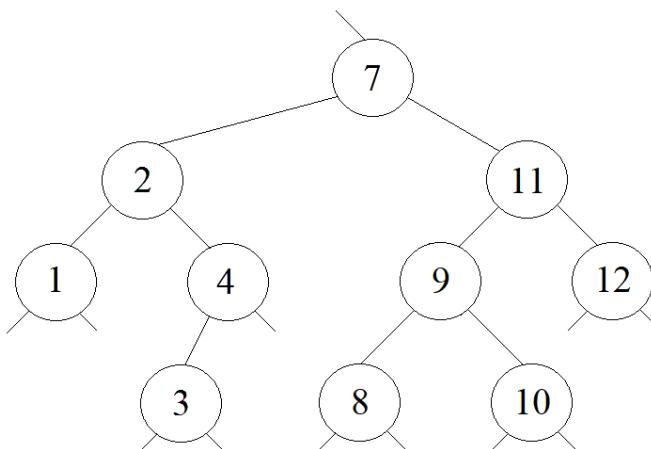
Si vede subito che anche questo albero non è perfettamente bilanciato.

Ma anche un albero perfettamente bilanciato, a causa di inserimenti e cancellazioni di chiavi, non tarderebbe a sbilanciarsi. Come può essere fatta una cancellazione? Se la chiave da cancellare (cioè da togliere dall'insieme rappresentato) è contenuta in una foglia, basta eliminare il nodo che la contiene e rendere nullo il *link* ad esso. Se la chiave da cancellare è contenuta in un nodo con *un* sottoalbero vuoto (o sinistro o destro), basta cambiare il *link* del nodo “padre” facendolo puntare all’unico nodo “figlio” di quello che contiene la chiave da cancellare: si fa insomma un *by-pass* locale, come mostrato nella precedente figura qualora si volessero cancellare le chiavi 5 e 7. Se si cancella la chiave 5, si ottiene l’albero mostrato qui sotto.



Rimane l’ultimo caso: la chiave da cancellare si trova in un nodo con entrambi i sottoalberi non vuoti; ad esempio, prendiamo la chiave 6 nella figura qui sopra. Si va a eliminare, in realtà, il nodo con chiave massima nel sottoalbero sinistro (che ha certamente vuoto almeno il suo sottoalbero destro: il nodo contenente 4, nel nostro caso), oppure il nodo con chiave minima nel sottoalbero destro (che ha certamente vuoto almeno il suo sottoalbero sinistro: il nodo contenente 7, nel nostro caso), e poi si mette la chiave che stava nel nodo eliminato al posto di quella da cancellare.

Se eliminiamo il nodo 7 e poi scriviamo 7 al posto di 6, ecco che abbiamo cancellato la chiave 6 dall’insieme, ottenendo per giunta un BST perfettamente bilanciato!



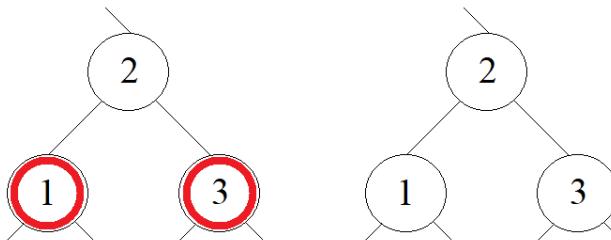
Tuttavia, nei casi più sfavorevoli, l'albero potrebbe persino ridursi a un solo ramo, e dunque la ricerca di una chiave diverrebbe sequenziale, precisamente come in una semplice lista.

Sono state ideate particolari realizzazioni dell'albero binario di ricerca, proprio allo scopo di limitarne lo sbilanciamento: una di queste è nota come “albero rosso-nero” (*red-black tree*), con nodi di questi due colori e le seguenti proprietà: 1) la radice è un nodo nero, 2) un nodo rosso non può avere alcun figlio rosso, e 3) tutti i percorsi dalla radice a un *link nullo* hanno lo stesso numero di nodi neri. Di conseguenza, il percorso più lungo possibile (con nodi neri e rossi alternati) ha un numero doppio di nodi rispetto al più corto possibile (con soli nodi neri), e proprio questo limite allo sbilanciamento permette un sensibile aumento delle prestazioni nei casi peggiori.

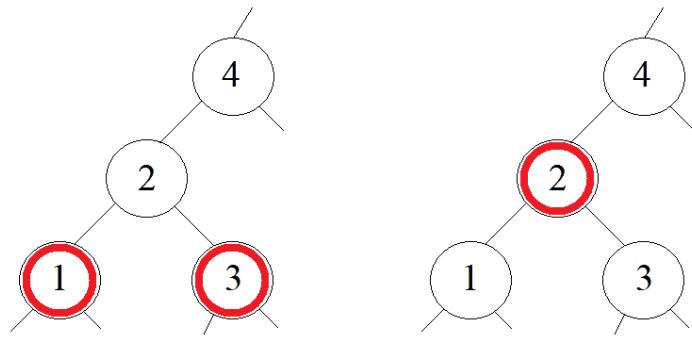
L'albero di Lucia non è affatto un albero rosso-nero! La proprietà di avere lo stesso numero di nodi neri non deve valere solo per i rami (e già questo requisito manca), ma per *tutti i percorsi dalla radice a un link nullo* (come il sinistro della pallina 200).

Già, sembra facile... ma come si fa a mantenere le suddette proprietà, ammesso che un albero le possegga, a fronte di inserimenti o cancellazioni di chiavi? Non è poi tanto difficile, e per di più si tratta di operazioni locali, attuabili all'occorrenza quando l'albero è visitato per la ricerca di una chiave, indipendentemente dall'esito di tale ricerca. Facciamo una rapida carrellata sulle situazioni che richiedono un intervento, e vediamo caso per caso in che cosa precisamente consista tale intervento; la sua giustificazione e la garanzia dell'ottenimento delle prestazioni desiderate, a seguito della sua attuazione, sono lasciate al lettore interessato, che potrà consultare allo scopo uno dei classici testi *Algorithms* di Robert Sedgewick, uno degli inventori degli alberi *red-black* (il primo volume, con algoritmi descritti in stile Pascal, fu pubblicato da Addison-Wesley nel 1983). L'autore vi spiega, in particolare, come tali alberi furono concepiti quale efficace realizzazione degli alberi di ricerca 2-3-4 *top-down* ideati da Rudolf Bayer nel 1972, e ciò permette, in effetti, di comprendere meglio le procedure ricorsive che compiono quelle trasformazioni locali atte a mantenere le proprietà desiderate dell'albero *red-black*.

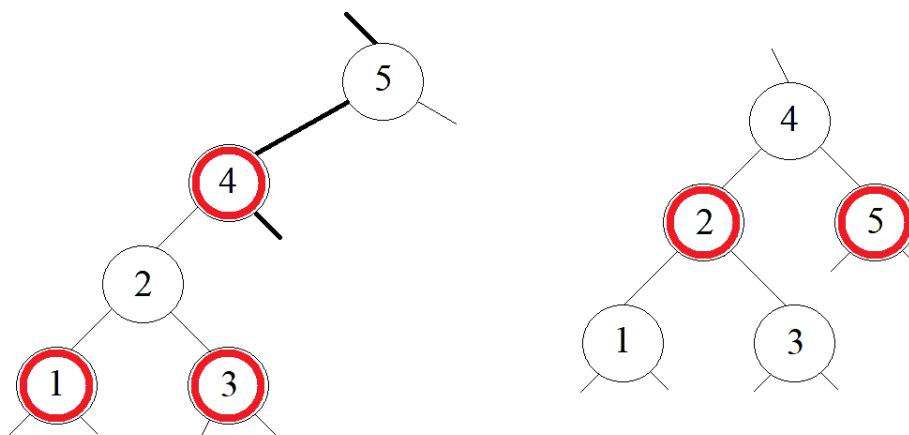
Iniziamo dal caso in cui la *radice* (che è un nodo nero) ha due figli rossi: i due figli diventano neri, e quando ciò accade aumenta di un'unità il numero di nodi neri dalla radice a un *link nullo*. (Nelle prossime figure sono mostrate mere porzioni di albero, ove i *link* che, nel disegno, non giungono ad alcun nodo di solito *non* sono nulli!)



Analogamente, quando s'incontra un nodo nero (figlio di un altro nero) con due figli rossi, i due figli diventano neri, ma il padre diventa rosso: l'operazione consiste quindi nel cambiamento di colore di tre nodi, come illustrato nella successiva figura.

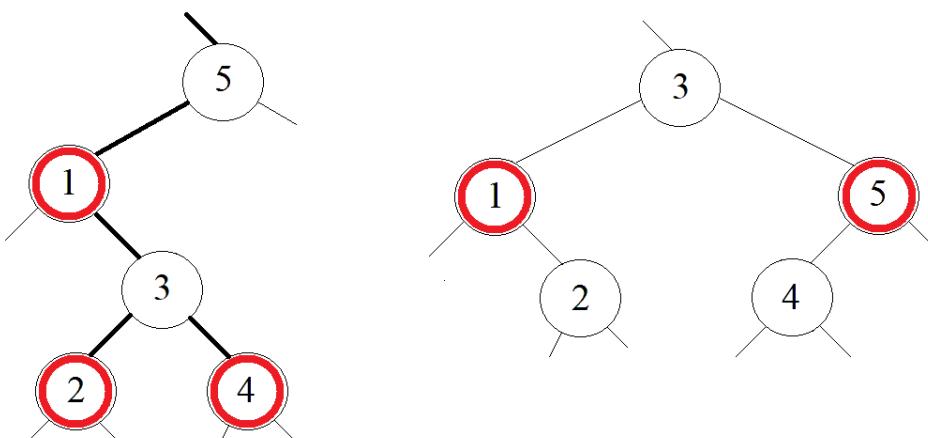


Nelle altre due situazioni, più problematiche, che si possono presentare, cambiare semplicemente il colore non basta, perché porterebbe a due nodi rossi consecutivi. Nel caso illustrato qui sotto, i due rossi consecutivi si troverebbero dalla stessa parte (in figura, lungo i *link* sinistri), e allora è sufficiente una sorta di “rotazione” per ottenere lo stato rappresentato a destra.



Cambia il colore di cinque nodi, ma devono cambiare anche tre *link* (quelli marcati in neretto nella figura a sinistra).

Infine, il caso più spinoso: facendo salire il colore rosso, questa volta i due rossi consecutivi non si trovano dalla stessa parte (in figura qui sotto a sinistra, prima si scende lungo il *link* sinistro, poi si segue il destro), e allora occorre una “doppia rotazione” che, complessivamente, comporta il cambiamento di colore di tre nodi e la modifica di cinque *link* (in neretto); nella figura a destra si vede il risultato finale.

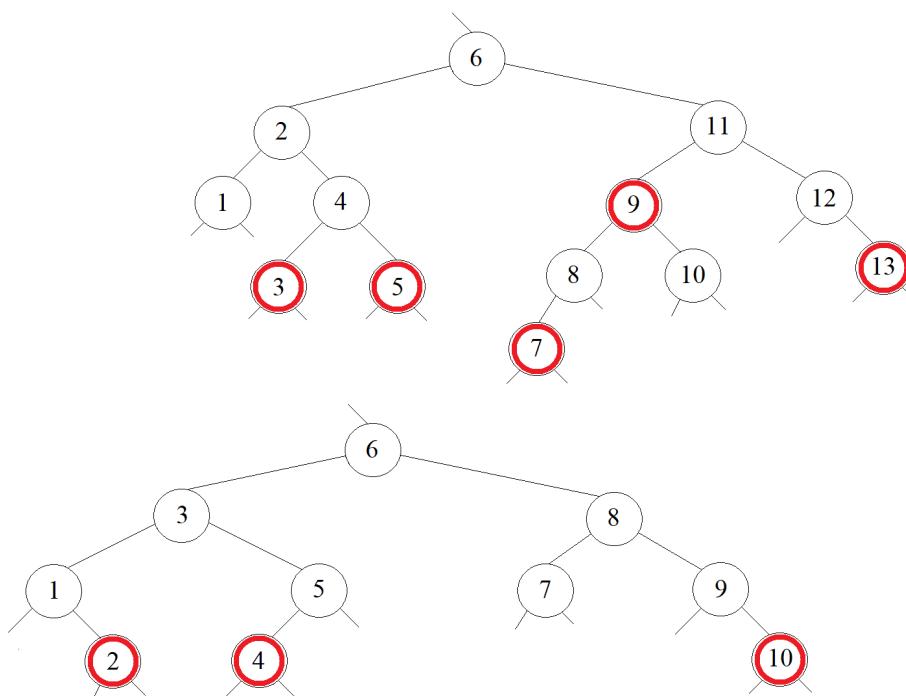


È ovvio che si possano presentare le situazioni esattamente speculari rispetto a quelle mostrate nelle tre figure della pagina precedente, da trattare alla stessa maniera.

L'operazione fondamentale su questo genere di alberi è la ricerca di una chiave; tale operazione può essere fine a sé stessa, o può preludere a un inserimento (in caso di insuccesso) o a una cancellazione (in caso di successo). Durante una ricerca, l'albero è visitato a partire dalla sua radice, e in questa fase si compiono le trasformazioni locali poc'anzi descritte (per comodità e maggior efficienza, le eventuali "rotazioni" avvengono dopo la chiamata ricorsiva dell'operazione, cioè al momento della risalita dell'albero). Se la ricerca di una chiave non ha successo e tale chiave va allora inserita, basta aggiungerla in un nuovo nodo *rosso*, da collegare al *link* (attualmente nullo) che ha decretato il fallimento della ricerca. L'operazione di cancellazione, da compiersi qualora la ricerca abbia successo, è simile ma più complicata, e pertanto rimando il lettore ai testi di Sedgewick, ove troverà anche una limpida descrizione dei B-alberi (da Bayer) e di una loro variante per la costruzione di strutture di ricerca esterna, particolarmente utili nella gestione di database di grandi dimensioni.

Comunque, l'aggravio computazionale per il mantenimento di un albero *red-black* rispetto a un semplice BST è modesto, e quanto a spazio di memoria basta un bit in più in ciascun nodo, per tener conto del suo colore. D'altro canto, le prestazioni nei casi più sfavorevoli migliorano sensibilmente: una ricerca comporta, al più,  $2 \log_2 N$  confronti fra chiavi; volendo essere precisi, non più di  $2 \text{ floor}(\log_2(N+2)) - 1$ , se  $N$  è il numero di chiavi, ossia di nodi dell'albero. Ciò significa, ad esempio, che per cercare una chiave in un miliardo di chiavi basteranno meno di 60 confronti.

Concludo proponendovi la costruzione di due alberi rosso-neri, a partire dall'albero vuoto. Nel primo caso, supponete di dover inserire le chiavi: 1, 12, 11, 2, 5, 6, 9, 4, 3, 13, 8, 10, 7, man mano che arrivano in quest'ordine; nel secondo caso: 1, 9, 3, 8, 2, 5, 6, 7, 4, 10. Seguendo le indicazioni illustrate in questo paragrafo, i due alberi finali saranno quelli disegnati qui sotto...



## 16. Fuochi d'artificio, bandiere e filastrocche.

Nel 2015 il gruppo Bebras del Canada propose un altro interessante quesito, sebbene lo stesso tema avesse già ispirato analoghi giochetti presentati nelle precedenti competizioni, persino nella primissima gara *Kangourou dell'Informatica* (“Alfabeto Morse”, marzo 2009).

Due castori, che vivono in tane situate da parti opposte del grande fiume, decidono di comunicare tramite il lancio di fuochi d'artificio di due diversi colori, azzurro e viola. Ogni messaggio è una sequenza di parole, e il vocabolario dei castori è assai limitato: comprende infatti soltanto cinque parole, così codificate:

<b>tronco</b>	azzurro	viola
<b>albero</b>	viola	azzurro
<b>roccia</b>	azzurro	viola
<b>fiume</b>	azzurro	azzurro
<b>cibo</b>	viola	

Un giorno uno di loro lancia questa sequenza di fuochi d'artificio:



azzurro    viola    azzurro    viola    azzurro    viola    azzurro    azzurro

ma, sbadatamente, dimentica di fare una pausa tra una parola e la successiva.  
Quali sono le diverse interpretazioni del messaggio visto dall'altro castoro?

**Soluzione.** I possibili messaggi sono ben quattro:

- tronco, tronco, tronco, fiume;
- tronco, roccia, cibo, fiume;
- roccia, cibo, tronco, fiume;
- roccia, albero, fiume.

Evidentemente, il codice usato dai castori richiede una pausa tra una parola e la successiva (proprio come doveva accadere tra una lettera e la successiva quando, molti anni or sono, si trasmettevano messaggi usando l'*alfabeto Morse*, ideato da Samuel Morse insieme col telegrafo elettrico), altrimenti possono sorgere parecchie ambiguità nell'interpretazione di un messaggio, come nel caso qui proposto.

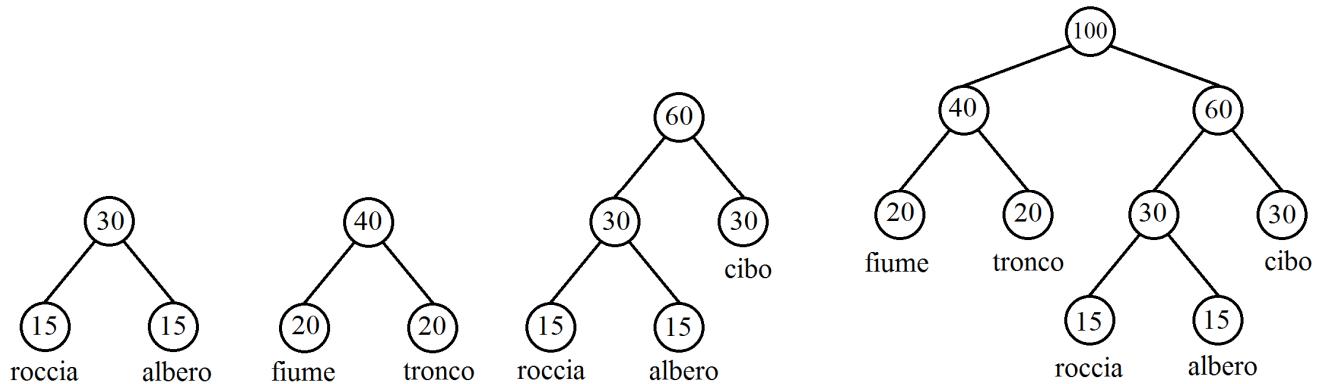
È possibile non avere mai ambiguità, pur lanciando fuochi di soli due colori senza soluzione di continuità? Certamente sì! Essendo il vocabolario costituito da sole cinque parole, basterebbe usare cinque delle otto diverse sequenze di tre fuochi. Così facendo, ciascuna parola avrebbe la stessa lunghezza delle altre... Si può sfruttare, insomma, l'usuale codifica binaria!

Questa comoda soluzione non è certo l'unica, e magari non è nemmeno la migliore. Se certe parole ricorressero più spesso di altre nei messaggi che i castori sono soliti scambiarsi, allora sarebbe più conveniente adottare un'altra tecnica, introdotta da David A. Huffman nel 1952. Supponiamo, ad esempio, che le frequenze statistiche (espresso in percentuale) delle parole del vocabolario siano le seguenti:

cibo 30, fiume 20, tronco 20, albero 15, roccia 15.

L'idea di base consiste nel codificare le parole più frequenti con le sequenze (di fuochi d'artificio) più corte, in modo tale, però, che nessuna parola sia *prefisso* di un'altra: e questa è una condizione sufficiente (sebbene non necessaria) per non creare ambiguità. In questo caso, bastano due piccole modifiche alla codifica ideata dai nostri castori, come ora vedremo...

Anche per risolvere il problema della codifica costruiamo un albero binario. Inizialmente, ad ogni parola è associato un nodo, etichettato con la frequenza relativa a tale parola. Ad ogni passo, sono raggruppati sotto un *nuovo nodo padre* i due nodi (ancora senza padre) con etichetta minore (in caso di parità, una scelta vale l'altra), uno come figlio sinistro e l'altro come figlio destro (indifferentemente), e il nuovo nodo è etichettato con la *somma* delle etichette dei due nodi figli.



Alla fine si ottiene un albero binario (come mostrato a destra); tutti gli archi sinistri si etichettano con “azzurro” e tutti i destri con “viola” (o viceversa), e ogni parola è codificata con la sequenza di colori lungo il ramo che parte dalla radice e giunge alla foglia relativa alla parola stessa. Nel nostro esempio:

Parola	Codifica	
cibo	viola	viola
fiume	azzurro	azzurro
tronco	azzurro	viola
albero	viola	azzurro
roccia	viola	azzurro

Per ottenere questa codifica (non ambigua) bastano due piccoli “aggiustamenti” a quella ideata dai castori: alla codifica di “cibo” aggiungere un secondo fuoco viola, e nella codifica di “roccia” invertire i primi due fuochi.

Da quanto detto, si intuisce che, in generale, il procedimento di Huffman non è univoco, ma porta comunque a uno dei diversi codici ugualmente “ottimi”, nel senso che statisticamente minimizzano il numero di fuochi impiegati nello scambio di messaggi.

Nel 2016 il gruppo Bebras della Repubblica Ceca presentò un quesito dove si chiedeva di ordinare quattro bandiere rispetto alla lunghezza della codifica necessaria a descriverle “per righe”, mediante una sequenza di coppie (colore, lunghezza), nota la lunghezza delle righe. Fra le quattro bandiere proposte, quella che aveva la codifica più breve era la bandiera della Germania: poiché in ogni riga è presente un solo colore, per ogni riga è sufficiente una sola coppia (colore, lunghezza)!

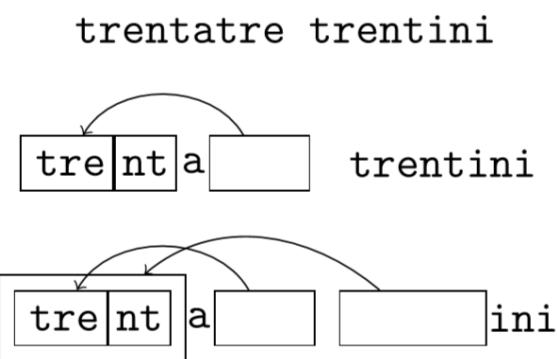


Ad esempio, se ciascuna riga consta di 90 pixel, l’intera bandiera sarà codificata dalla sequenza:  
(nero, 90), ..., (nero, 90), (rosso, 90), ..., (rosso, 90),  
(oro, 90), ..., (oro, 90),  
con un numero appropriato di coppie, che qui è lo stesso per ogni colore.

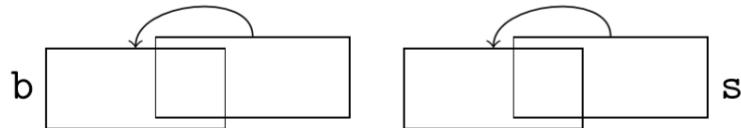
Per codificare la bandiera italiana, invece, ci vogliono tre coppie per riga.

In generale, usando questo tipo di codifica, detto *run-length*, bisogna conoscere la lunghezza delle righe per poter ricostruire esattamente l’immagine, mentre non è necessario sapere a priori il numero delle righe: quando la sequenza è terminata (e di ciò ci si deve accorgere), non ce ne saranno più! Vedremo un esempio più avanti.

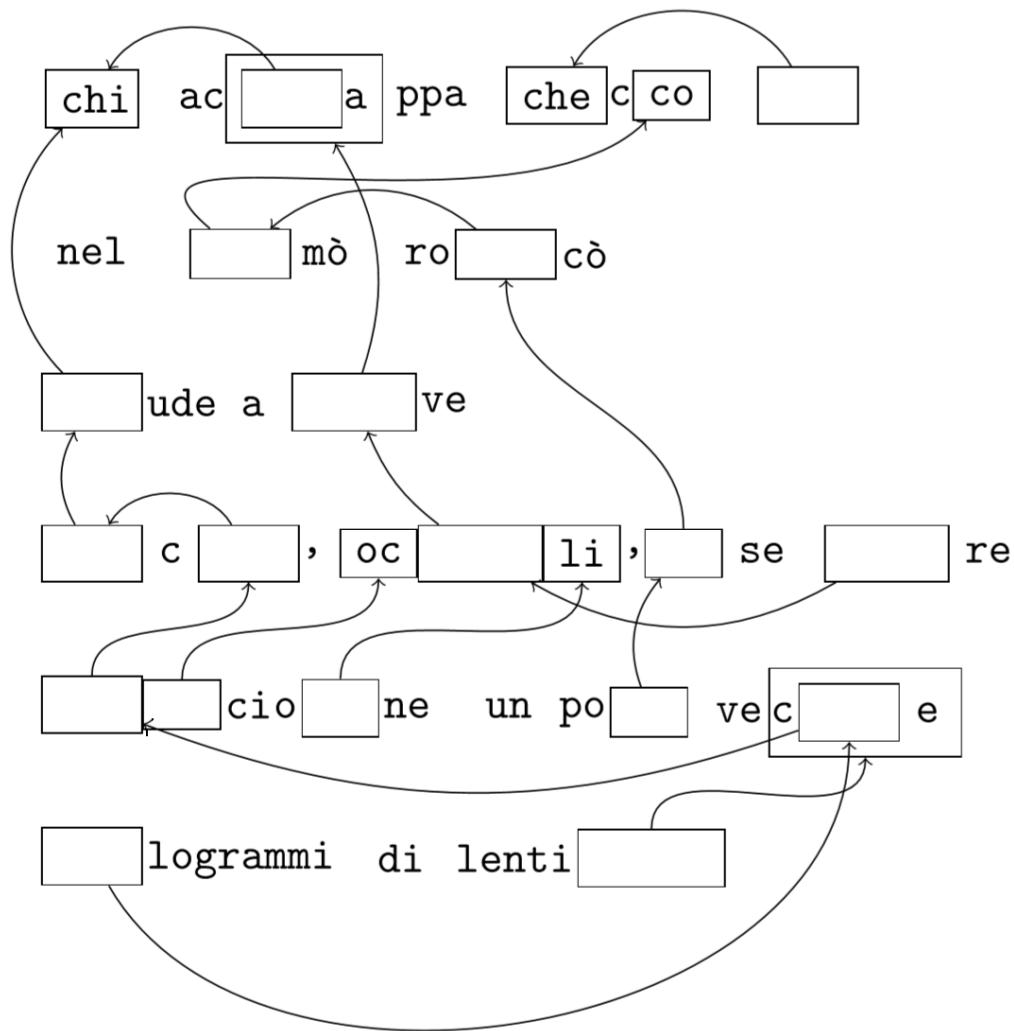
Alla gara finale *Kangourou dell’Informatica* di maggio 2011, tra i quesiti da svolgere su carta, furono presentati dei testi in cui alcune sequenze di lettere si ripetevano più volte. Ad esempio, in “trentatre trentini”, sia “tre” sia “trent” appaiono varie volte, e per giunta “tre” è contenuto in “trent”. Ogni volta che una sequenza di lettere appare di nuovo, invece di riscriverla, mettiamo una freccia che indica un punto del testo dove quella sequenza di lettere era già apparsa, come illustrato qui sotto.



Questo puzzle contiene i nomi di due frutti: sapete risolverlo?



Siete in grado di ricostruire il testo originale della filastrocca che, nella seguente figura, è stata “compressa”?



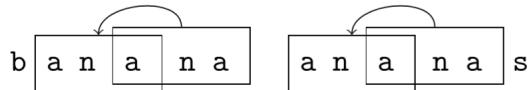
Alla pagina successiva troverete le risposte. I problemi riportati in questo paragrafo, apparentemente eterogenei, sono tuttavia accomunati dai temi della *codifica* e della *compressione* dei dati, ove può tornar utile, e in più occasioni, la tecnica di Huffman. La compressione dei dati è rilevante nelle applicazioni informatiche, avendo lo scopo di ridurre sia l’occupazione di memoria sia il tempo di trasmissione dei dati stessi, e dunque anche il traffico nelle reti. Sono stati messi a punto algoritmi efficaci, che riescono a ridurre sensibilmente le dimensioni dei file, in particolare di quelli contenenti la codifica di suoni, immagini e video, o la conversione in forma digitale di segnali analogici d’altro genere, di solito piuttosto “ingombranti”.

Tipicamente si richiede che un algoritmo di compressione fornisca una sequenza di bit più breve di quella che costituisce i dati di partenza (almeno nel caso di dati del tipo specifico per cui l'algoritmo è stato studiato, ad esempio immagini), ma dalla quale si riesca a risalire (mediante un algoritmo “inverso”, di decompressione o espansione) precisamente alla sequenza originaria. Talvolta, però, si accetta qualche perdita di informazione (ad esempio una definizione un po' più bassa di un video o una gamma di colori ridotta) pur di ottenere una maggior compattezza.

Nel prosieguo di questo paragrafo descriveremo alcuni tra i più celebri algoritmi di compressione senza perdita di informazione, iniziando da quello di Huffman, che può rivelarsi efficace per lunghi file di testo. Nel 1977 Ziv e Lempel studiarono un altro notevole metodo, una cui variante, realizzata da Storer e Szymanski nel 1982, portò al noto programma “gzip”. Nel 1978 ancora Ziv e Lempel ebbero un’idea un po’ diversa e altrettanto rimarchevole che, migliorata da Welch nel 1984, fu impiegata per la codifica di dati geofisici e di immagini nel formato GIF.

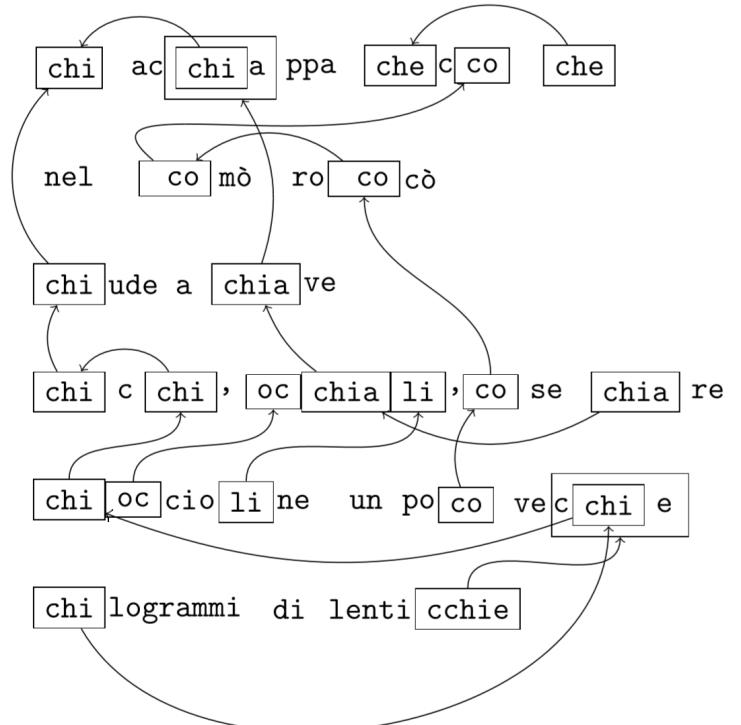
Bisogna notare che un algoritmo di compressione non può essere sempre efficace su qualsiasi tipo di dati: pensate, come caso limite, a un file di byte “casuali”. Infatti, se fosse sempre efficace, la sua applicazione potrebbe continuare *ad libitum*, giungendo a un file arbitrariamente piccolo!

Prima di procedere, diamo però le soluzioni dei quesiti esposti alla pagina precedente. I frutti sono la banana e l’ananas: e negli spazi andavano scritte le stesse lettere!



Ed ecco infine la filastrocca ricostruita:

Chi acchiappa Checco che  
nel comò rococò  
chiude a chiave  
chicchi, occhiali, cose chiare  
chioccioline un poco vecchie  
chilogrammi di lenticchie  
...



## 16.1. Codifica *run-length*.

Iniziamo dal metodo di compressione più semplice. Sia dato un file di byte; s'intende sì byte qualsiasi, ma diciamo con una buona probabilità di ripetizioni consecutive, come ad esempio nella sequenza di 38 byte:

```
aaaabbbaabbbbbcccccccdabcbaaabbbbcccd
```

Qui i byte sono indicati da lettere minuscole (e, ovviamente, a uguale lettera corrisponde uguale byte), sebbene i possibili byte siano più delle lettere (come sappiamo, sono 256). Si può allora pensare di codificare la sequenza scritta sopra con la seguente:

```
4a3baa5b8cdabcb3a4b3cd
```

Le sequenze massimali (di lunghezza  $\geq 3$ ) di uno stesso byte sono rimpiazzate dalla lunghezza seguita da una sola occorrenza di tale byte. Tuttavia, così facendo, si crea confusione: anche le lunghezze, infatti, devono essere codificate con (un) byte! C'è quindi necessità di un byte particolare di *escape*, che preceda ciascuna lunghezza... Possiamo procedere nel seguente modo: assumiamo il byte 0 (otto bit tutti zero) come *escape* e codifichiamo ciascuna lunghezza con un byte da 1 a 255 (il byte 1 può corrispondere alla lunghezza 4, che ora è la minima che fa risparmiare spazio, ..., il byte 255 può corrispondere alla lunghezza 258; eventuali sequenze più lunghe verranno suddivise...); ciascuna eventuale occorrenza del byte 0 nel file originario è codificata con *due* byte 0 consecutivi (00), sì da non avere alcuna ambiguità.

Nel caso esemplificato otteniamo:

```
04abbbaa05b08cdabcbaaa04bcccd
```

con un risparmio di 9 byte su 38.

Un caso particolare, e significativo, è costituito dai file *binari* (s'intende visti come sequenze di bit singoli), ad esempio i cosiddetti file *raster* (cioè di grafica *bitmap*); ivi le sequenze di 0 e 1 sono alternate, e quindi basta memorizzare le lunghezze! Se il file inizia con una sequenza di 1, allora la prima lunghezza sarà 0...

Se il file è suddiviso in linee di uguale lunghezza, il primo numero nella codifica rappresenta tale lunghezza; e il primo numero relativo a una linea dice quanti 0 si trovano all'inizio di tale linea.

All'inizio della pagina successiva è riportato un esempio. Le linee sono 19, ciascuna di 50 bit. Per rappresentare i numeri da 0 a 50 occorrono 6 bit. Quindi il file originario, compresa la codifica del numero 50 all'inizio, occupa 956 bit ( $6 + 19 \times 50 = 956$ ). Codificandolo invece con il numero 50 seguito dai 63 numeri riportati sopra a destra, si impiegano soltanto 384 bit ( $64 \times 6 = 384$ ): il risparmio è prossimo al 60%. Chiaramente, se la lunghezza (della generica sequenza di bit ripetuti o della linea di bit) può eccedere i 63 bit, allora 6 bit non sono sufficienti per rappresentarla ( $63 = 111111_2$ ); e naturalmente, per la corretta decodifica, bisogna sapere quanti bit sono stati usati per la codifica dei numeri.

00000000000000000000000000000000111111111111000000000	27	14	9		
0000000000000000000000000000000011111111111100000000	25	18	7		
000000000000000000000000000000001111111111111111000000	22	24	4		
0000000000000000000000000000000011111111111111111111000	21	26	3		
0000000000000000000000000000000011111111111111111111110	19	30	1		
0000000000000000000000000000000011111110000000000000000111111	18	7	18	7	
0000000000000000000000000000000011111000000000000000000011111	18	5	22	5	
000000000000000000000000000000001110000000000000000000000111	18	3	26	3	
000000000000000000000000000000001110000000000000000000000111	18	3	26	3	
000000000000000000000000000000001110000000000000000000000111	18	3	26	3	
000000000000000000000000000000001110000000000000000000000111	18	3	26	3	
000000000000000000000000000000001110000000000000000000000111	18	3	26	3	
000000000000000000000000000000001110000000000000000000000110	19	4	23	3	1
00000000000000000000000000000000111000000000000000000000011000	21	3	20	3	3
111	0	50			
111	0	50			
111	0	50			
111	0	50			
111	0	50			
1100011	0	2	46	2	

Si noti che, nel caso di file suddivisi in linee di uguale lunghezza, il primo numero nella codifica stabilisce la somma di ciascuno dei successivi gruppi di numeri da considerare per un’immediata e giusta suddivisione in linee delle sequenze di bit da essi rappresentate. (Avete capito che cosa rappresenta l’immagine qui sopra?)

In generale, per un algoritmo di compressione, possiamo considerare il dato di input come una *stringa* T (*testo*) di lunghezza  $n > 0$  (che tratteremo come un array di  $n$  elementi, con indice da 0 a  $n - 1$ ) sull’*alfabeto* A = { $a_0, \dots, a_{m-1}$ } (costituito da  $m > 1$  simboli distinti: ad esempio, nel caso dei byte  $m$  è 256, nel caso dei bit 2). In altre parole, T è una sequenza di  $n$  simboli (il primo occupa il posto 0, l’ultimo il posto  $n - 1$ ) tipicamente con ripetizioni (di solito, infatti,  $n$  è molto maggiore di  $m$ ).

Nel seguito, useremo il termine “stringa” per dire “sequenza, di lunghezza finita, di simboli dell’alfabeto considerato”.

Il risultato fornito in output sarà il *testo compattato*, o *codificato*, cioè una sequenza ancora di byte, oppure di numeri o anche – come vedremo – di dati di formati o tipi diversi, che infine sarà comunque riconducibile a una sequenza di bit possibilmente più breve di quella che costituisce T, ma dalla quale si riesca a risalire precisamente a T.

## 16.2. Codifica a lunghezza variabile.

Dell’algoritmo dovuto a David A. Huffman abbiamo già parlato nelle considerazioni a margine del quesito sui fuochi d’artificio, ma qui lo rivediamo alla luce dei suoi utili impieghi per la compressione senza perdita di informazione. Ricordiamo che esso si basa sulla tabella delle occorrenze (o delle frequenze statistiche, espresse in percentuale) dei simboli di A in T. Può essere efficace per (lunghi) file di testo.

L’idea di base consiste nel codificare i simboli più frequenti con i *pattern* di bit più corti, senza bisogno di usare poi un separatore (come invece accade, ad esempio, nel codice Morse usato una volta nelle comunicazioni via telegrafo), sebbene i *pattern* che codificano i simboli di A possano appunto avere diverse lunghezze.

Inizialmente, ad ogni simbolo di A è associato un nodo, etichettato con il numero delle relative occorrenze in T (o con la relativa frequenza in percentuale).

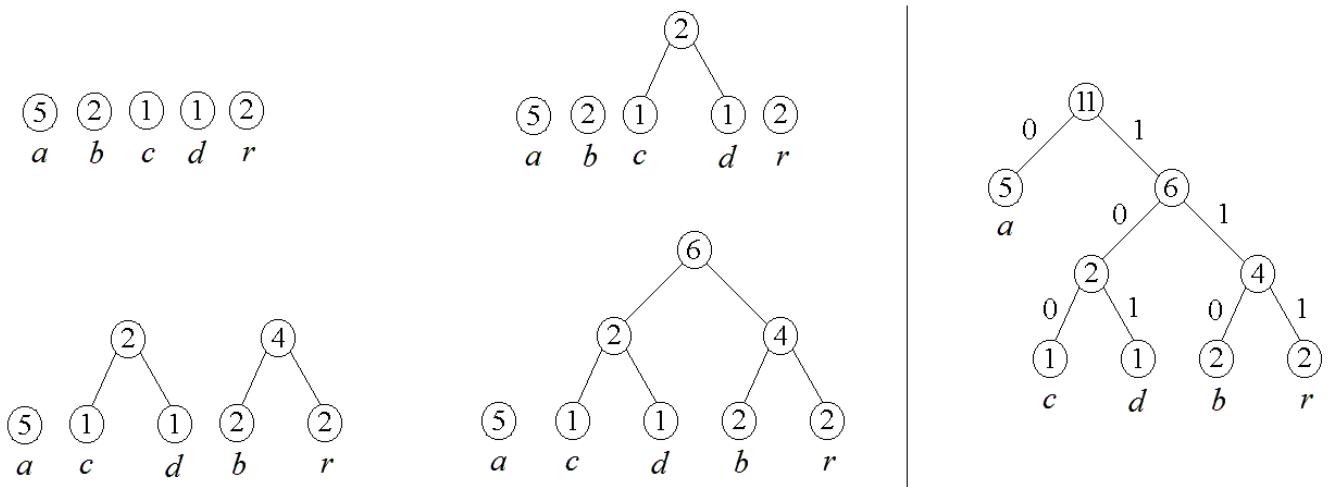
Ad ogni passo, sono raggruppati sotto un nuovo nodo padre i due nodi (ancora senza padre) con etichetta minore (in caso di parità, una scelta vale l'altra), uno come figlio sinistro e l'altro come figlio destro (indifferentemente), e il nuovo nodo è etichettato con la somma delle etichette dei due nodi figli.

Alla fine si ottiene un albero binario; tutti gli archi sinistri si etichettano con 0 e tutti i destri con 1 (o viceversa), e ogni simbolo di A è codificato con la sequenza di bit lungo il ramo che parte dalla radice e giunge alla foglia relativa al simbolo stesso.

Da quanto detto, si intuisce che il procedimento non è univoco, ma porta comunque a uno dei diversi codici ugualmente ottimi. Vediamo un esempio.

$$A = \{a, b, c, d, r\} \quad T = abracadabra \quad n = 11$$

Per rappresentare cinque simboli occorrono tre bit (due infatti non bastano, perché le loro configurazioni sono soltanto quattro): ciò porterebbe a pensare che servano 33 bit per la codifica di T. L'algoritmo di Huffman può far di meglio per questo T, non usando tre bit per ciascuno dei simboli di A.



Seguendo i passi sopra illustrati (a destra l'albero finale), si ottiene il codice

$$a = 0 \quad b = 110 \quad c = 100 \quad d = 101 \quad r = 111$$

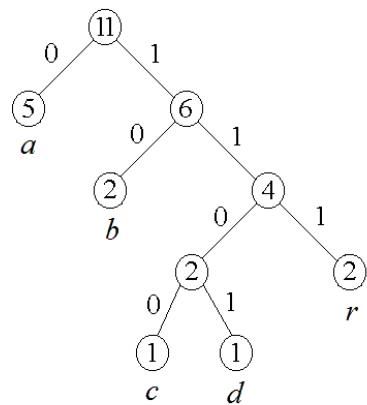
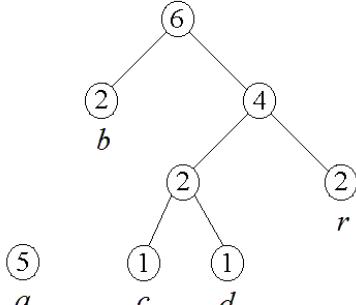
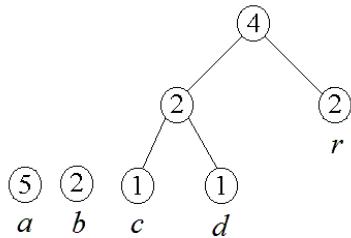
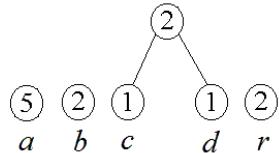
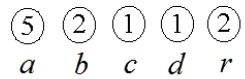
mediante il quale T sarà rappresentato da una sequenza di soli 23 bit:

$$01101110100010101101110.$$

Una delle altre possibilità è raffigurata all'inizio della pagina successiva; da essa si ottiene il codice

$$a = 0 \quad b = 10 \quad c = 1100 \quad d = 1101 \quad r = 111$$

per cui T potrà essere ancora rappresentato da una (diversa) sequenza di 23 bit.



Che cosa si può dire per altri testi, sempre sullo stesso alfabeto? I codici ottenuti sono efficaci per quei file dove i cinque simboli di A mantengono le frequenze circa proporzionali alle occorrenze in *abracadabra*. È evidente che se, ad esempio, nel file predominasse la presenza dei simboli *c* e *d* e fosse scelto il secondo codice, allora non si otterrebbe alcuna “compressione”! Tutto sommato, è preferibile scegliere il primo dei due codici sopra ricavati, il cui albero è meno “sbilanciato”...

Per quanto concerne la fase di *decompressione*, o *decodifica*, è sufficiente conoscere l’albero finale (o semplicemente il codice da esso ottenuto): si parte dalla radice e ad ogni bit in input (ovvero ricevuto) si scende lungo l’arco etichettato con quel bit; non appena si arriva a una foglia, si riconosce (e si scrive in output) il simbolo associato a tale foglia, e si riparte dalla radice.

In ogni caso, a meno che non sia “cablato” una volta per tutte nel software di codifica/decodifica (come il codice ASCII), il codice usato è un’informazione che deve accompagnare il testo compattato per la sua corretta decodifica. Per file di testo sufficientemente lunghi, il risparmio può comunque essere rilevante (il 20% o più); com’è ovvio, non è così per file “casuali”, dove statisticamente le frequenze dei simboli sono circa uguali.

Si può anche pensare all’adozione di un codice di Huffman “standard”, basato sulle frequenze dei simboli alfabetici nella lingua usata per scrivere i testi (ma, oltre alle lettere, si devono pure considerare i segni di punteggiatura, lo spazio, eccetera).

In origine, questo sistema di codifica fu sviluppato per minimizzare la quantità di informazione necessaria nei sistemi di comunicazione: quindi con l’intento di risparmiare non lo spazio ma il tempo. Abbiamo constatato che la tecnica di Huffman non ha bisogno di “pause” tra i simboli codificati; tuttavia, vale la pena di riflettere sul fatto che, in trasmissione o in ricezione, la perdita anche di un solo bit può avere effetti drammatici! (Qual è un plausibile rimedio a tale inconveniente?)

### 16.3. Algoritmo di Lempel-Ziv (1977).

Nel 1977 Jacob Ziv e Abraham Lempel pubblicarono un algoritmo di compressione che fa uso di una “finestra scorrevole” (*sliding window*) sul testo. Una versione del loro metodo, divenuta assai popolare, comprende una modifica introdotta da James Storer e Thomas Szymanski nel 1982; per semplicità, e senza esplicitare l’uso di finestre scorrevoli, noi possiamo descriverla, ad alto livello, nel seguente modo:

```
prende in input T, array di  $n$  simboli, con indice da 0 a  $n - 1$  ( $n > 0$ );  
 $wmin \leftarrow$  minima lunghezza delle sottostringhe da riconoscere come ripetute ( $> 1$ );  
crea una sequenza C (non omogenea: vedi oltre) e la inizializza con  $T[0]$ , primo simbolo di T;  
int  $w, k, i = 1$ ;  
while ( $i < n$ ) {  
    cerca la corrispondenza (match) più lunga di una sottostringa di T  
    iniziante da  $T[i]$  con una sottostringa di T iniziante precedentemente;  
    if (è stata trovata) sia lunga  $w$  ( $\geq 1$ ) e inizi da  $T[k]$ , con  $k < i$ ; else  $w = 0$ ;  
    if ( $w < wmin$ ) {  
        aggiunge il simbolo  $T[i]$  in fondo a C;  $i = i + 1$ ;  
    } else {  
        aggiunge la coppia di numeri ( $w, k$ ) in fondo a C;  $i = i + w$ ;  
    }  
}  
}
```

fornisce in output la sequenza C.

La sequenza C è non omogenea dato che vi compaiono simboli dell’alfabeto ed eventualmente coppie (lunghezza, posizione): infatti, nell’ultima istruzione **if**, a seconda del valore di verità della condizione, è aggiunto un simbolo, o altrimenti una coppia di numeri, in fondo alla sequenza C (che pertanto ne risulta modificata). Alla fine, la sequenza C contiene la codifica di Lempel-Ziv “ad alto livello” del testo originario T.

Ad esempio: sia  $T = abracadabrabracadabracadarbar$ , con  $n = 32$ , e fissiamo  $wmin = 3$  (anche 4 porta allo stesso risultato). Finché  $i \leq 6$  a C sono aggiunti simboli di T, ma quando  $i = 7$  è trovato un match di lunghezza 4 con la sottostringa che inizia dalla posizione  $k = 0$ :

*a b r a c a d a b r a b r a c a d a b r a b r a c a d a r b a r*

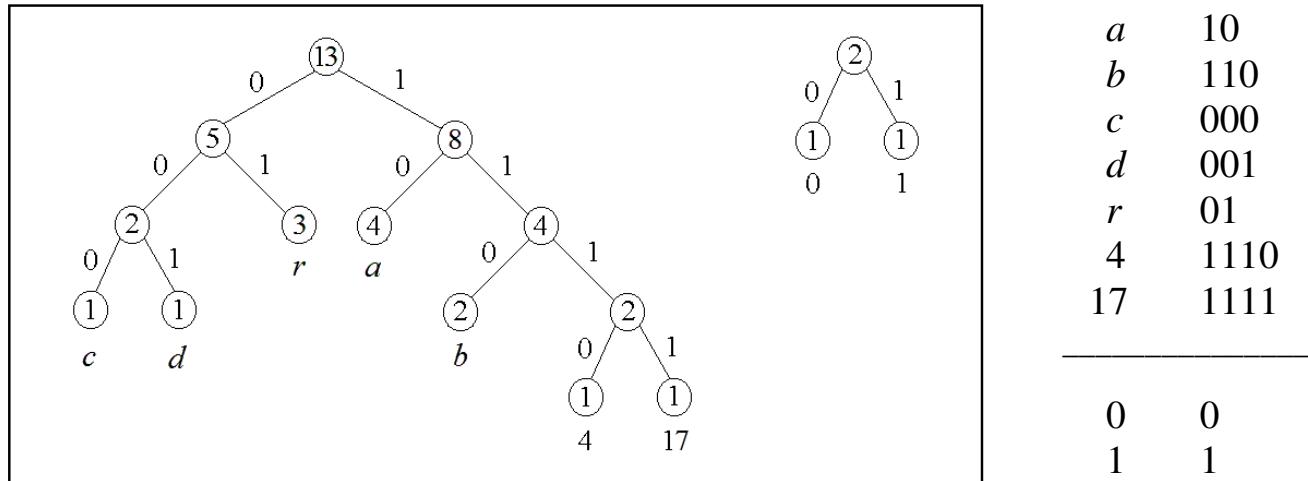
sicché a C è aggiunta la coppia (4, 0); subito dopo, con  $i = 7 + 4 = 11$ , è trovato un match di lunghezza 17 con la sottostringa che inizia dalla posizione  $k = 1$  (si noti che ora le due sottostringhe uguali si sovrappongono parzialmente, nella parte in verde):

*a b r a c a d a b r a b r a c a d a b r a b r a c a d a r b a r*

e quindi a C è aggiunta la coppia (17, 1); dopodiché  $i$  aumenta al valore  $11 + 17 = 28$ , ma negli ultimi quattro simboli non vi sono più sottostringhe lunghe 3 (anzi, neppure lunghe 2) uguali a sottostringhe precedenti. Si ottiene così l'output

$$C = a b r a c a d (4, 0) (17, 1) r b a r$$

Come si può procedere a questo punto per una buona codifica finale? L'idea che porta alla cosiddetta compressione *gzip* consiste nel ricavare *due* alberi di Huffman: uno per i simboli e le lunghezze e un altro per le posizioni. Nel caso in esame, un codice ottimo di Huffman è dato dai due alberi in figura (il secondo è evidente):



La codifica *gzip* finale è una sequenza di 36 bit soltanto:

101100110000100011110011111011101001.

Non c'è ambiguità, perché dopo una lunghezza ci si deve aspettare una posizione. Ma, come sempre, per risalire al testo originario bisogna conoscere le tabelle create e usate per la codifica (riportate qui sopra, a destra della figura)...

Questo è un caso fortunato: se avessimo codificato ciascuna delle cinque lettere con tre bit, avremmo ottenuto una sequenza di 96 bit ( $32 \times 3 = 96$ ); se avessimo applicato invece l'algoritmo di Huffman descritto precedentemente, la sequenza sarebbe stata di 69 bit, com'è facile verificare.

#### 16.4. Algoritmo di Lempel-Ziv-Welch (1984).

Nel 1978 gli stessi Jacob Ziv e Abraham Lempel presentarono un'altra ricerca sul tema “compressione”, questa volta basata su “dizionari”. Nel 1984 Terry Welch migliorò la loro idea, dando origine all'algoritmo oggi noto con l'acronimo LZW. In questa sede ci limiteremo a descrivere sia il procedimento di compressione sia quello di decompressione, senza motivarne la correttezza.

All'inizio è sufficiente introdurre in un “dizionario” i simboli dell'alfabeto seguendo un ordine prestabilito; via via che la codifica procede, vi sono aggiunte nuove voci (*entries*), e lo stesso preciso dizionario sarà poi ricostruito nella fase di decodifica, senza bisogno di averlo già pronto e memorizzato insieme col testo codificato!

L'algoritmo di *compressione* può essere descritto, ad alto livello, nel seguente modo:

prende in input T, array di  $n$  simboli, con indice da 0 a  $n - 1$  ( $n > 0$ ) ;

**int**  $v, w, i$  ;

alloca un array D (il dizionario), di lunghezza variabile (o prefissata: *cfr.* la nota finale), in cui ciascun elemento possa ospitare una stringa, e inizializza i suoi primi  $m$  elementi con i simboli dell'alfabeto A, nell'ordine stabilito, cioè:

**for** ( $v = 0$  ;  $v < m$  ;  $v++$ )  $D[v] \leftarrow a_v$  ;

$v = m$  ;

crea una sequenza C di numeri naturali, vuota ;

crea una stringa B e la inizializza con  $T[0]$ , primo simbolo di T ;

$w \leftarrow$  indice in D del simbolo  $T[0]$  ;

**for** ( $i = 1$  ;  $i < n$  ;  $i++$ ) {

aggiunge il simbolo  $T[i]$  in fondo a B ; // quindi B risulta modificata

**if** (B non si trova in D) {

$D[v] \leftarrow B$  ; // B è inserita nel primo posto libero del dizionario

$v++$  ; // e  $v$  ricorda quante voci si trovano nel dizionario

aggiunge il numero  $w$  in fondo a C ;

$B \leftarrow T[i]$  ; // rimpiazza il contenuto di B con  $T[i]$ , l'ultimo suo simbolo ;

}

$w \leftarrow$  indice in D della stringa contenuta in B ;

}

aggiunge il numero  $w$  in fondo a C ;

fornisce in output la sequenza C.

Quindi il testo compattato consiste in una sequenza di numeri naturali, ciascuno dei quali è certamente inferiore al numero finale  $v$  di voci nel dizionario. Comunque, il numero di bit necessari per la codifica è calcolato in base al più grande numero in C. Vediamo un semplice esempio.

$$A = \{a, b, c\} \quad T = bcababbcabcbcbbccaabbababb \quad n = 26$$

Se rappresentiamo i tre simboli con due bit ciascuno, la codifica di T occupa 52 bit. Applicando l'algoritmo descritto sopra, si ottiene una sequenza di 15 naturali:

$$C = 1, 2, 0, 1, 5, 3, 5, 2, 3, 1, 3, 4, 7, 5, 7.$$

Il più grande è 7, per cui bastano tre bit per ognuno di essi; dunque la codifica di T può essere fatta impiegando 45 bit, con un risparmio superiore al 13%.

Si invita il lettore ad eseguire manualmente la procedura sull'esempio ora proposto, verificando anche che, al termine, il dizionario contiene le 17 voci di seguito riportate accanto al rispettivo indice.

0	<i>a</i>	5	<i>ab</i>	9	<i>abc</i>	13	<i>bcc</i>
1	<i>b</i>	6	<i>ba</i>	10	<i>cb</i>	14	<i>caa</i>
2	<i>c</i>	7	<i>abb</i>	11	<i>bcb</i>	15	<i>abba</i>
3	<i>bc</i>	8	<i>bca</i>	12	<i>bb</i>	16	<i>aba</i>
4	<i>ca</i>						

L’algoritmo di *decompressione* opera nel seguente modo:

prende in input C, visto come array di  $l$  naturali, con indice da 0 a  $l-1$  ( $l > 0$ );

**int**  $v, i$ ;

alloca un array D (il dizionario), inizializzato come sopra, cioè:

**for** ( $v = 0$ ;  $v < m$ ;  $v++$ )  $D[v] \leftarrow a_v$ ;

$v = m$ ;

crea due stringhe, T e B1, e inizializza entrambe con  $D[C[0]]$ , vale a dire quella stringa che, nel dizionario D, si trova al posto di indice C[0];

crea una stringa B2, vuota;

**for** ( $i = 1$ ;  $i < l$ ;  $i++$ ) {

// prepara in B2 la prossima stringa da aggiungere in fondo a T:

**if** ( $C[i] < v$ )  $B2 \leftarrow D[C[i]]$ ; **else**  $B2 \leftarrow B1$  seguita dal primo simbolo di B1;

aggiunge la stringa B2 in fondo a T;

$D[v] \leftarrow B1$  seguita dal primo simbolo di B2;

$v++$ ;

$B1 \leftarrow B2$ ; // rimpiazza il contenuto di B1 con quello di B2

}

fornisce in output la stringa T.

Si invita ancora il lettore a verificare che sull’alfabeto A e sull’output C dell’esempio precedente (con  $l = 15$ ), questo algoritmo ricostruisce lo stesso dizionario D e dà in output lo stesso testo di 26 simboli che là costituiva l’input.

Un altro piccolo esempio, che serve per provare e giustificare il caso particolare del ramo **else** dell’istruzione condizionale in fase di decompressione, è su

$$C = 0, 1, 2, 2, 3, 7, 4$$

(con  $l = 7$ ) e lo stesso alfabeto A = { *a, b, c* }.

Si risale al testo

$$T = abccabababc$$

(con  $n = 11$ ), mediante la costruzione di un dizionario con 9 voci:

$$0 \text{ } a \quad 1 \text{ } b \quad 2 \text{ } c \quad 3 \text{ } ab \quad 4 \text{ } bc \quad 5 \text{ } cc \quad 6 \text{ } ca \quad 7 \text{ } aba \quad 8 \text{ } abab$$

Con ancora un poco di pazienza, si può fare la verifica in entrambi i sensi. (Qui tuttavia il risparmio non è significativo.)

I pregi dell'algoritmo LZW consistono nella semplicità di realizzazione e nella applicabilità a qualsiasi tipo di file; ma, proprio a causa di questa genericità, le sue prestazioni spesso non sono elevate, sebbene si sia rivelato particolarmente efficace per file di dati geofisici, e sia utilizzato ad esempio per la codifica di immagini nel formato GIF.

Una nota finale a proposito del dizionario: nella proposta originale, esso può comprendere fino a 4096 voci (12 sono infatti i bit che rappresentano l'indice). Raggiunto tale limite, il dizionario diventa “statico”, e di conseguenza l'algoritmo non lavora bene su quei file (come certi *eseguibili*) in cui la prima parte differisce notevolmente dal seguito, poiché le voci create nel dizionario durante la fase iniziale del processo (cioè nel cosiddetto “periodo di adattamento”) non sono rappresentative dell'intero file.

Come suo ultimo impegno, almeno per il presente paragrafo, raccomando al lettore di provare, sugli esempi proposti in quest'ultima sezione, gli algoritmi descritti in 16.2 e 16.3 (con  $w_{min} = 3$ ); e, infine, di provare tutti gli algoritmi delle sezioni 16.2, 16.3 e 16.4 anche sui seguenti input:

*ababcabcabca*  
*ababcabcabcabca*  
*acabaacaabaacacaabaacba*

con lo stesso alfabeto di tre simboli, mettendo poi a confronto i risultati ottenuti.

## 17. Torri di Hanoi e ribaltamento delle frittelle.

Una classica variante della torre di Hanoi (*cfr.* p. 108) suggerita dallo stesso inventore del famoso puzzle, Édouard Lucas, prevede semplicemente un piolo ausiliario in più: così in tutto i pioli sono quattro, e lo scopo rimane la ricostruzione della torre su uno dei pioli liberi, possibilmente col minor numero di spostamenti di dischi.

È facile verificare che, quando la torre è formata da 3 o 4 dischi, sono necessari e sufficienti 5 o 9 spostamenti, rispettivamente; e con un poco di fatica in più si riesce a provare che, quando i dischi sono 5 o 6, occorrono 13 o 17 mosse. Potrebbe allora sorgere il sospetto di una progressione aritmetica di ragione 4, ma non è affatto così! Si veda infatti la sequenza A007664 in OEIS, chiamata *The Reve's Puzzle*. Tal nome è dovuto a Henry Dudeney, che appose questo titolo al primo dei problemi raccolti nei suoi già citati *Canterbury Puzzles*, ove immaginò che il Castaldo al seguito dei pellegrini, durante la sosta in una taverna, avesse esposto alla compagnia un rompicapo di sua invenzione, promettendo al solutore una pinta della miglior birra: sistemate otto forme di formaggio di dimensione decrescente su uno sgabello, con la più piccola in cima, e chiesti altri tre sgabelli, il problema proposto consisteva nello spostare la pila sull'ultimo sgabello, trasportando una forma alla volta e senza mai posarla su una più piccola. Dudeney costruì la soluzione esatta, oltre che per 8 forme di formaggio (in 33 mosse, mentre per 7 ne occorrono 25), per un qualsiasi numero triangolare (*cfr.* p. 364):

$$\begin{aligned} \text{mosse}(T_1) &= \text{mosse}(1) = 1 \\ \text{mosse}(T_{k+1}) &= 2 \cdot \text{mosse}(T_k) + 2^k - 1 \quad \text{per } k \geq 1 \end{aligned}$$

sicché, ad esempio, per 10 e 21 forme di formaggio sono necessarie, rispettivamente, 49 e 321 mosse, e affermò che se il numero di forme non è triangolare esiste più di un modo per risolvere il problema nel minor numero di mosse.

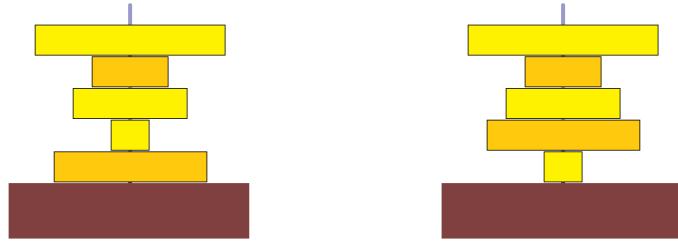
La succitata sequenza A007664 in OEIS è ottenuta con la formula:

$$\text{mosse}(n) = \min \{ 2 \cdot \text{mosse}(k) + 2^{n-k} - 1 \mid k < n \}$$

(B. M. Stewart e J. S. Frame, 1941) basata sul seguente ragionamento: con  $\text{mosse}(k)$  spostamenti, si muovono dapprima i  $k$  dischi più piccoli su un piolo ausiliario; poi, adoperando soltanto l'altro piolo ausiliario, si risolve il classico problema con i restanti  $n - k$  dischi, portandoli sul piolo di destinazione in  $2^{n-k} - 1$  mosse; infine, con altri  $\text{mosse}(k)$  spostamenti, si portano a destinazione i  $k$  dischi più piccoli.

Naturalmente, occorre trovare quel numero  $k$  che rende minimo il valore di questa espressione ricorrente!

I dischi della torre, ovvero le forme di formaggio di diametro decrescente, si prestano alla perfezione per illustrare un altro famoso puzzle. Usiamo questa volta soltanto un piolo, sul quale impiliamo tutti i dischi alla rinfusa; lo scopo è rimetterli in ordine, dal disco più grande in basso, fino al disco più piccolo in alto, sempre facendo il minor numero di mosse. Ogni mossa consiste nell'estrarrre un certo numero di dischi (da 2 a tutti quanti) dalla parte superiore del piolo, *capovolgerli*, e impilarli di nuovo sull'unico piolo disponibile. Vi propongo due casi, entrambi con 5 dischi:



**Soluzione.** Indicando per ciascuna mossa il numero di dischi che si estraggono dalla cima della pila (per poi rimetterli capovolti), l'obiettivo può essere raggiunto

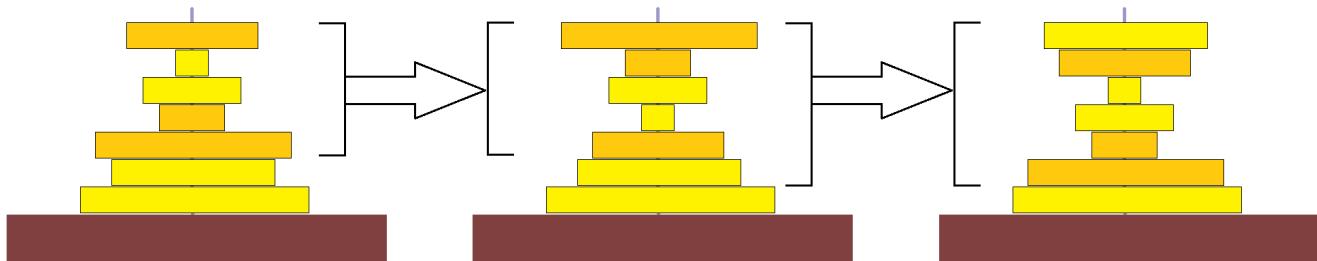
- nel primo caso (figura a sinistra), in quattro mosse: 5, 4, 2, 3;
- nel secondo caso (figura a destra), in cinque mosse: 5, 2, 4, 2, 3.

In letteratura, questo problema, proposto nel 1975, è noto come *pancake sorting* o *pancake flipping* (ordinamento o ribaltamento delle frittelle), in termini più tecnici *sorting by prefix reversals*, poiché ad ogni passo si può soltanto invertire l'ordine degli elementi di un prefisso della sequenza da ordinare. L'ordinamento dev'essere raggiunto nel minor numero di passi, cioè di operazioni di capovolgimento di un prefisso arbitrario.

Non voglio guastarvi il piacere di scoprire la storia di questo problema, facendo da soli una piccola ricerca in rete... Constaterete, tra l'altro, che la questione non è fine a sé stessa, ma – come più spesso accade con tal genere di arzigogoli – ha trovato interessanti applicazioni, nel calcolo parallelo e persino in biologia computazionale!

Vediamo piuttosto di concepire un'idea che ci porti a una soluzione del problema. Possiamo assumere come caso base una pila di  $n = 2$  dischi; i casi sono due: o è già ordinata, o altrimenti il disco più grande sta sopra e allora basta ribaltare tutta la pila (di due dischi) con una sola mossa. Dunque, con  $n = 2$ , faremo, al più, una mossa.

Se la pila è formata da  $n$  dischi, con  $n > 2$ , e non è ordinata, individuiamo il disco più grande che non sia ancora al suo posto definitivo e, se non è già in cima alla pila, ve lo portiamo con una mossa (basta sollevare la parte di pila che ha sul fondo proprio questo disco e ribaltarla); poi, con la mossa successiva, portiamo tale disco al suo posto definitivo. Quindi, otteniamo lo scopo con una o due mosse, diciamo, al più, due.



Nella figura qui sopra sono mostrate le due mosse che portano al suo posto definitivo il secondo disco dal basso; il più grande era già in fondo, al suo posto giusto.

Fatte queste due mosse, rimane da ordinare una pila con un disco in meno, costituita dalla parte più in alto, non ancora in ordine, dell'intera pila.

Basta allora risolvere la seguente equazione alle ricorrenze:

$$\max\_mosse(2) = 1$$

$$\max\_mosse(n) = 2 + \max\_mosse(n - 1) \quad \text{per } n > 2$$

che, com'è facile verificare, ha soluzione  $\max\_mosse(n) = 2n - 3$ , per  $n \geq 2$ .

Abbiamo così trovato un metodo che ci permette di risolvere il problema in non più di  $2n - 3$  mosse... ma purtroppo non conduce, in generale, alla soluzione nel numero minimo di mosse! È tuttavia un ulteriore buon esempio di strategia *decrease-and-conquer* (cfr. p. 375); si noti che qui, in realtà, la dimensione del problema decresce sì, ma in modo non regolare: né di una costante, né di un fattore costante ad ogni ricorrenza.

È altresì evidente che, fissato il numero  $n$  di dischi, il numero minimo di mosse necessarie per ordinarli dipende dalla disposizione iniziale dei dischi stessi: fra le  $n!$  permutazioni, ce n'è una che richiede zero mosse; quante e quali richiedono invece il *massimo* numero minimo di mosse? Il minimo numero di mosse necessarie nel caso peggiore (sequenza A058986 in OEIS) è compreso tra  $(15/14)n$  e  $(18/11)n$ , ma il suo valore preciso non è noto. Alcuni hanno cercato di classificare le permutazioni in base alla loro "difficoltà"; il numero di pile che richiedono il massimo numero minimo di mosse si trovano nella sequenza A067607 in OEIS: ad esempio, si scopre che soltanto due delle 720 permutazioni di 6 dischi richiedono 7 mosse (il massimo), mentre tutte le altre possono essere ordinate con un numero inferiore di ribaltamenti (si veda anche la sequenza A092113 in OEIS).

Un risultato degno di nota è che il problema di determinare una soluzione ottima, data una qualsiasi permutazione degli  $n$  dischi, è NP-arduo (L. Bulteau e altri, 2011).

Un bel libro che parla – tra i tanti altri – dei due problemi visti nel presente paragrafo è *Algorithmic Puzzles*, scritto da Anany Levitin e Maria Levitin (Oxford University Press, New York, 2011). A proposito di *flipping pancakes*, vi è citata la pagina web <https://www.cut-the-knot.org/SimpleGames/Flipper.shtml> con un *applet* che visualizza il procedimento risolutivo; ma tale pagina è soltanto una piccola parte di un ricchissimo e istruttivo sito, *Interactive Mathematics Miscellany and Puzzles*, curato da Alexander Bogomolny. Raccomando al lettore sia questo sito, sia il testo sopra citato, soprattutto per la valida impostazione didattica, particolarmente attenta sia alle strategie generali per progettare algoritmi risolutivi, sia alle diverse tecniche di analisi.

Vorrei proporre un ultimo problema, lasciando al lettore il compito di scoprire eventuali analogie con il ribaltamento delle frittelle. Supponiamo di dover elevare  $b$  all'esponente  $n$ , con entrambi i numeri interi positivi. Osserviamo che, per calcolare ad esempio  $2^8$ , non sono necessarie sette moltiplicazioni, ma ne bastano tre:

$$2 \times 2 = 4, 4 \times 4 = 16, 16 \times 16 = 256.$$

Desiderando minimizzare il numero di moltiplicazioni, procediamo come segue:

**se**  $n = 1$ , **allora** il risultato è  $b$ ,

**altrimenti** (cioè se  $n > 1$ )

calcoliamo (ricorsivamente)  $b$  elevato all'esponente  $n/2$ ,  
memorizzando il risultato in una variabile  $Z$ ;

**se**  $n$  è pari, **allora** il risultato è  $Z \times Z$ ,

**altrimenti** (cioè se  $n$  è dispari) il risultato è  $Z \times Z \times b$ .

Se operiamo in binario, la divisione intera per 2 comporta soltanto uno *shift* a destra, e il bit eliminato determina la parità (se 0) o la disparità (se 1); ad ogni applicazione del procedimento testé descritto, eseguiremo dunque o una o due moltiplicazioni. Quante moltiplicazioni comporterà il calcolo di  $2$  elevato a 10? E all'esponente 15? Siamo certi che il numero di moltiplicazioni sarà il minimo possibile?

**Soluzione.** Quando l'esponente è 10, il procedimento sopra descritto calcola:

$$2^{10} = 2^5 \times 2^5; \quad 2^5 = 2^2 \times 2^2 \times 2; \quad 2^2 = 2 \times 2,$$

e poi le moltiplicazioni (in tutto quattro) sono eseguite “a ritroso”.

Quando invece l'esponente è 15, le applicazioni ricorsive portano a:

$$2^{15} = 2^7 \times 2^7 \times 2; \quad 2^7 = 2^3 \times 2^3 \times 2; \quad 2^3 = 2 \times 2 \times 2,$$

sicché in questo caso le moltiplicazioni sono sei.

Tuttavia, facendo i conti con carta e penna, per calcolare  $2^{15}$  potremmo procedere così:

$$2 \times 2 = 2^2; \quad 2^2 \times 2 = 2^3; \quad 2^3 \times 2^2 = 2^5; \quad 2^5 \times 2^5 = 2^{10}; \quad 2^{10} \times 2^5 = 2^{15},$$

risparmiando una moltiplicazione. Questo controesempio dimostra che, procedendo come sopra descritto, cioè ripetendo elevamenti al quadrato (*by repeated squaring*), non sempre il numero di moltiplicazioni risulta il minimo possibile.

E questo non accade soltanto in casi particolari (l'esponente 15 è un'unità in meno di una potenza di 2), e d'altro canto il minimo è ottenuto non solo in casi altrettanto particolari (come quando l'esponente è una potenza di 2: lo dimostra il caso  $n = 10$ ). Ad esempio, con esponente 23 il procedimento per ripetizione di quadrati comporta sette moltiplicazioni, ma anche qui una si può risparmiare...

Chi è giunto a leggere sino a questo punto sarà stato sostenuto da validi motivi, ma avrà pure avuto una buona dose di pazienza, sia nel seguire il filo dei ragionamenti, sia nello svolgere i compiti proposti; per parte mia, mi auguro che abbia accresciuto almeno un poco le proprie conoscenze, o quantomeno possa averle viste sotto nuova luce o diversamente fra loro collegate.



## Indice dei nomi

- Ackermann, W. F., 69, 99  
Adleman, L. M., 141, 147, 150  
Agrawal, M., 100, 141  
Akin, E., 365  
Al-Adli ar-Rumi, 192, 196, 212  
Alberti, L. B., 145  
Alessandro III di Russia (Romanov), 154  
Alford, W. R., 139  
Ali ibn Mani, 196  
Allen, J. D., 368  
Allis, L. V., 259, 368  
Allouche, J.-P., 29  
Archimede di Siracusa, 74  
Aristotele, 45, 55  
Assaf, A., 316  
Austin, D., 316
- Babbage, C., 61-62, 78, 86, 146, 247  
Bach, J. S., 40  
Backus, J. W., 92  
Bal, H. E., 264  
Ball, W. W. Rouse, 237-239  
Balogh, J., 127  
Barnette, D., 108  
Bayer, R., 407  
Bellaso, G. B., 145  
Bellman, R. E., 19-20, 131  
Bernoulli, J., 61  
Bernstein, A., 280  
Beverly, W., 203  
Bezzel, M., 211  
Bogomolny, A., 426  
Böhm, C., 170  
Bombelli, R., 75  
Borůvka, O., 123  
Bosák, J., 108  
Bourgeois, N., 347  
Bouton, C. L., 298  
Brady, A. H., 185  
Broline, D. M., 362  
Bulteau, L., 426  
Burns, A. M., 157
- Bylander, T., 362  
Byron, A. A., contessa di Lovelace, 61-62
- Cairns, G., 292  
Campbell, M., 274  
Campbell, P. J., 360  
Carmichael, R. D., 139  
Cayley, A., 333  
Ceriani, L., 58  
Cesare, C. Giulio, 145-146  
Chaitin, G. J., 71-72  
Chapman, N. P., 232  
Chavey, D. P., 360  
Chomsky, A. N., 92  
Christofides, N., 124  
Church, A., 39-40, 44, 55, 62-63, 71  
Clark, K. L., 383-384, 387, 390  
Cocke, J., 357  
Coffin, S. T., 332  
Cohen, H., 141  
Cole, A. J., 290  
Cole, F. N., 143  
Connett, J. E., 153  
Conway, J. H., 173, 175, 182  
Cook, M., 175  
Cook, S. A., 102-103  
Crisippo di Soli, 45, 48  
Crowe, D. W., 108  
Ctesibio di Alessandria, 74  
Curry, H. B., 40, 52-53
- Dantzig, G. B., 113, 119, 131  
Davie, A. J. T., 290  
Davis, M., 365  
Delahaye, J.-P., 303  
Delannoy, H. A., 220, 359  
Dewdney, A. K., 151-156, 162, 168  
Diffie, B. W., 147  
Dijkstra, E. W., 11, 13-14, 19-20, 123  
Dinitz, Y., 22  
Dinur, I., 345  
Dixon, J. D., 144

- Doig, A. G., 114  
 Dósa, G., 126-127  
 Dossena, G., 239  
 Douady, A., 167-168  
 Doyle, A. I. Conan, 229  
 Drensky, V., 365  
 Dudeney, H. E., 192*n*, 216, 228-230, 232, 237-239, 300, 317-319, 323, 424  
  
 Edmonds, J. R., 22, 100, 142, 295  
 Ehrenfeucht, A., 353  
 El-Said, I., 157  
 Enrico I d'Inghilterra, 318  
 Eratostene di Cirene, 73  
 Erdős, P., 362  
 Eriksson, H., 363  
 Erone di Alessandria, 61, 73, 77-78  
 Euclide, 73-74, 100, 139, 147, 290-292, 296  
 Eulero (Euler, L.), 107, 193-195, 200, 303, 333  
 Euwe, M., 29  
  
 Fabergé, P. C., 154  
 Faidutti, B., 283  
 Falkener, E., 201, 204  
 Fatou, P. J. L., 166-167  
 Feige, U., 124, 337  
 Fermat, P. de, 138-141, 143-144, 147  
 Fibonacci (Leonardo Pisano, detto), 27, 32, 188, 190, 291-292  
 Filone di Bisanzio, 74  
 Filone di Mégara, 48  
 Fiorentini, M., 317  
 Fischer, M. J., 99  
 Fitting, M., 40  
 Fletcher, J. G., 323  
 Floyd, R. W., 15-16  
 Ford, L. R., 22  
 Fraenkel, A. A. H., 73  
 Fraenkel, A. S., 299  
 Frame, J. S., 424  
 Fredkin, E., 156, 181-182, 184  
 Friedman, D. P., 53  
 Fulkerson, D. R., 22, 113  
  
 Gantt, H. L., 24  
 Gardner, M., 48, 108-109, 232, 258, 299, 317, 363  
 Garey, M. R., 103, 126  
 Garns, H., 303  
 Gauss, C. F., 100, 137, 139, 144  
 Gautheron, V., 360  
 Gentzen, G. K. E., 56  
 Gerasimov, V., 327  
 Gianutio della Mantia, H., 195  
 Goble, L., 39  
 Gödel, K. F., 39-40, 45, 47-48, 52, 54-56, 62-63, 73, 99, 377*n*  
 Goemans, M. X., 126, 337  
 Goldbach, C., 72-73, 139  
 Golomb, S. W., 317, 329, 331  
 Gonzales, T., 113  
 Goodger, D., 332  
 Gosper, R. W., Jr., 173  
 Gould, W., 303  
 Graham, R. L., 23, 125  
 Granville, A., 139  
 Gray, F., 109  
 Grensjö, A., 364  
 Grossman, J. W., 292  
 Grundy, P. M., 289, 298  
 Guarino, P., 192-193, 195  
 Guglielmo di Occam (Ockham), 72  
 Guglielmo I il Conquistatore, 318  
 Guglielmo II il Rosso, 318  
 Gurvich, V. A., 295  
  
 Hadlock, F., 337  
 Hamilton, W. R., 107-108  
 Harshbarger, E. C., 332  
 Haselgrove, C. B., 323  
 Håstad, J., 337  
 Hausdorff, F., 166  
 Hayes, B., 153-154  
 Hedlund, G. A., 29  
 Hein, P., 299  
 Held, M., 119, 121  
 Hellman, M. E., 147  
 Herbrand, J., 56, 377-379, 384-386, 391  
 Herda, H., 299

- Herik, H. J. van den, 259  
 Hilbert, D., 56, 63, 69  
 Ho, N. B., 292  
 Hopcroft, J. E., 105, 107  
 Hopkins, B., 366  
 Horn, A., 378-380, 383  
 Huang, M.-D. A., 141  
 Hubbard, J. H., 167-168  
 Huffman, D. A., 411-414, 416-418, 420  
 Hwang, R. Z., 124  
  
 Inkala, A., 312  
  
 Jacquard (J. M. Charles, detto), 61  
 Jarník, V., 13, 123  
 Jelliss, G. P., 191*n*  
 Johnson, D. S., 103, 126  
 Johnson, S. M., 113  
 Johnson, W. W., 233-234  
 Jombert, C., 193  
 Julia, G. M., 166-167  
  
 Kapanowski, A., 316  
 Kaplan, C. S., 153-154  
 Karacuba, A., 363  
 Karmarkar, N. K., 113*n*  
 Karp, R. M., 22, 103, 105, 119, 121,  
     314, 337, 346  
 Kasisiki, F. W., 146  
 Kasparov, G. K., 280  
 Kayal, N., 141  
 Khachiyan, L. G., 113*n*  
 Klärner, D. A., 325  
 Kleene, S. C., 62, 79  
 Knuth, D. E., 214, 273, 291, 316, 323,  
     325, 363, 371-374  
 Koch, N. F. H. von, 29, 165-166  
 Kolmogorov, A. N., 72  
 König, D., 104  
 Korf, R. E., 235  
 Kraïtchik, M., 144  
 Kruskal, J. B., Jr., 123-124  
 Kuratowski, K., 105, 345  
  
 Ladner, R. E., 99  
  
 Lamé, G. L. J.-B., 291-292  
 Land, A. H., 114  
 Langton, C. G., 185, 188, 190  
 Larsson, U., 282  
 Lederberg, J., 108  
 Leech Haselgrove, J., 323, 325  
 Lehmer, D. H., 140, 143  
 Leibniz, G. W., 61, 63, 72  
 Lempel, A., 414, 419-420  
 Lenstra, H. W., Jr., 141  
 Levitin, A. e M., 426  
 Lewin, M., 337  
 Linscott, G., 257  
 Little, J. D. C., 114  
 Lloyd, J. W., 386*n*  
 Loeb, D. E., 362  
 Lorenz, E. N., 164  
 Loyd, S., 212, 232-234, 237, 239  
 Loyd, S., Jr., 232  
 Lucas, F. É. A., 108, 140, 142-143, 220,  
     228, 282, 359, 424  
 Ludgate, P. E., 78  
  
 Mairan, J.-J. D. de, 193  
 Mandelbrot, B. B., 155, 165-169  
 Margolus, N. H., 182  
 Markov, A. A., Jr., 351  
 Marsland, T. A., 274  
 Martin, B., 155-156  
 Matijasevič, Ju. V., 71, 105, 351  
 McCarthy, J., 53, 63*n*, 272  
 McGuire, G., 335  
 Menabrea, L. F., 61  
 Mersenne, M., 142-143, 145  
 Meulen, M. van der, 259  
 Meyer, A. R., 69, 97, 99  
 Meyrignac, J.-C., 204  
 Miller, D. B., 184  
 Miller, G. L., 141  
 Minsky, M. L., 272, 357  
 Moivre, A. de, 193, 196  
 Montmort, P. R. de, 193  
 Moore, R. W., 273  
 Moorsel, C. van, 282  
 Morgenstern, O., 255*n*

- Morse, H. C. M., 29-30, 289  
 Morse, S. F. B., 410, 416  
 Mortensen, J. R., 58
- Nash, J. F., Jr., 255*n*, 299*n*  
 Nauck, F., 212  
 Naur, P., 92  
 Neary, T., 353, 357  
 Neumann, J. von, 78, 178, 181, 255*n*  
 Newell, A., 272, 280  
 Newton, I., 74  
 Nicola II di Russia (Romanov), 154  
 Nilakant-ha (della famiglia Bhatta),  
     193-194  
 Nilsson, N. J., 228  
 Norvig, P., 309, 312-313
- Orlin, J. B., 22  
 Orman, H. K., 330  
 Oskolkov, K., 363  
 Ovidio (Publio Ovidio Nasone), 281  
 Ozanam, J., 193, 195
- Pacioli, L. B. de, 32, 34  
 Paget, S. E., 229  
 Parekh, A. K., 342  
 Parman, A., 157  
 Pascal, B., 61, 71, 137  
 Pažitnov, A. L., 327  
 Peano, G., 54-55, 165-166  
 Pearl, J., 235, 274  
 Pickover, C. A., 31, 154-155, 179  
 Pisinger, D., 131  
 Pohl, I., 199  
 Pomerance, C. B., 139, 141, 144  
 Post, E. L., 55, 62, 351, 353, 355, 357  
 Pratt, V. R., 140  
 Presburger, M., 99  
 Prim, R. C., 13, 123  
 Prouhet, E., 29-30, 289
- Rabin, M. O., 99, 141  
 Raphson, J., 74  
 Reinfeld, A., 235, 274  
 Reiter, R., 383, 390
- Ritchie, D. M., 69  
 Rivest, R. L., 147, 150  
 Robertson, G. N., 134  
 Robinson, J. A., 57  
 Rocha, I., 282  
 Rochester, N., 272  
 Rojas, R., 78  
 Romein, J. W., 264  
 Roth, A., 199-200, 205  
 Rothvoß, T., 126  
 Royle, G., 303  
 Rubik, E., 233  
 Rucker, R. von Bitter, 183  
 Rudrata, 191  
 Rumely, R. S., 141  
 Russell, S., 63*n*
- Safra, S., 345  
 Sahni, S., 113  
 Salimbeni, L. e J., 326  
 Sanden, H. van der, 335  
 Saxena, N., 141  
 Scarlatti, G. Domenico, 40  
 Schaeffer, J. H., 235  
 Schönfinkel, M. I., 40, 52  
 Schubert, F. P., 40  
 Schubert, H. C. H., 233-234  
 Scott, D. S., 319, 324  
 Sedgewick, R., 407  
 Seiden, S. S., 127  
 Sgall, J., 126-127  
 Shallit, J. O., 29  
 Shamir, A., 147, 150  
 Shannon, C. E., 146, 272, 279  
 Shasha, D. E., 375  
 Shaw, J. C., 272  
 Shishikura, M., 167  
 Shor, P. W., 150  
 Sierpiński, W. F., 176-177  
 Silver, D., 257  
 Simchi-Levi, D., 126  
 Simon, H. A., 272, 280  
 Simpson, T., 74  
 Singh, M., 124  
 Slocum, J., 233

- Smith, W. E., 135  
 Smullyan, R. M., 35, 39-41, 45, 47-49,  
     52, 54, 57-59  
 Solovay, R. M., 141  
 Sonneveld, D., 233  
 Sprague, R. P., 298  
 Stewart, B. M., 424  
 Stirling, J., 96  
 Stockmeyer, L. J., 97, 104  
 Storer, J. A., 414, 419  
 Story, W. E., 233  
 Strassen, V., 141  
 Strongin, N., 368  
 Szymanski, T. G., 414, 419
- Tait, P. G., 107-108, 220, 222-223,  
     225, 236  
 Tarjan, R. E., 105, 347  
 Tarski, A., 39, 45, 49, 52, 99  
 Tartaglia (Niccolò Fontana, detto),  
     137-138, 141, 176  
 Thue, A., 29-30, 91, 289, 349, 351  
 Toffoli, T., 182  
 Torres y Quevedo, L., 75, 278-279  
 Trojanowski, A. E., 347  
 Tromp, J. T., 368  
 Trotter, H. F., 319  
 Turing, A. M., 43-45, 55, 62-63, 63n,  
     70-71, 78, 91, 95, 173, 175,  
     184-185, 190, 280, 349, 355, 357  
 Turk, G., 185  
 Turner, D. A., 53  
 Tutte, W. T., 108
- Ulam, S. M., 178-179  
 Ullman, J. D., 393, 401, 403
- Vandermonde, A.-T., 195  
 Verhulst, P. F., 162-164, 167  
 Vichniac, G. Y., 182  
 Vigenère, B. de, 145-146  
 Volpe, L. dalla, 195  
 Voronoi, G. F., 153
- Walker, J., 183  
 Wang, H., 357  
 Warnsdorf, H. C. von, 196, 198-200, 205  
 Warshall, S., 16  
 Welch, T., 414, 420  
 Wellerton, H., 282  
 Wenzelides, C., 204  
 Wexler, H., 368  
 Wickelgren, W. A., 228  
 Williamson, D. P., 337  
 Wilson, J., 140  
 Wise, D. S., 53  
 Wolfram, S., 29, 174-175, 183-184, 190n  
 Wong, J. K., 107  
 Woodhead, R., 306  
 Wordsworth, W., 247  
 Wrathall, C., 351
- Zampolini, C., 263, 265, 281  
 Zermelo, E. F., 73, 255  
 Zhao, L., 353  
 Ziv, J., 414, 419-420  
 Zuse, K., 75-78, 184, 279

(Le fotografie alle pagine 178 e 278 sono dell'autore.)

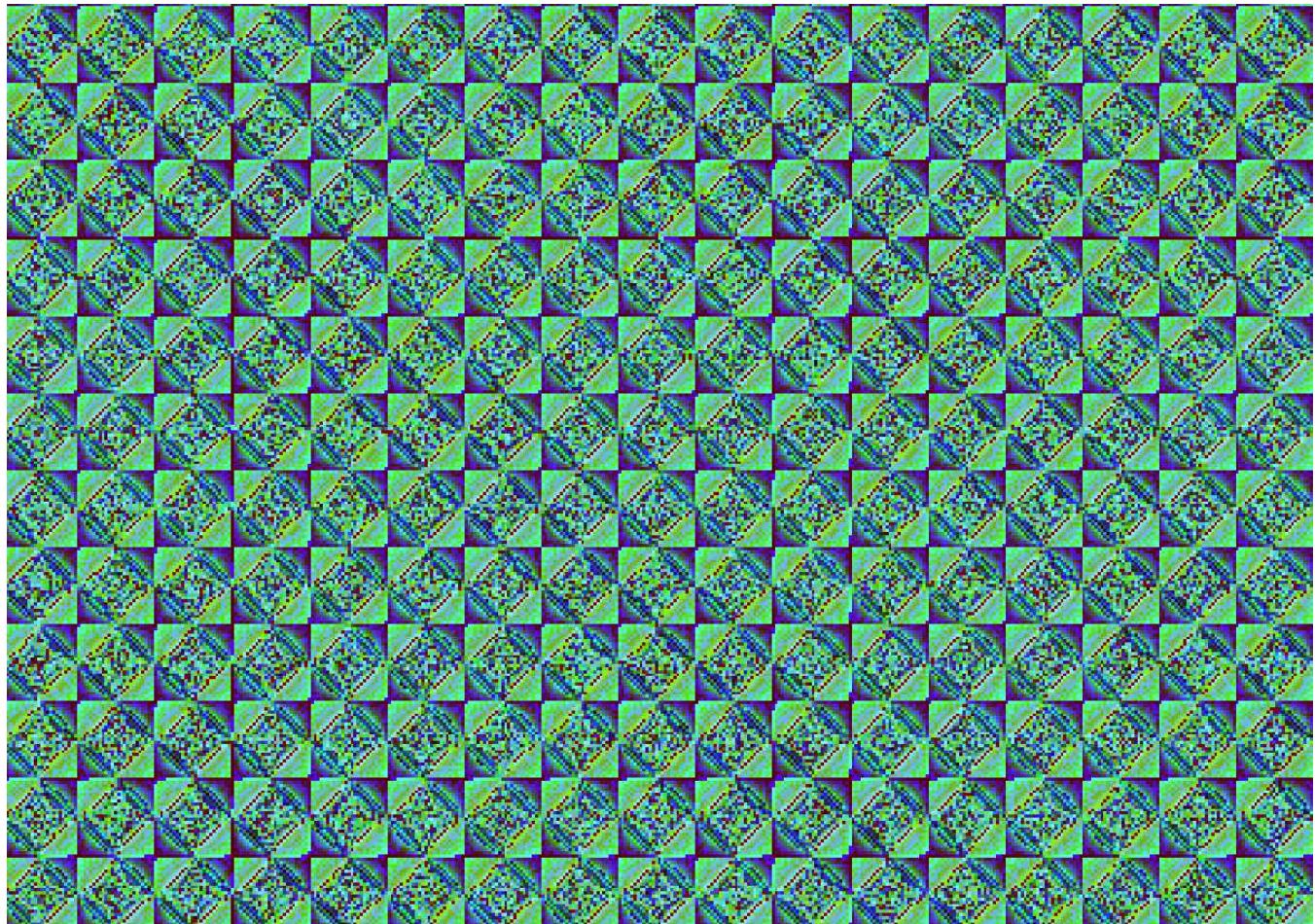
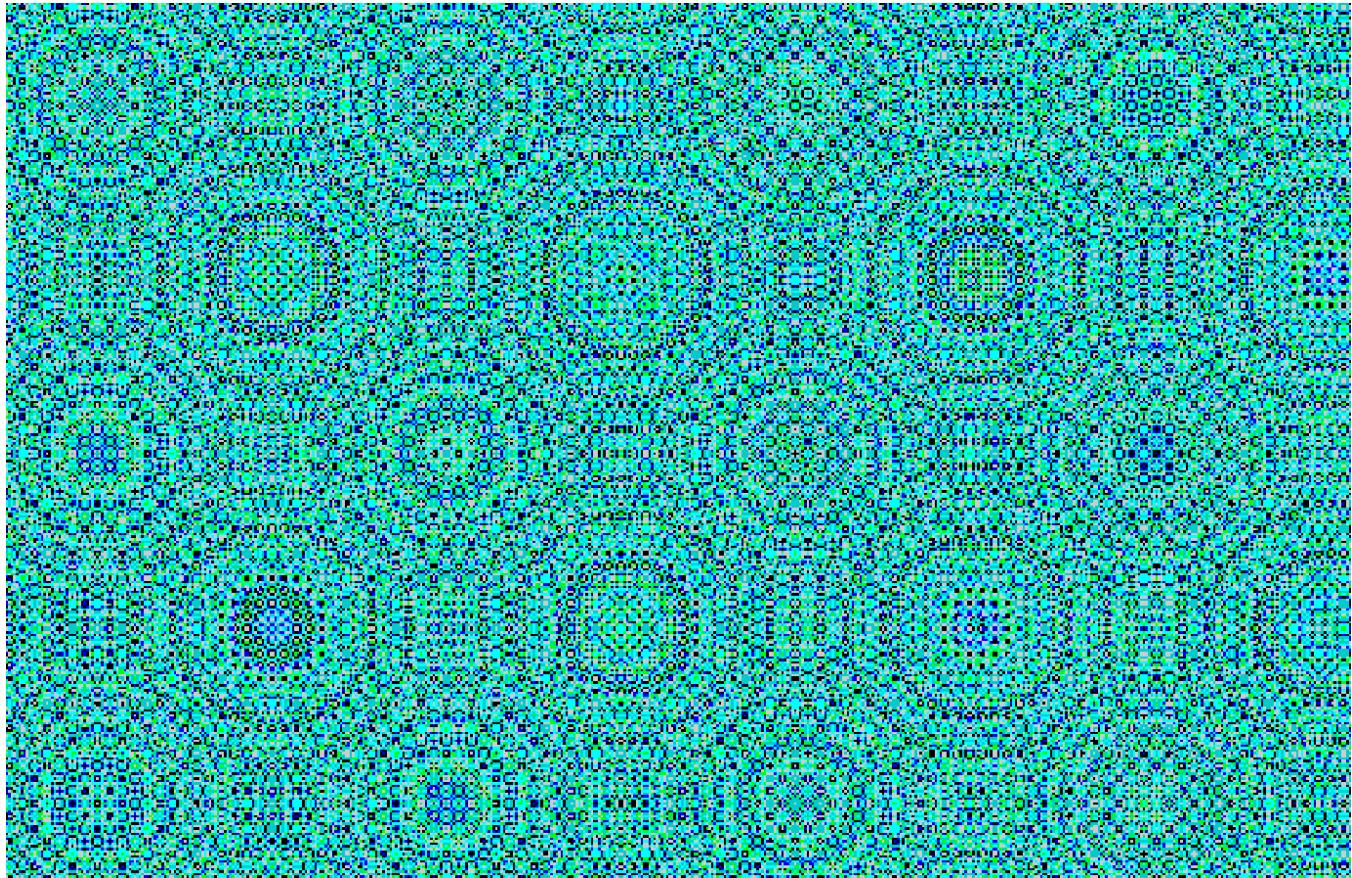


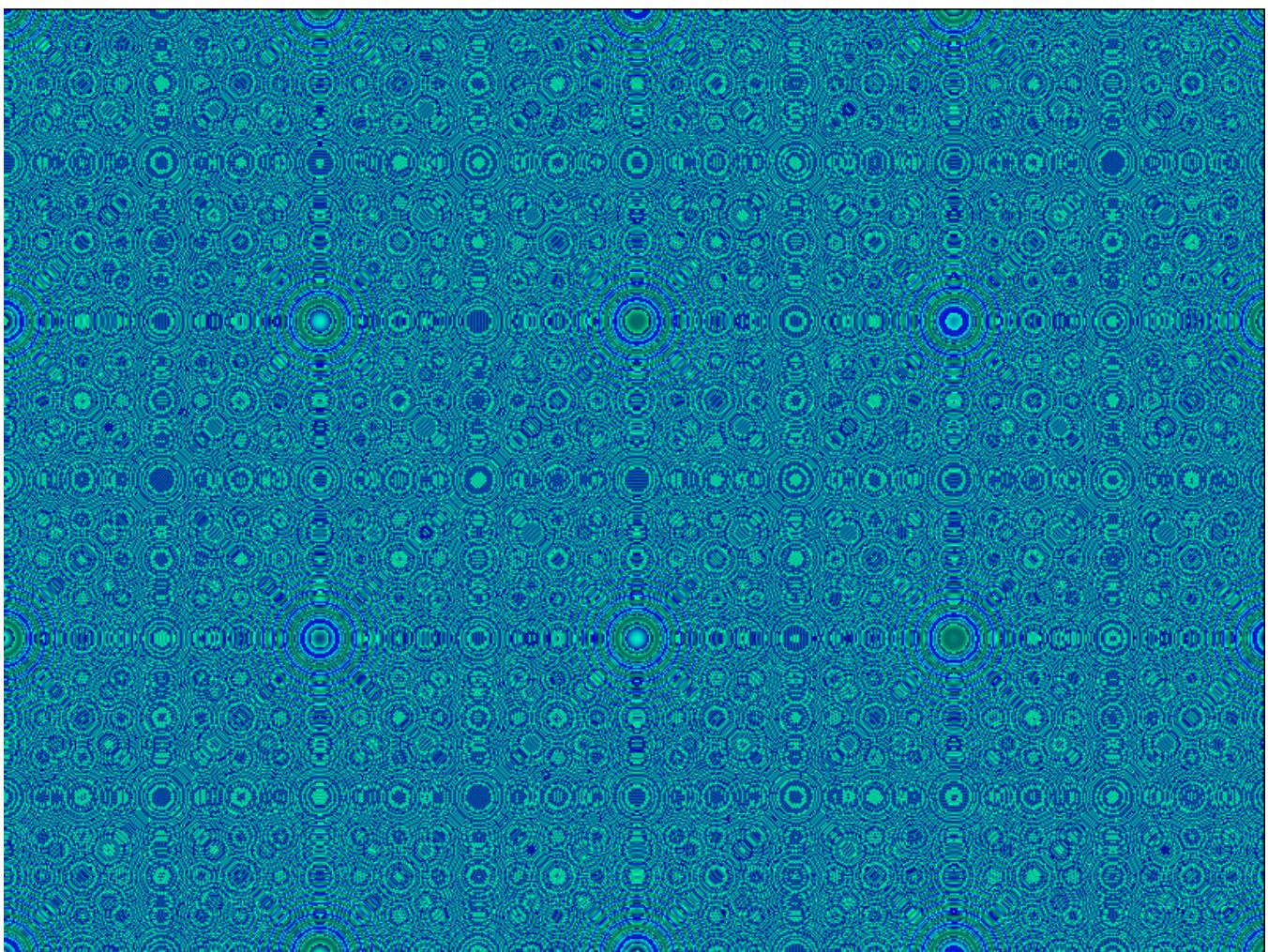
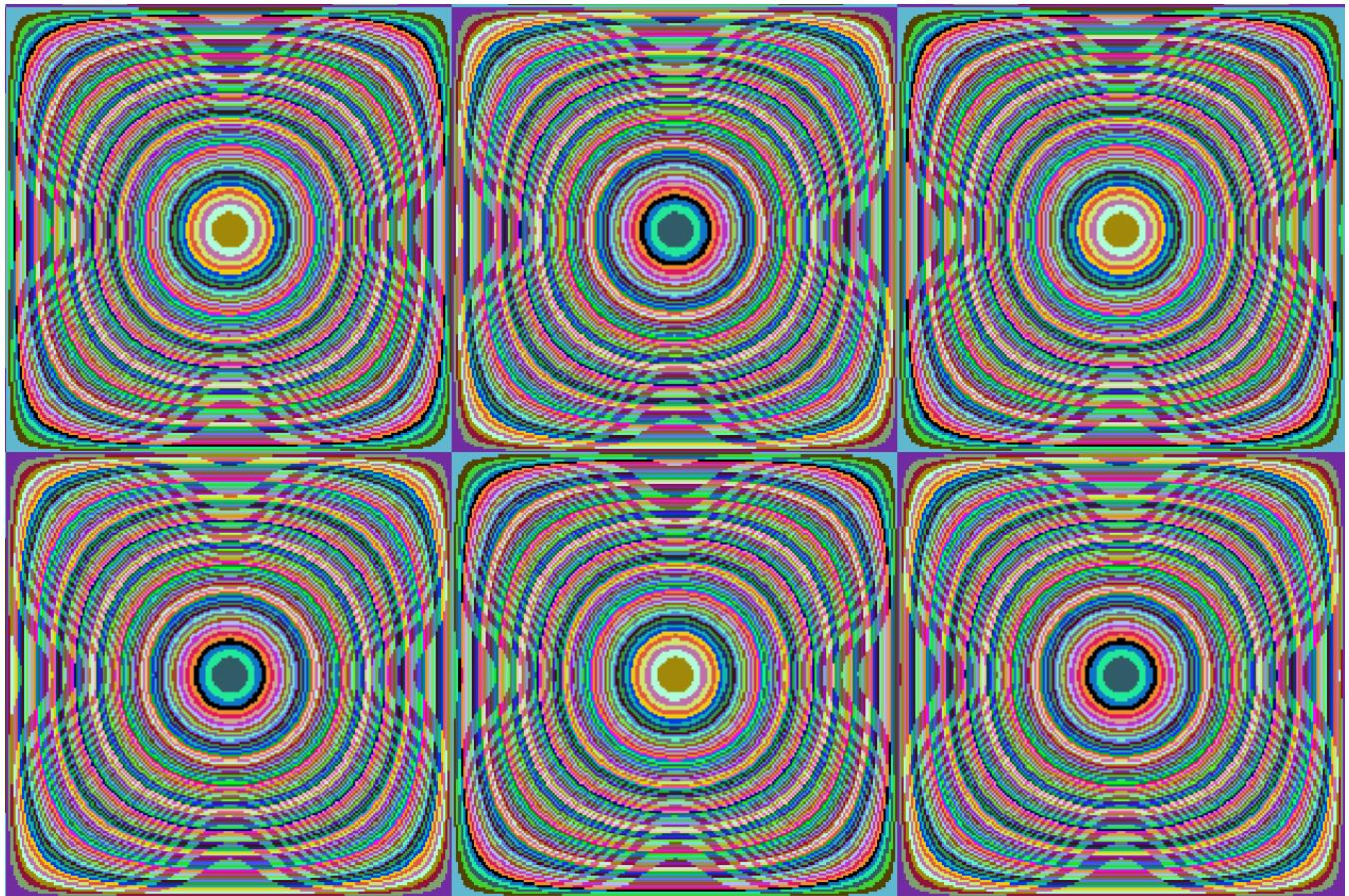
# **Galleria di immagini**

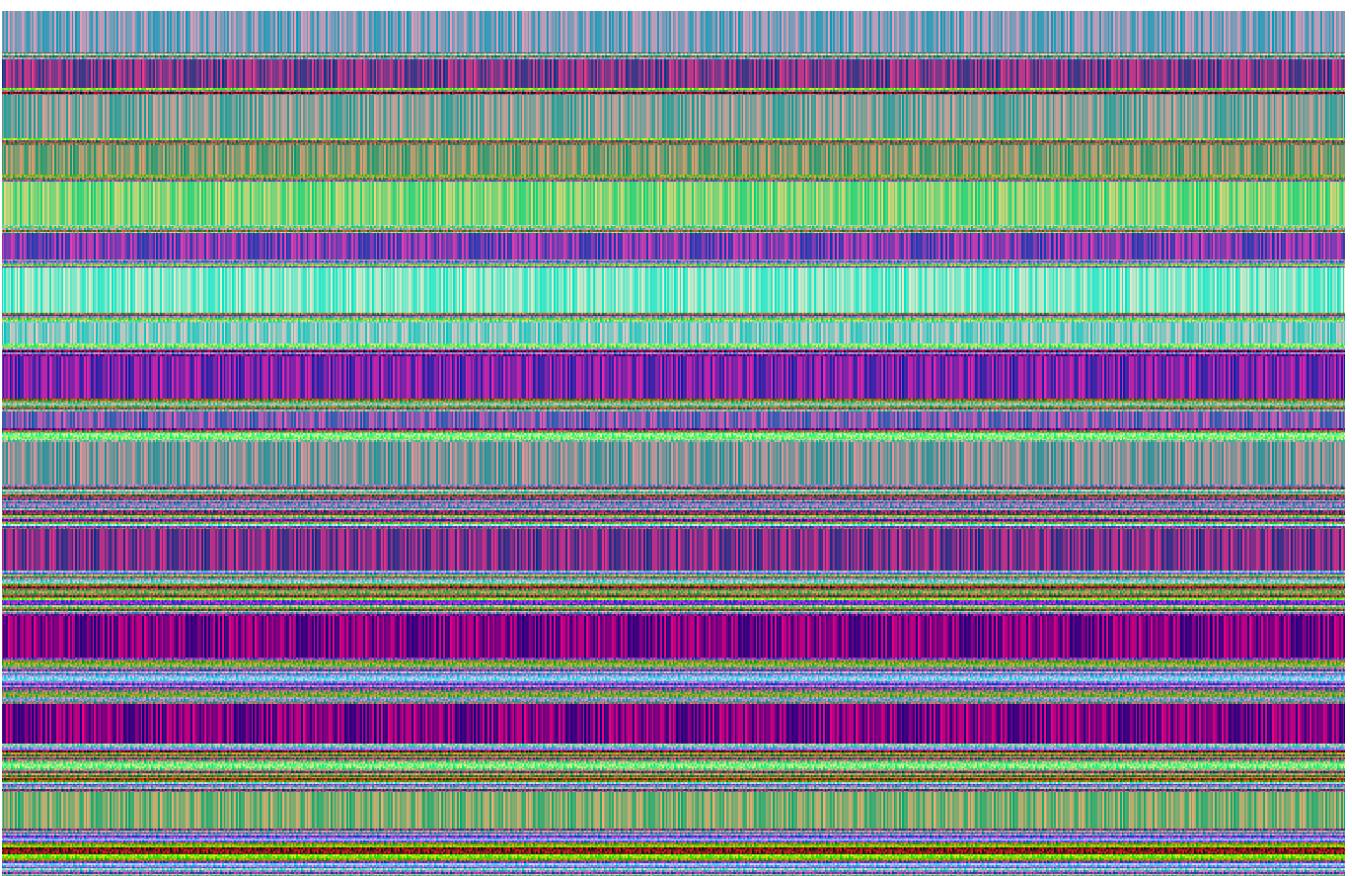
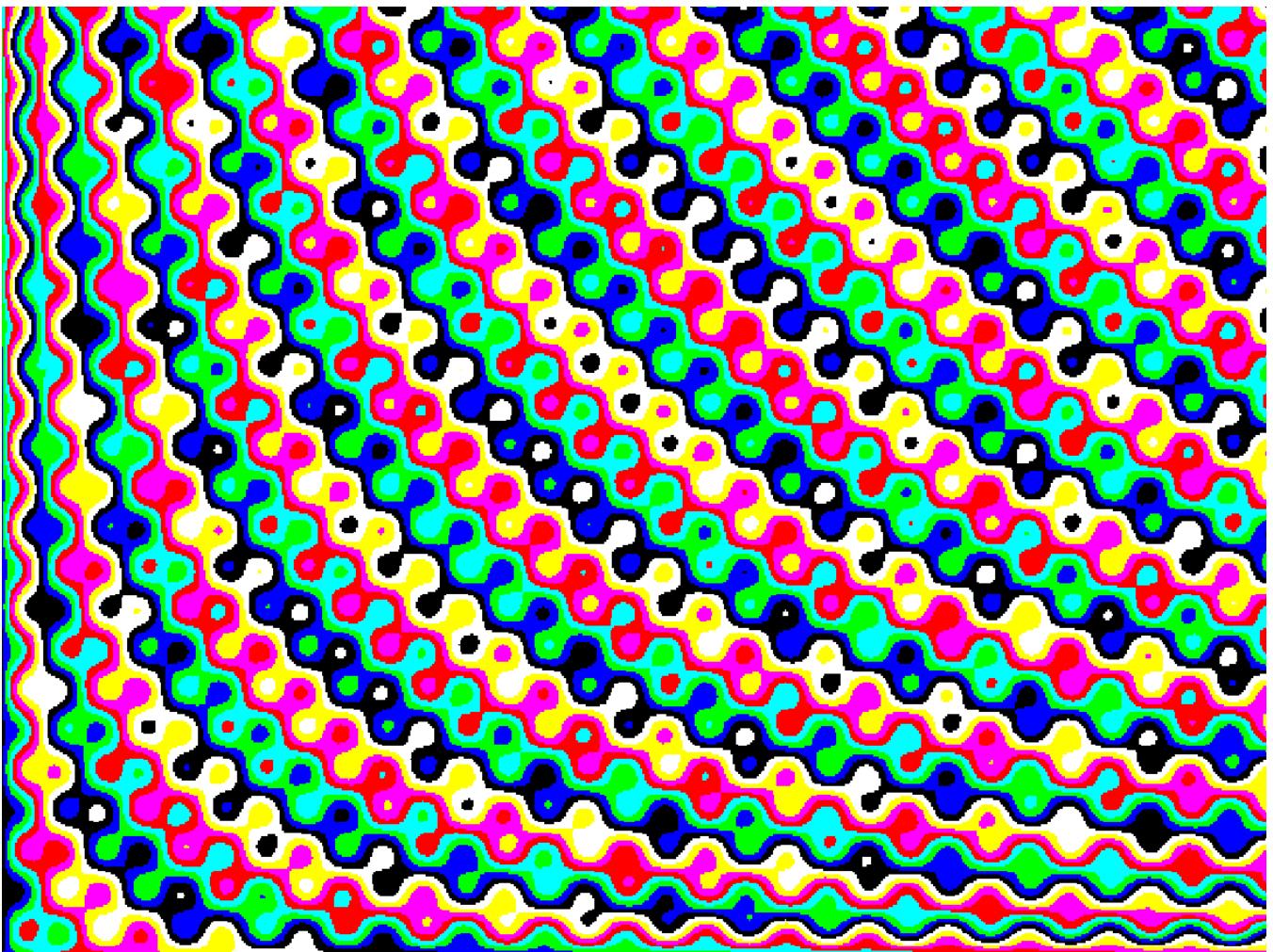
**Tavole a colori fuori testo**

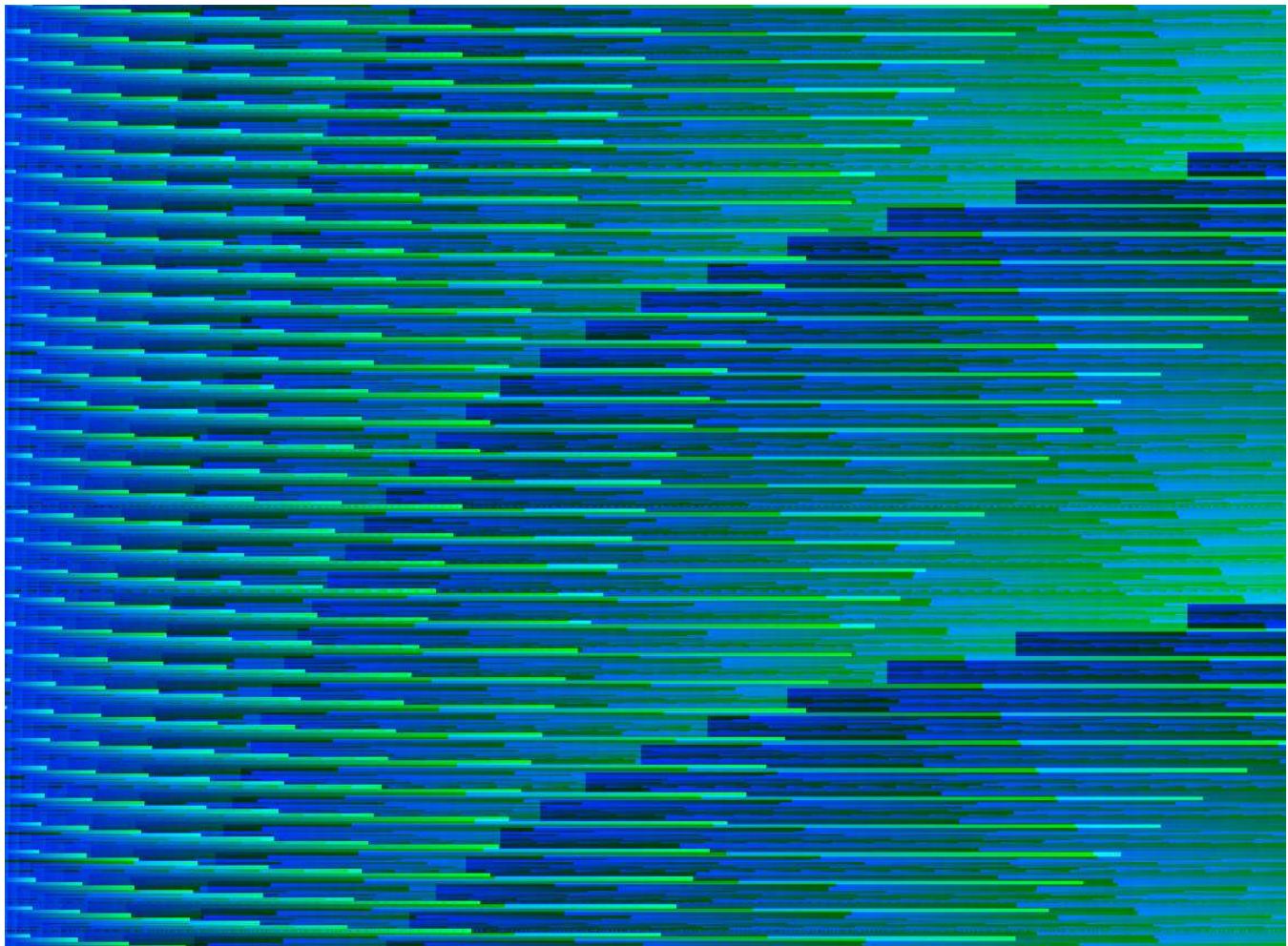
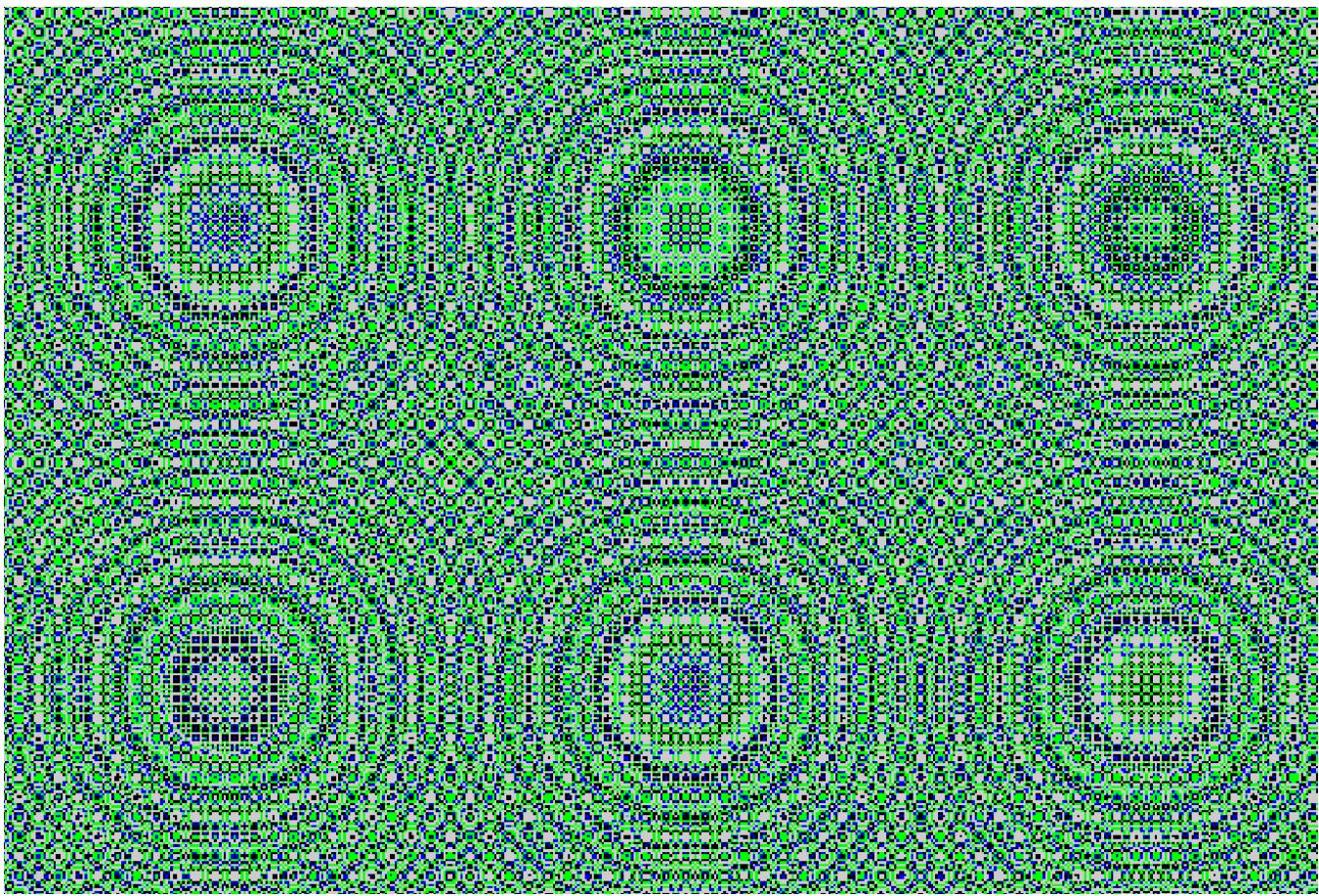


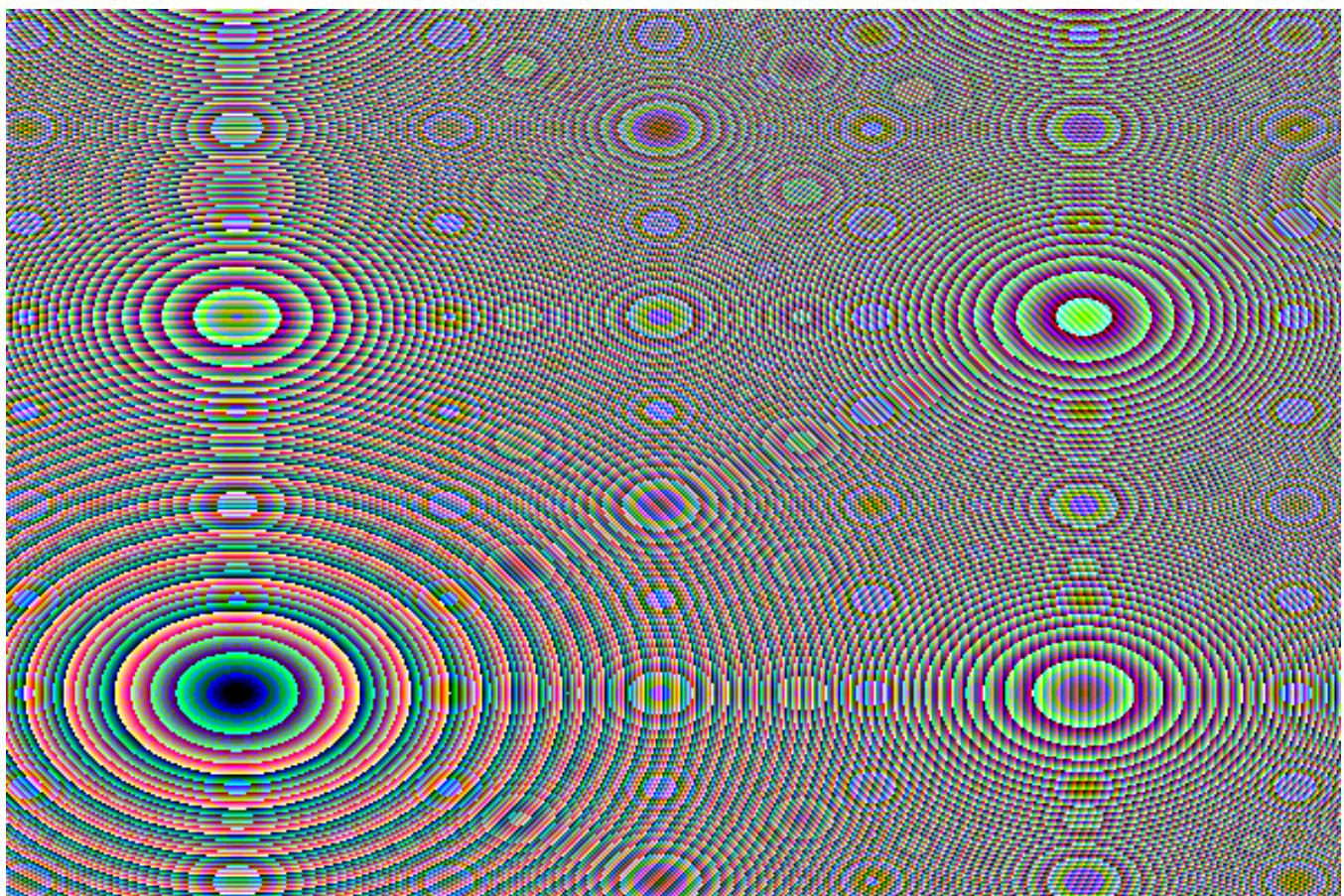
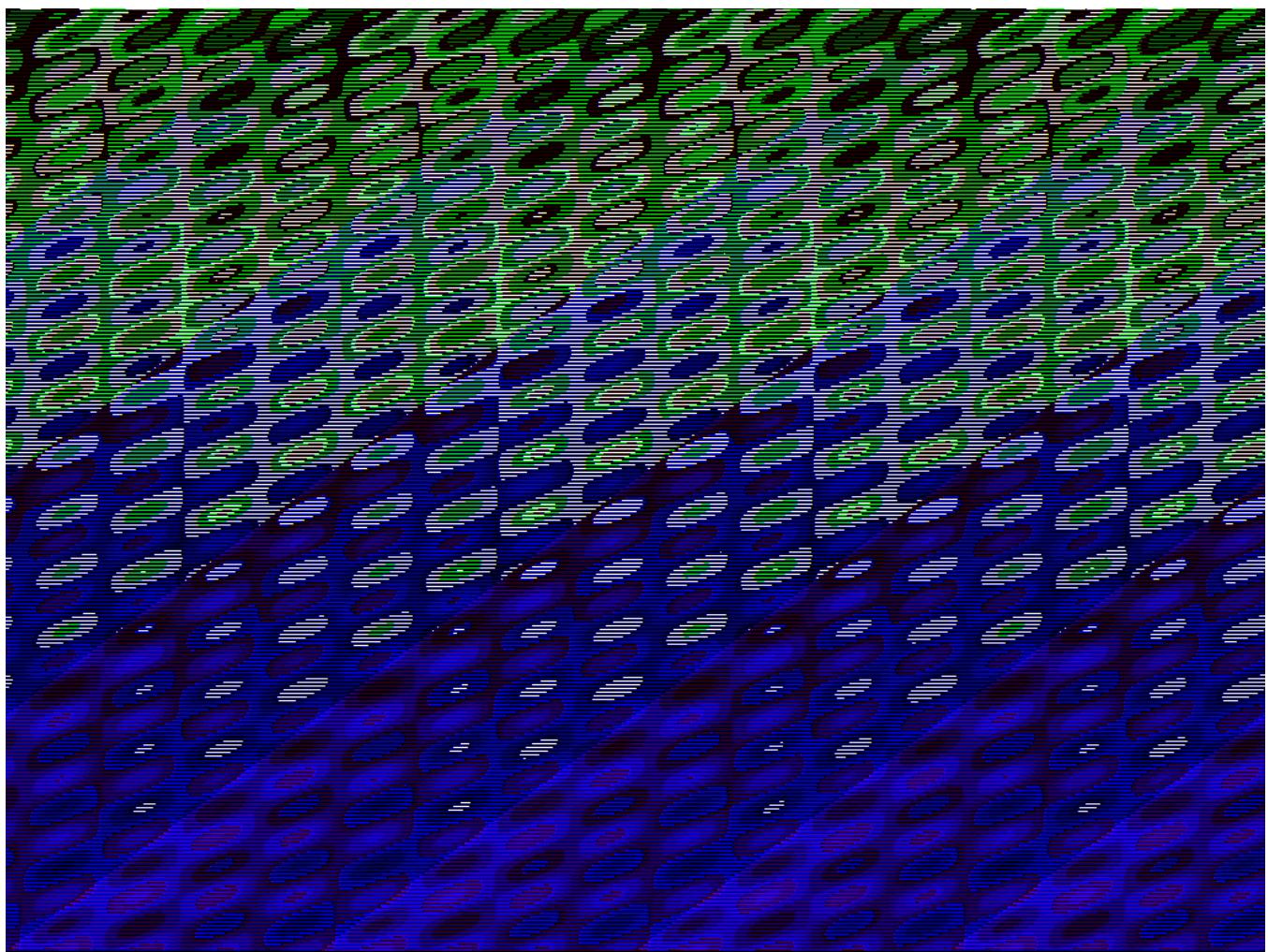
## **Sezione I – Tappezzerie**

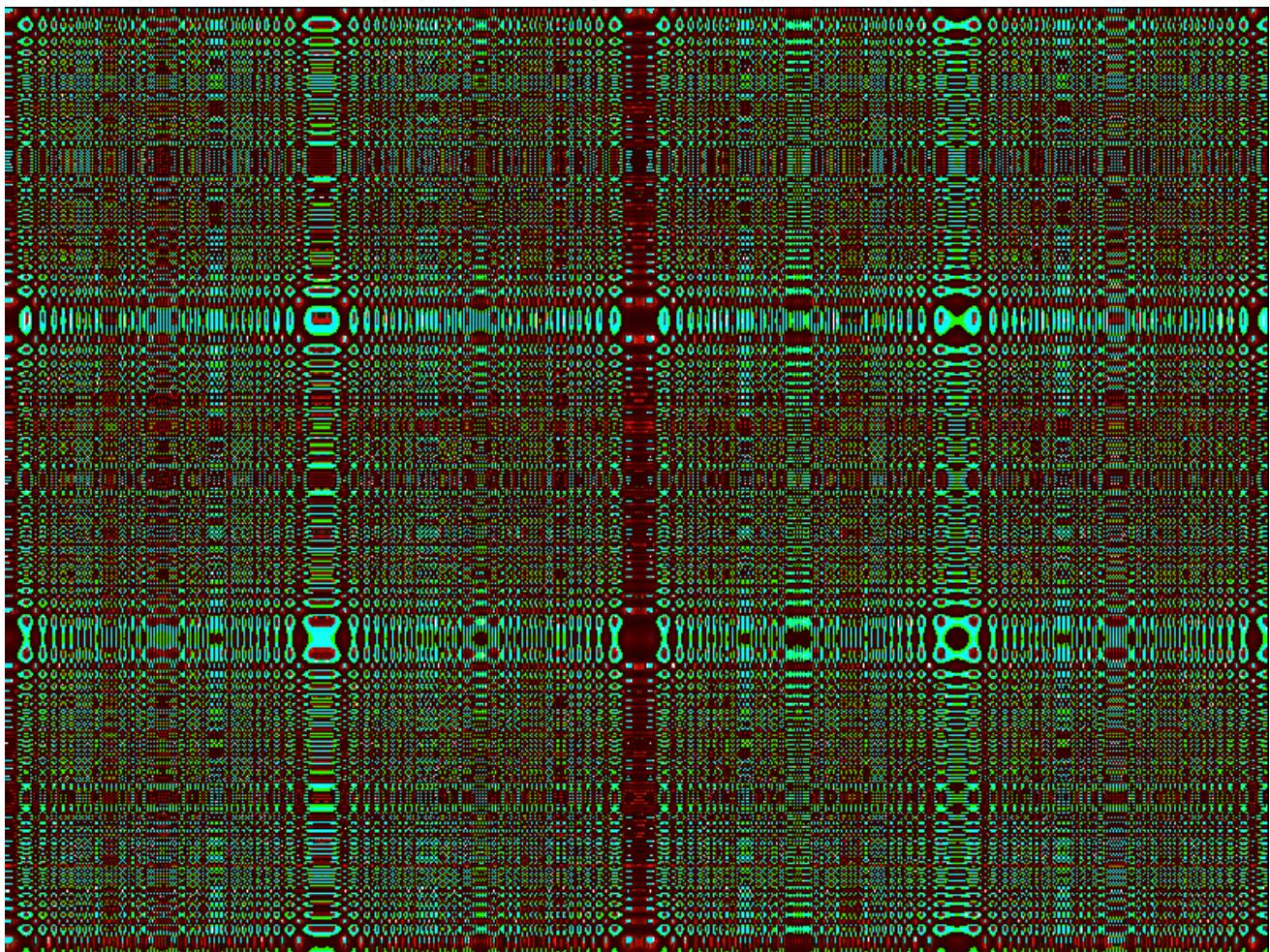
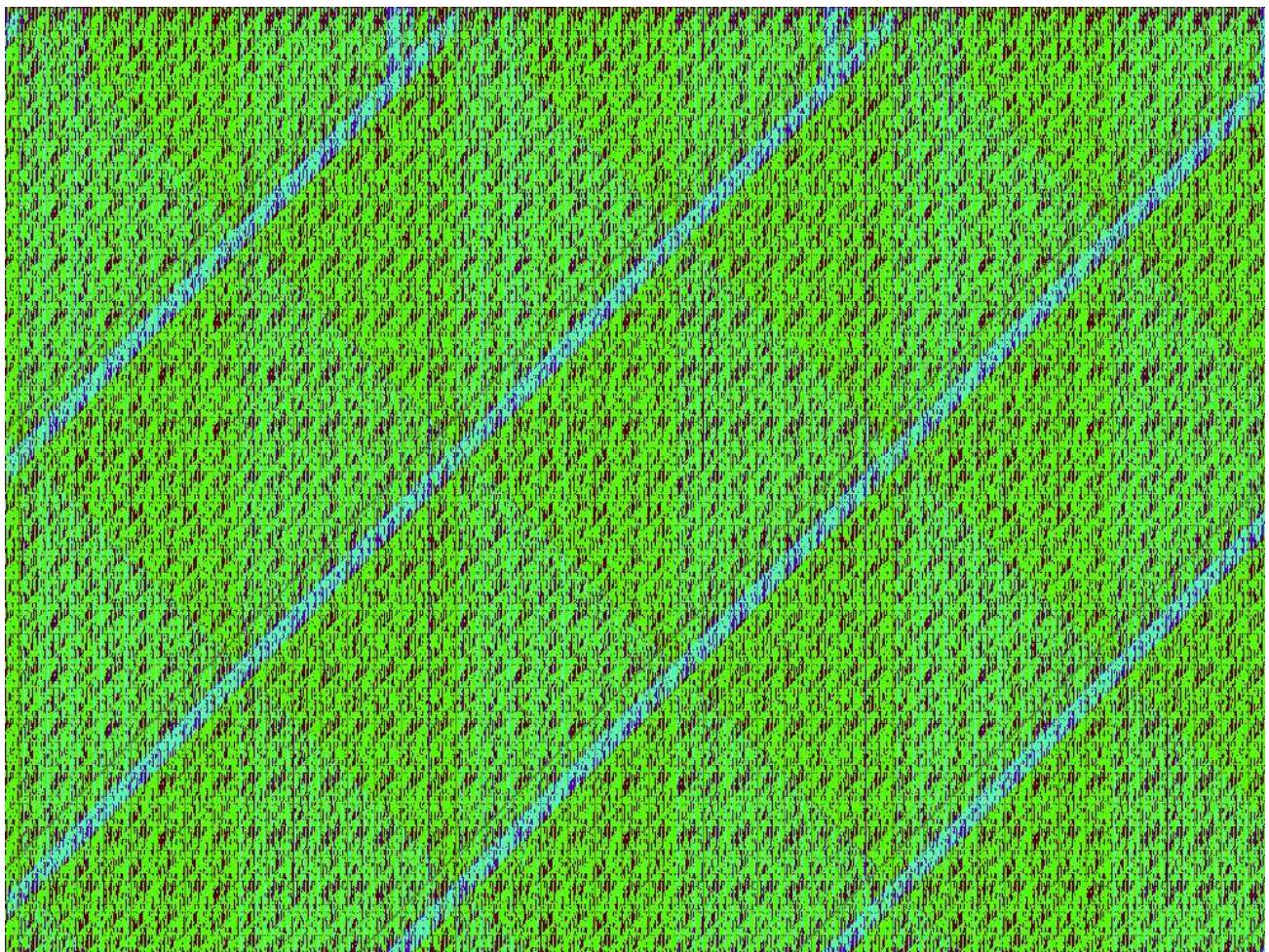


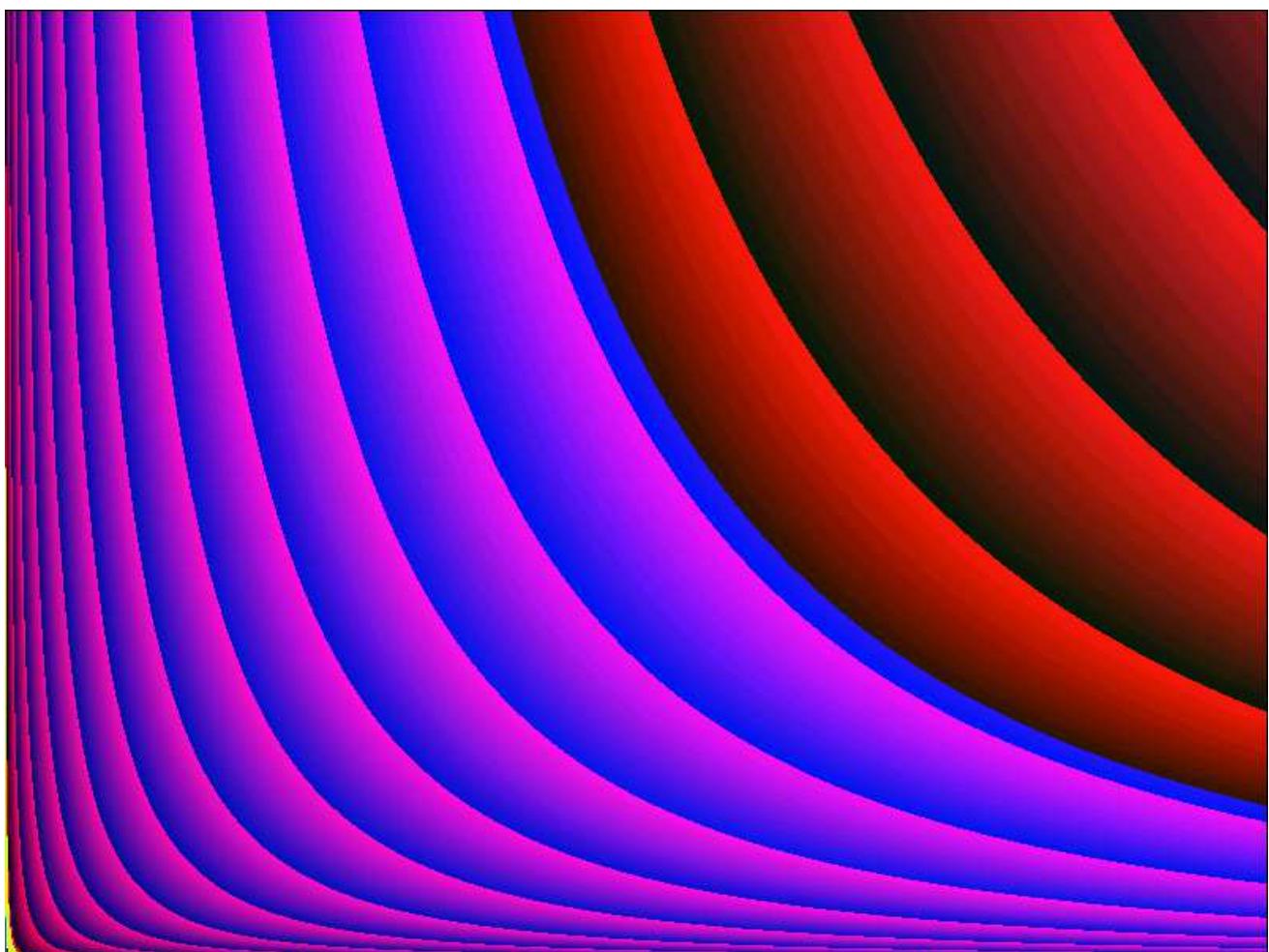
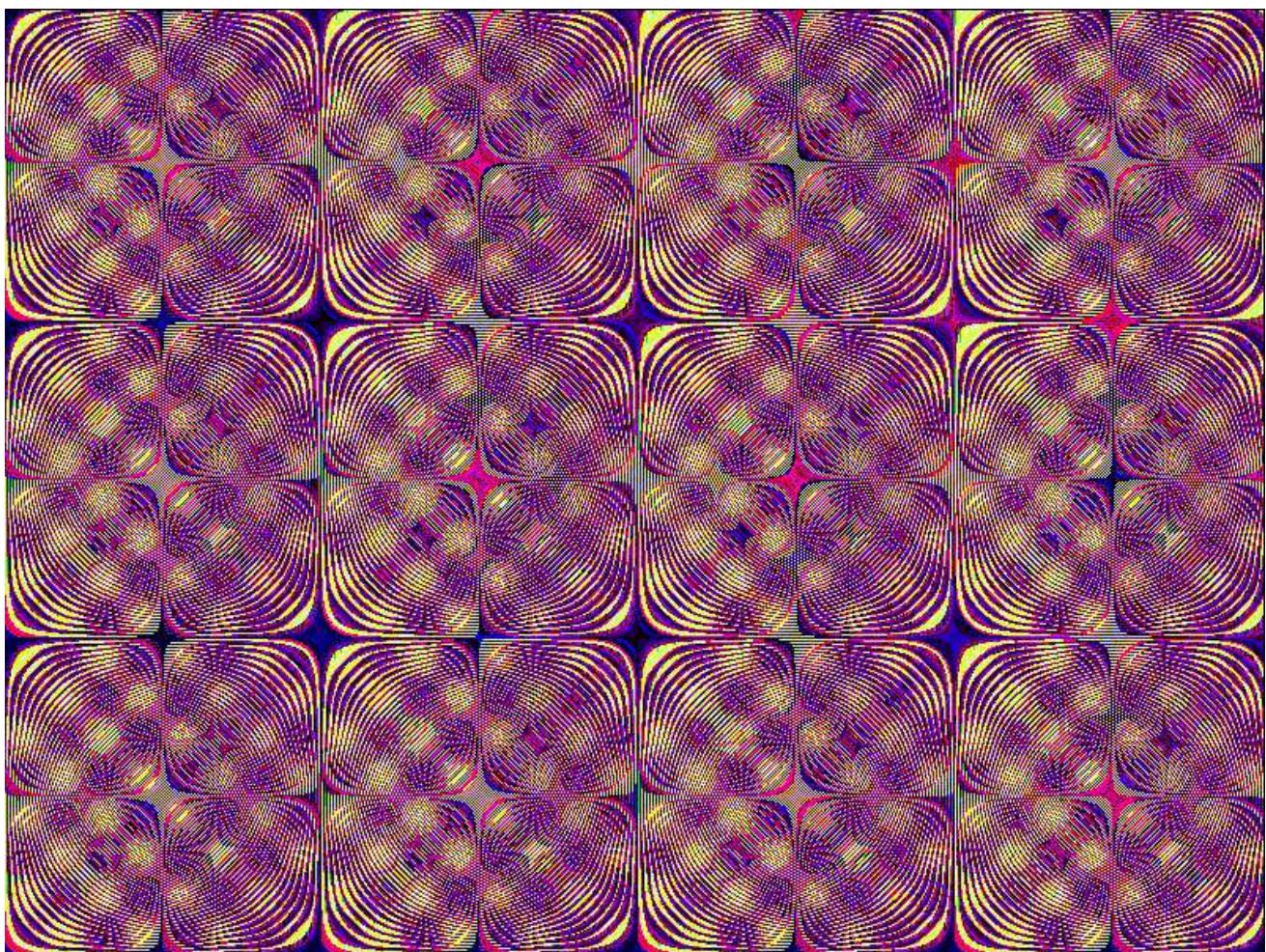


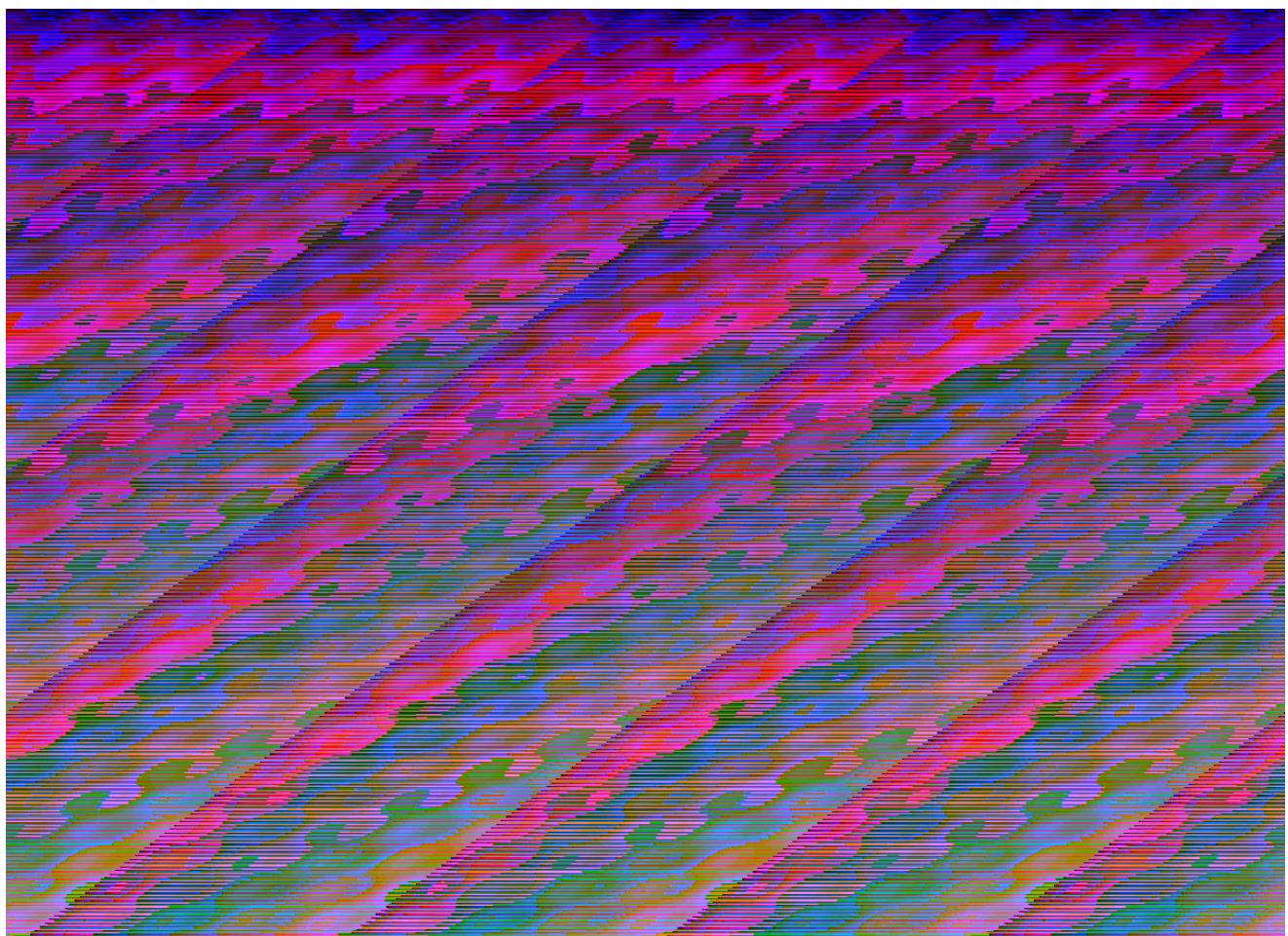
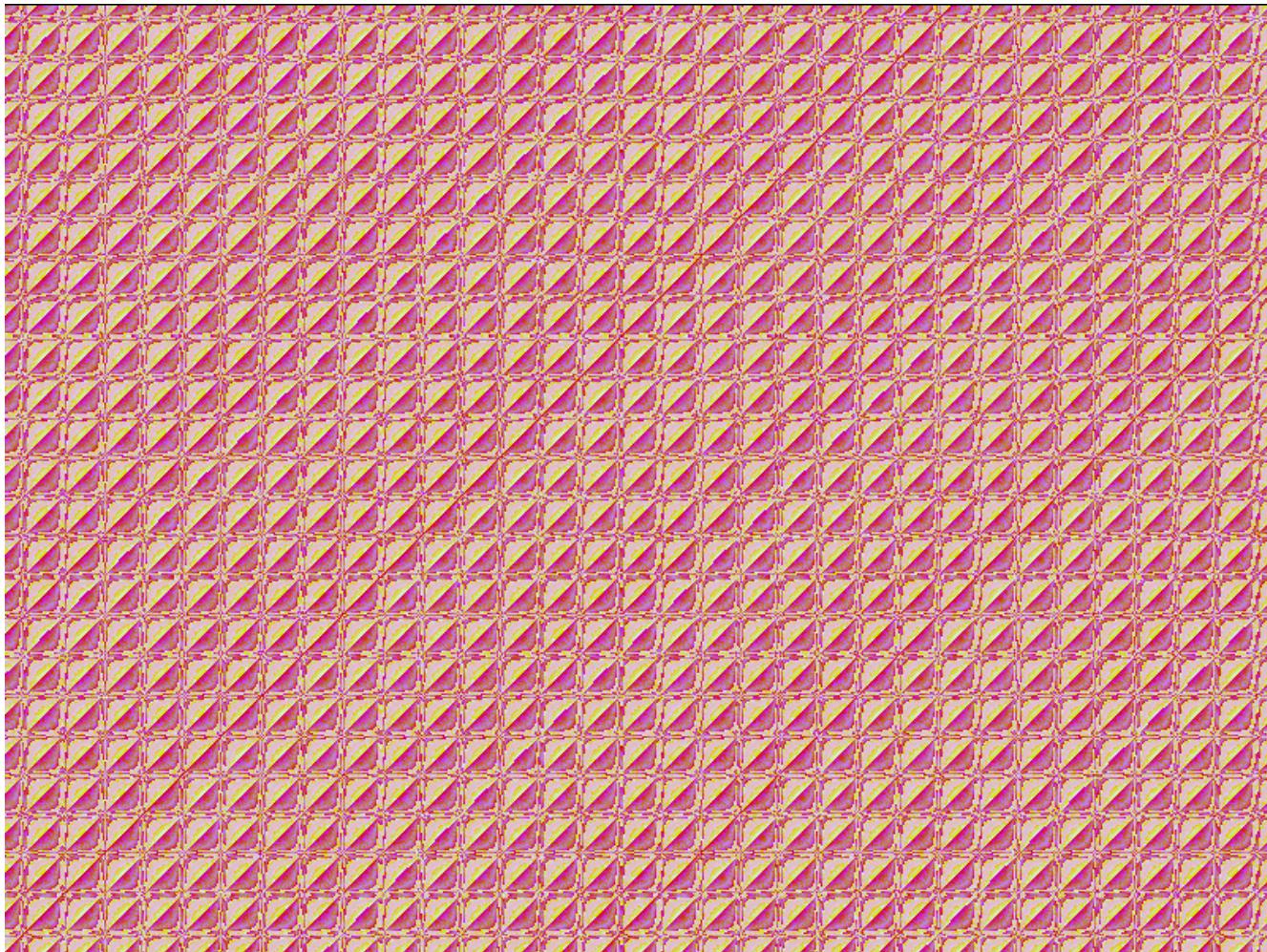


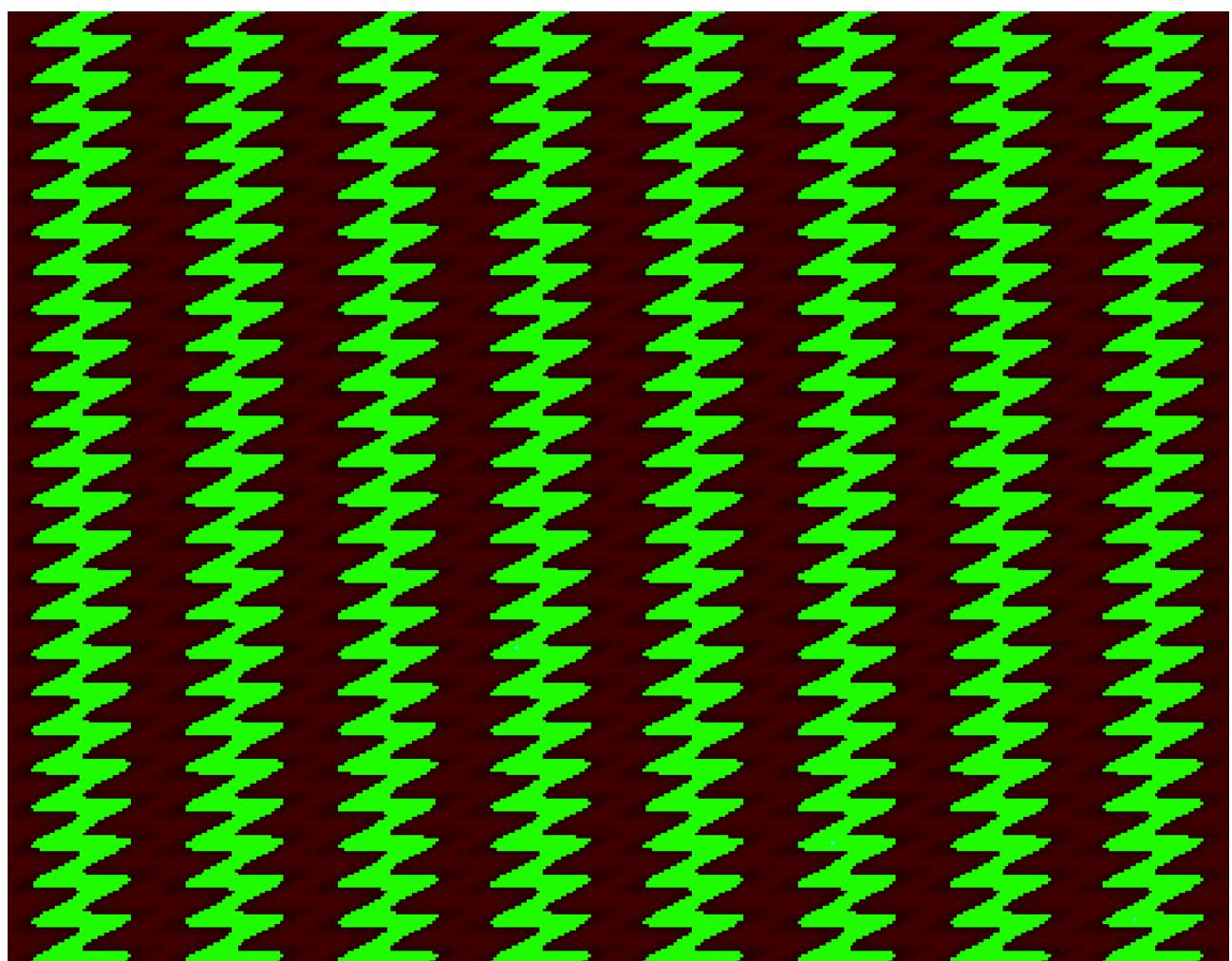
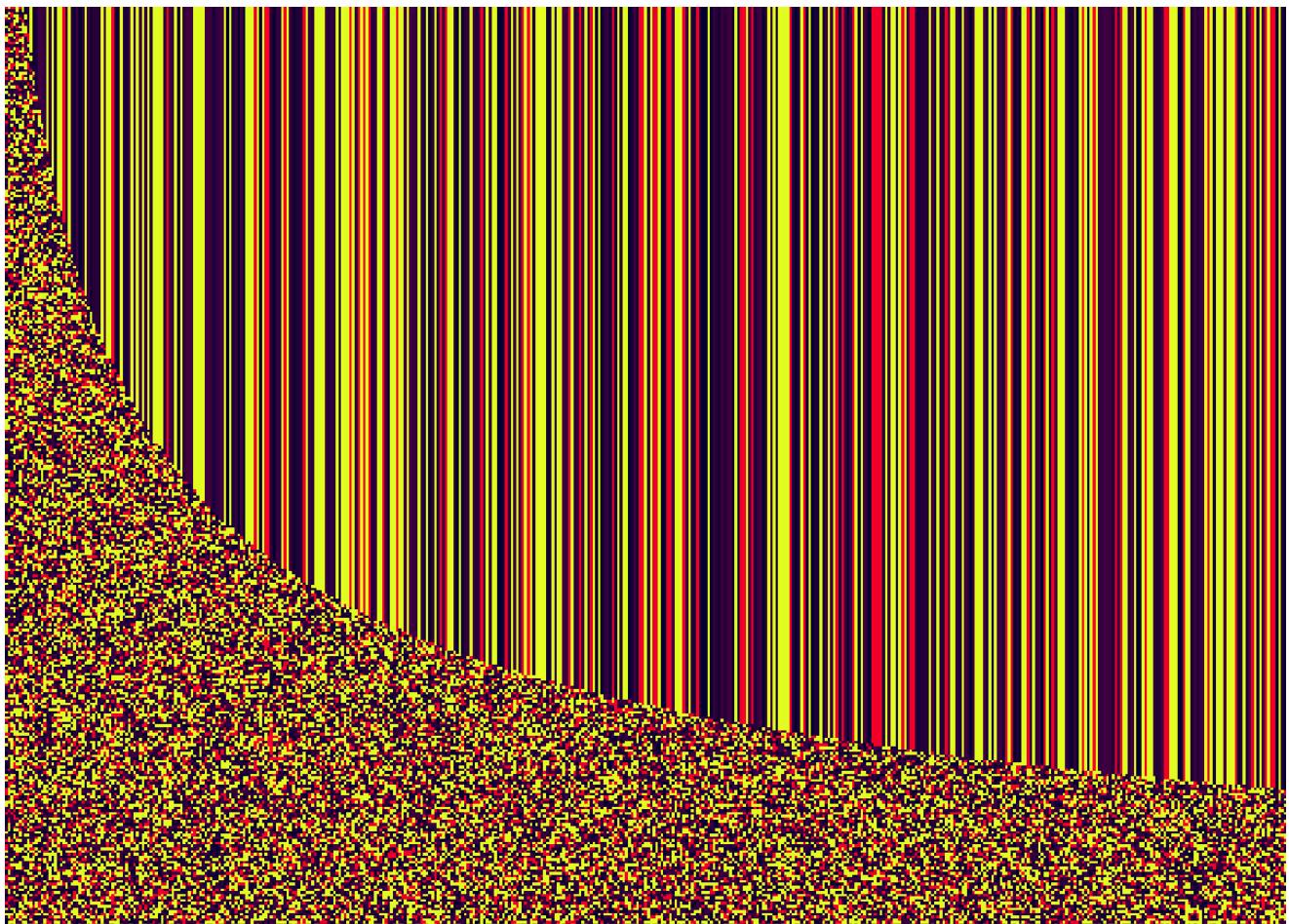


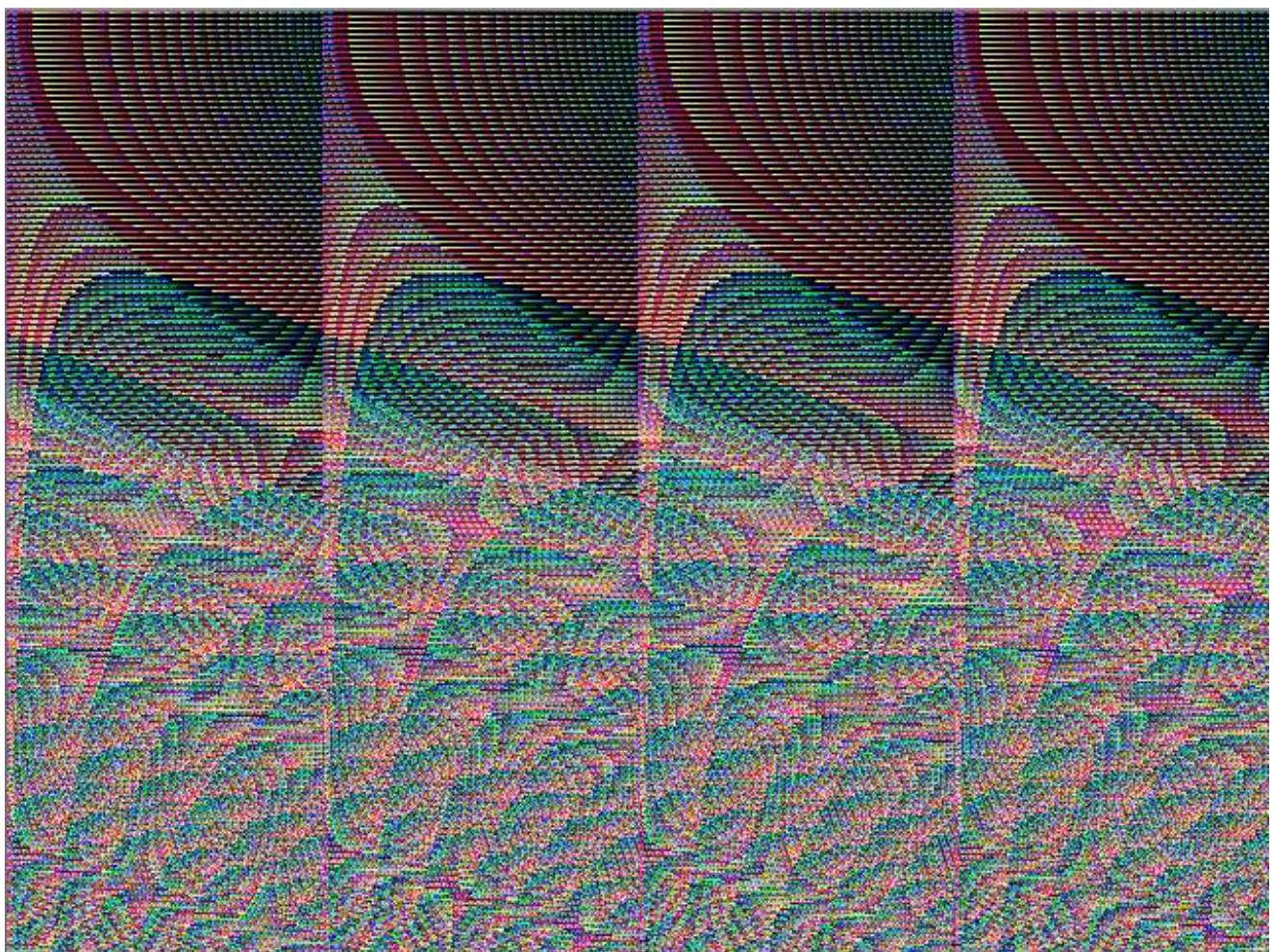
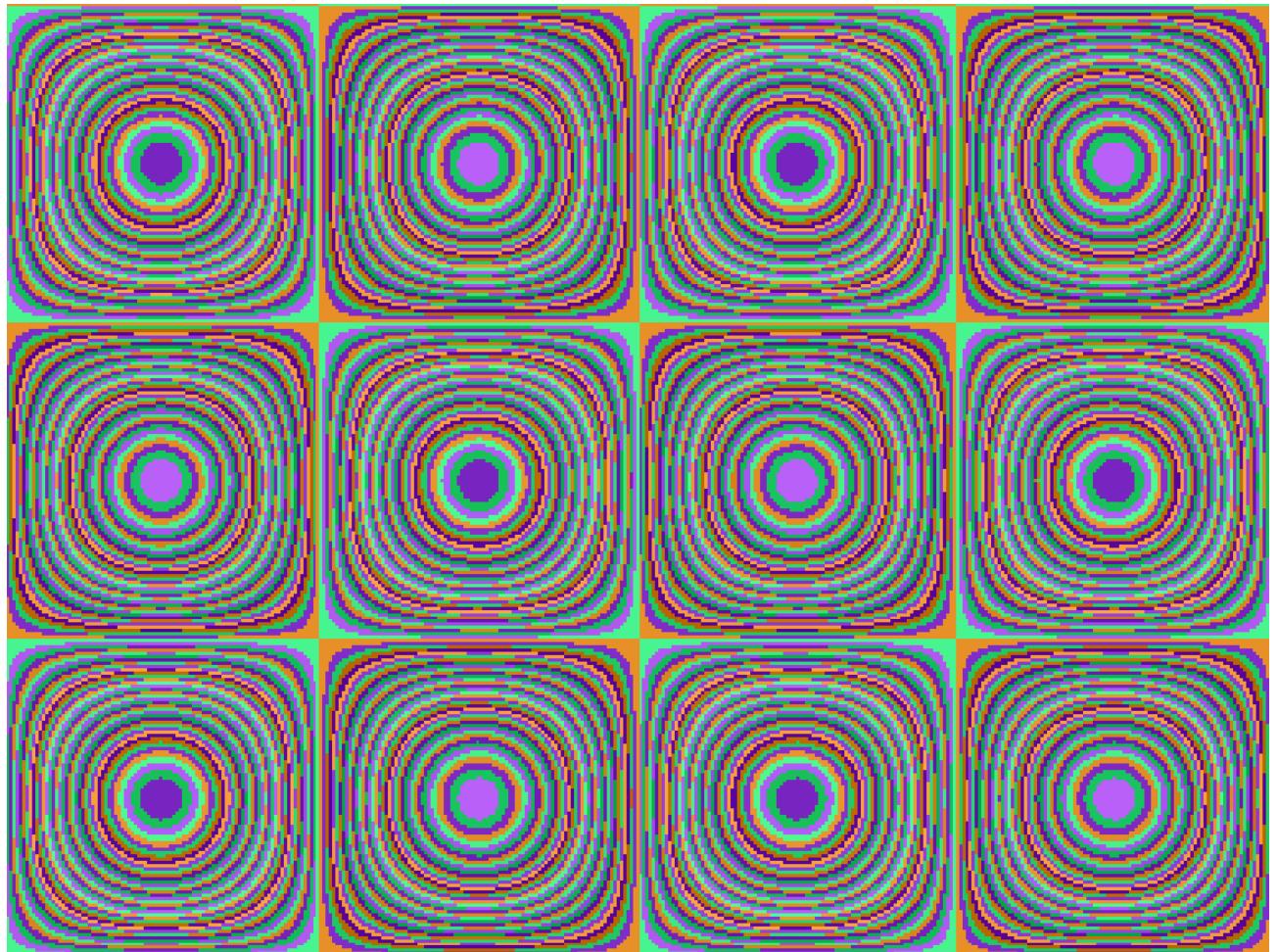


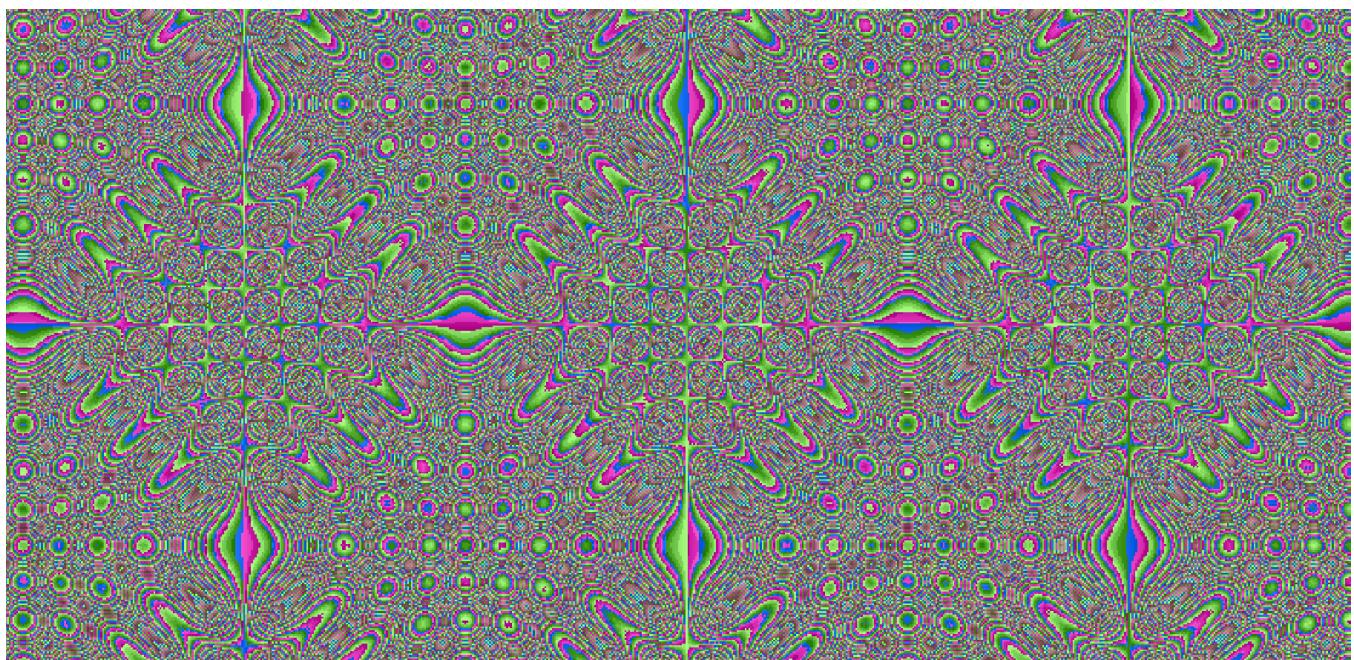
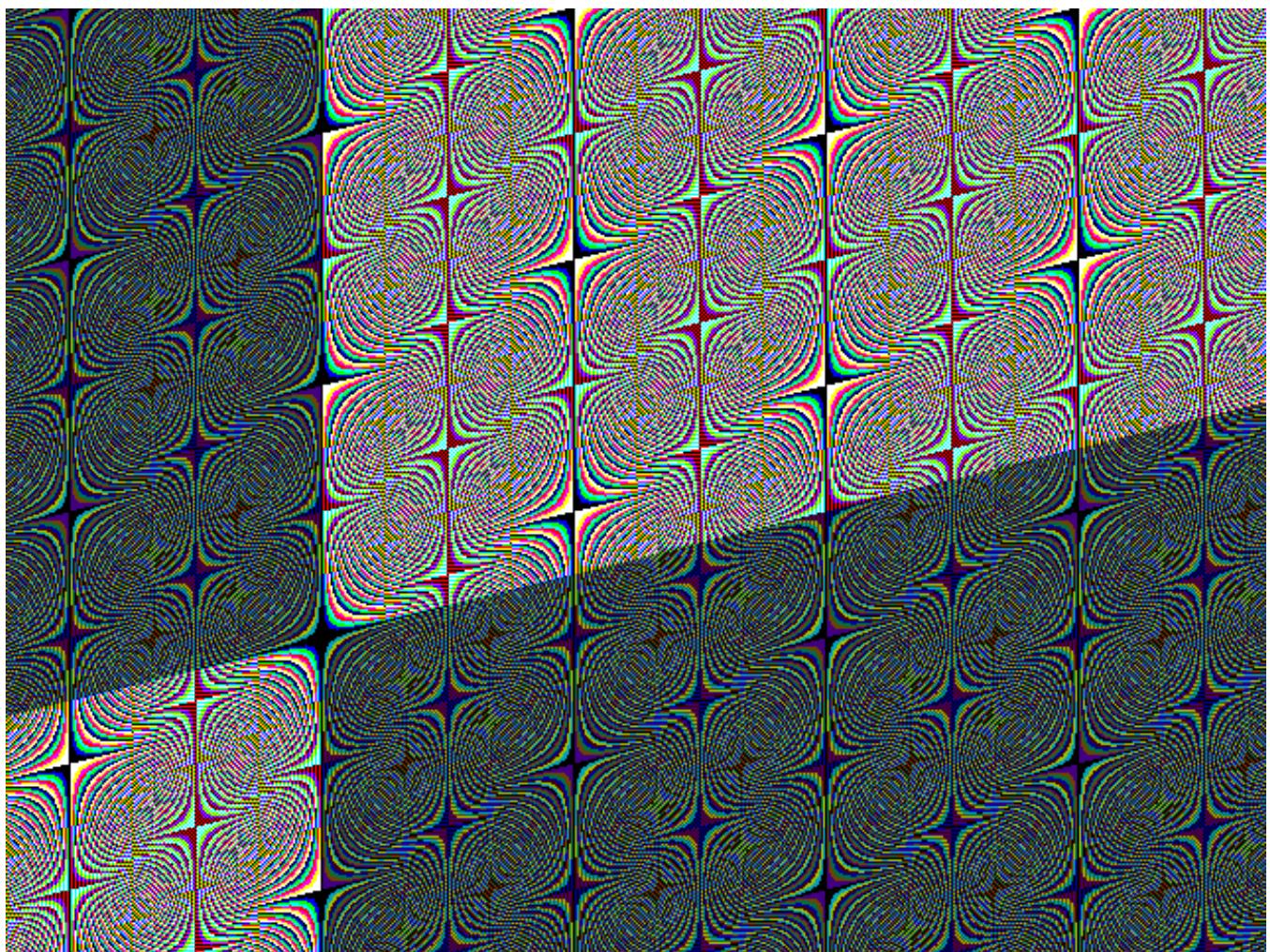


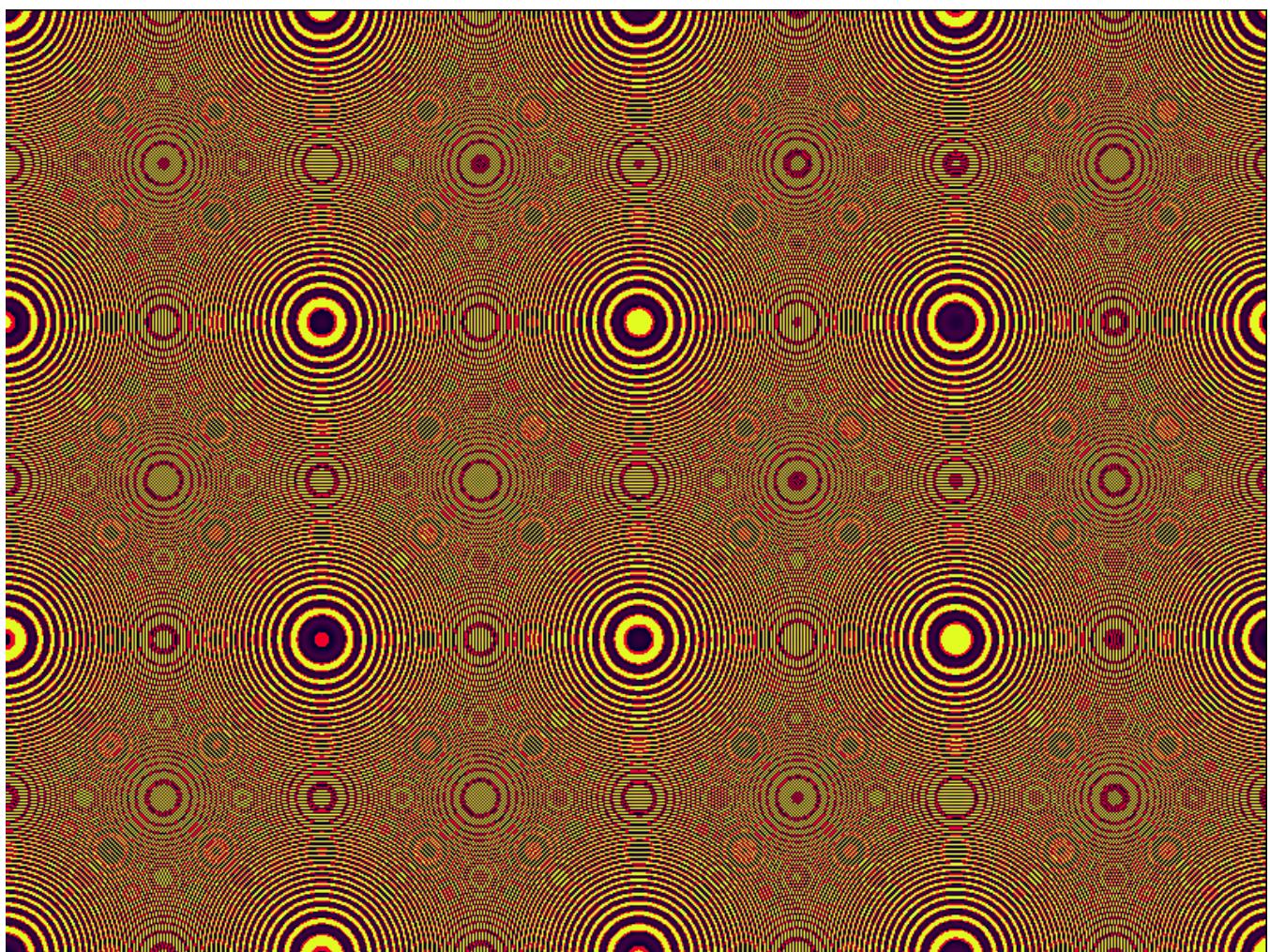
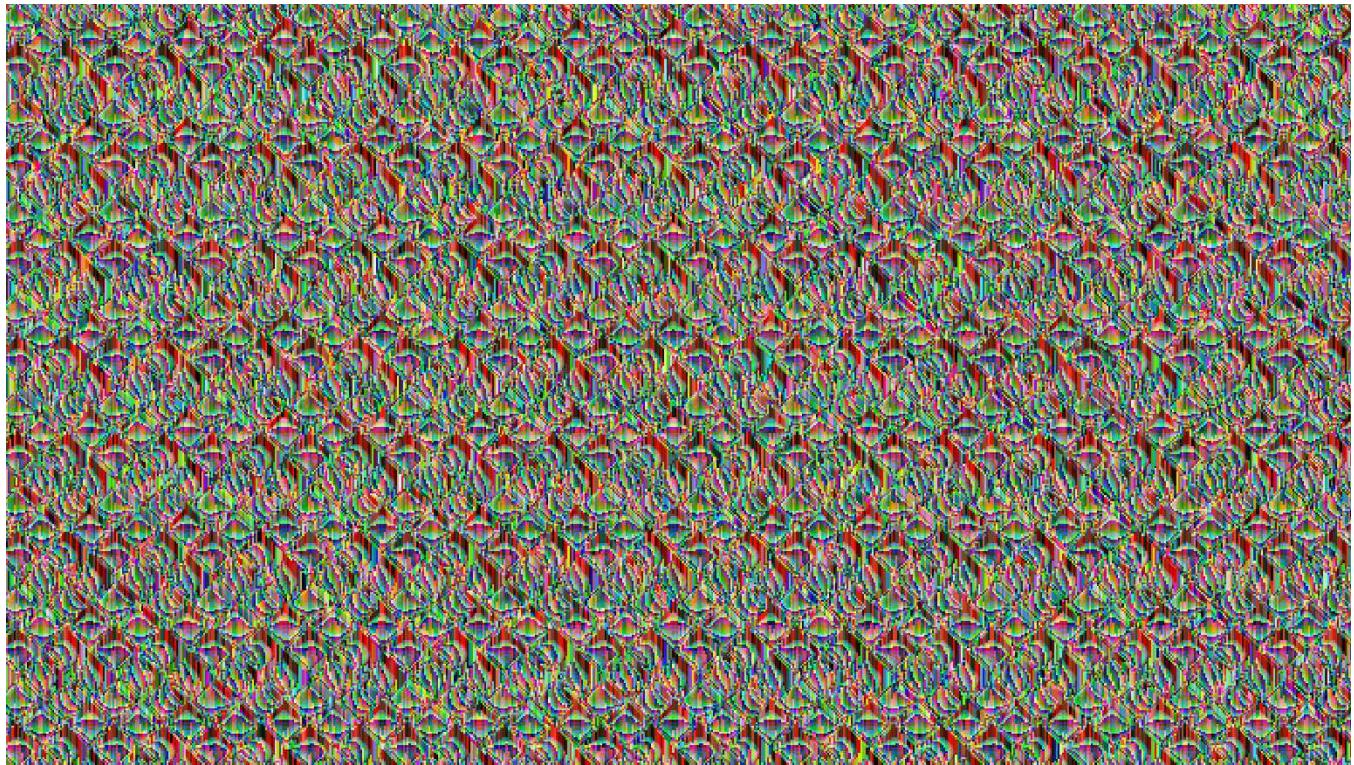


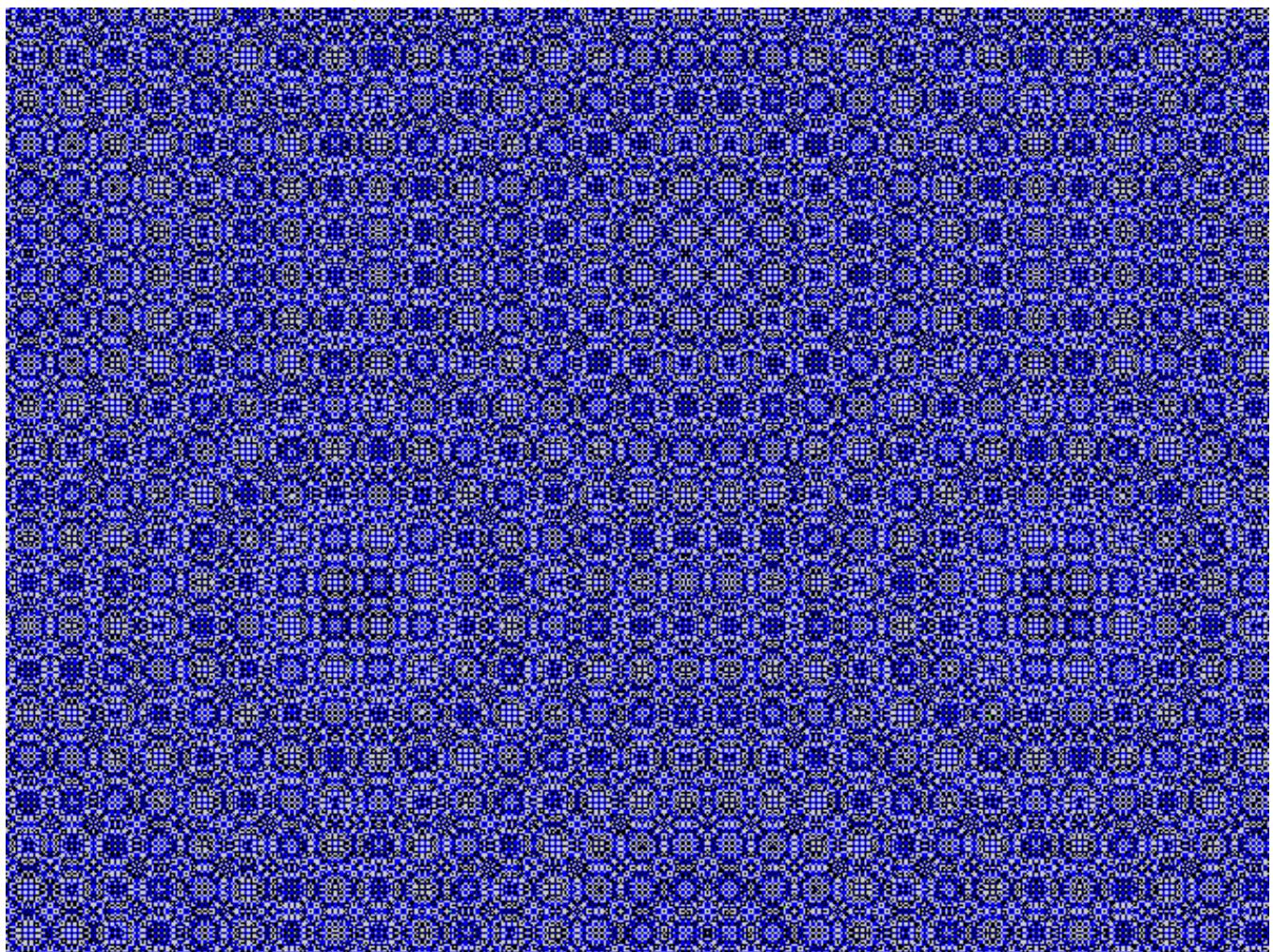
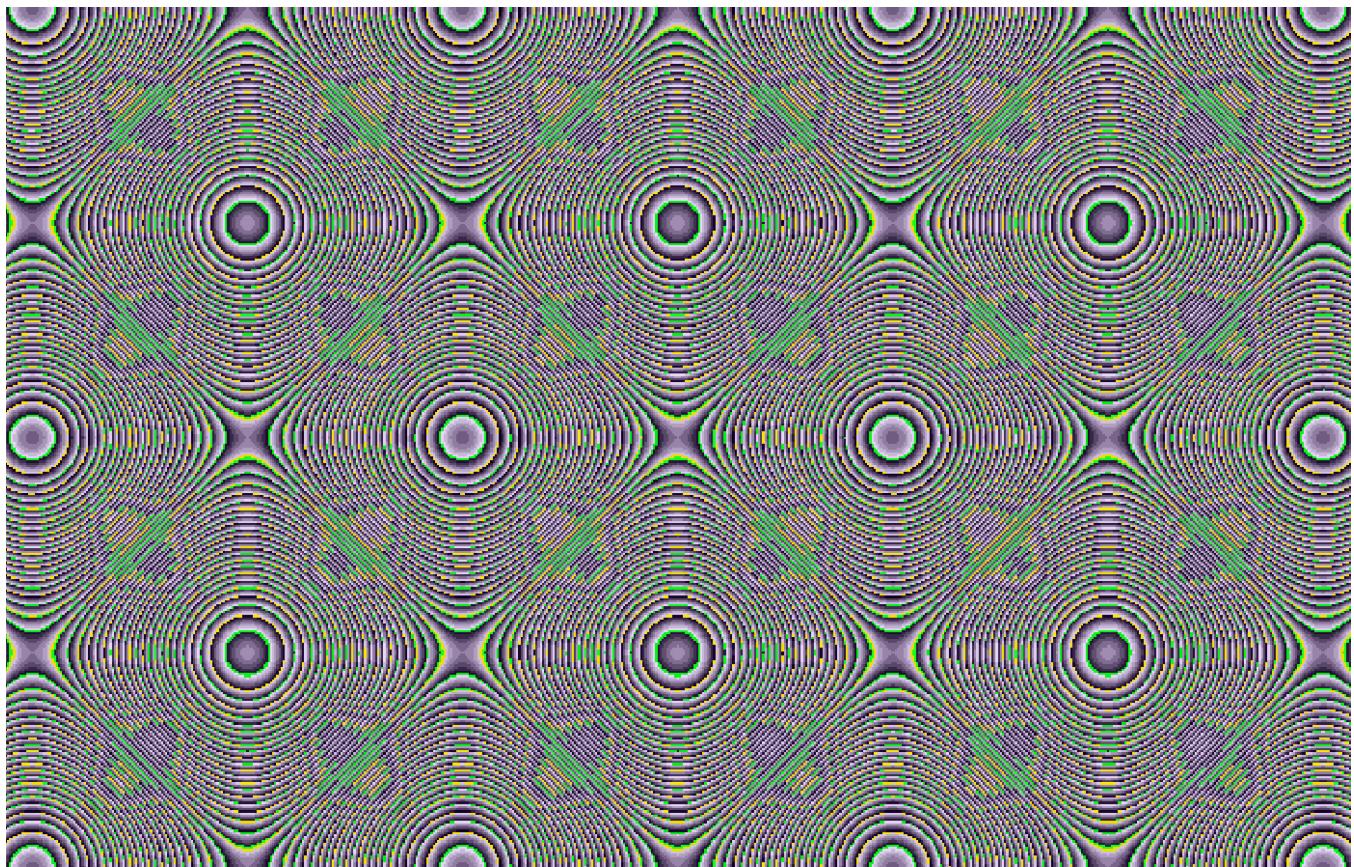


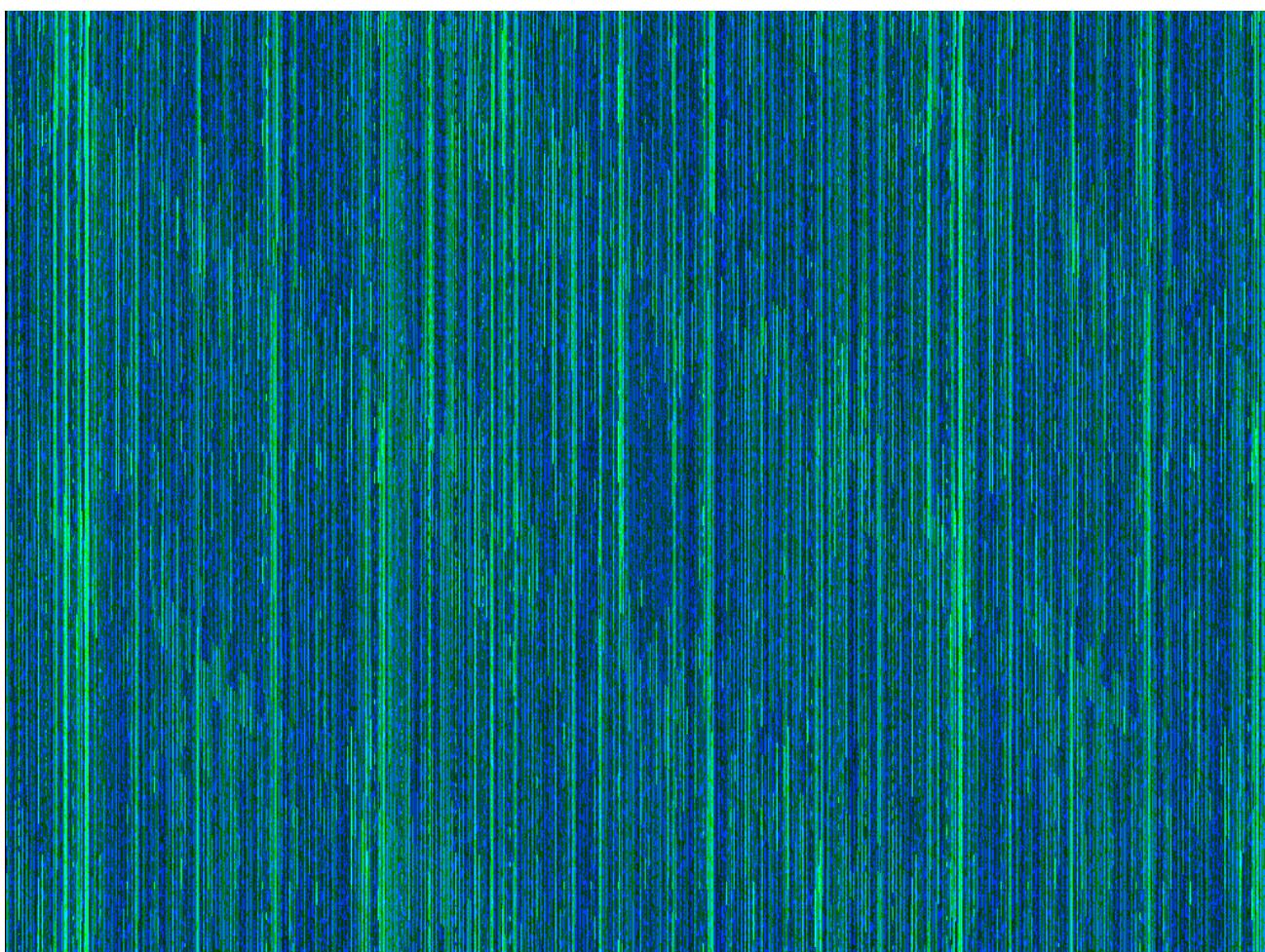
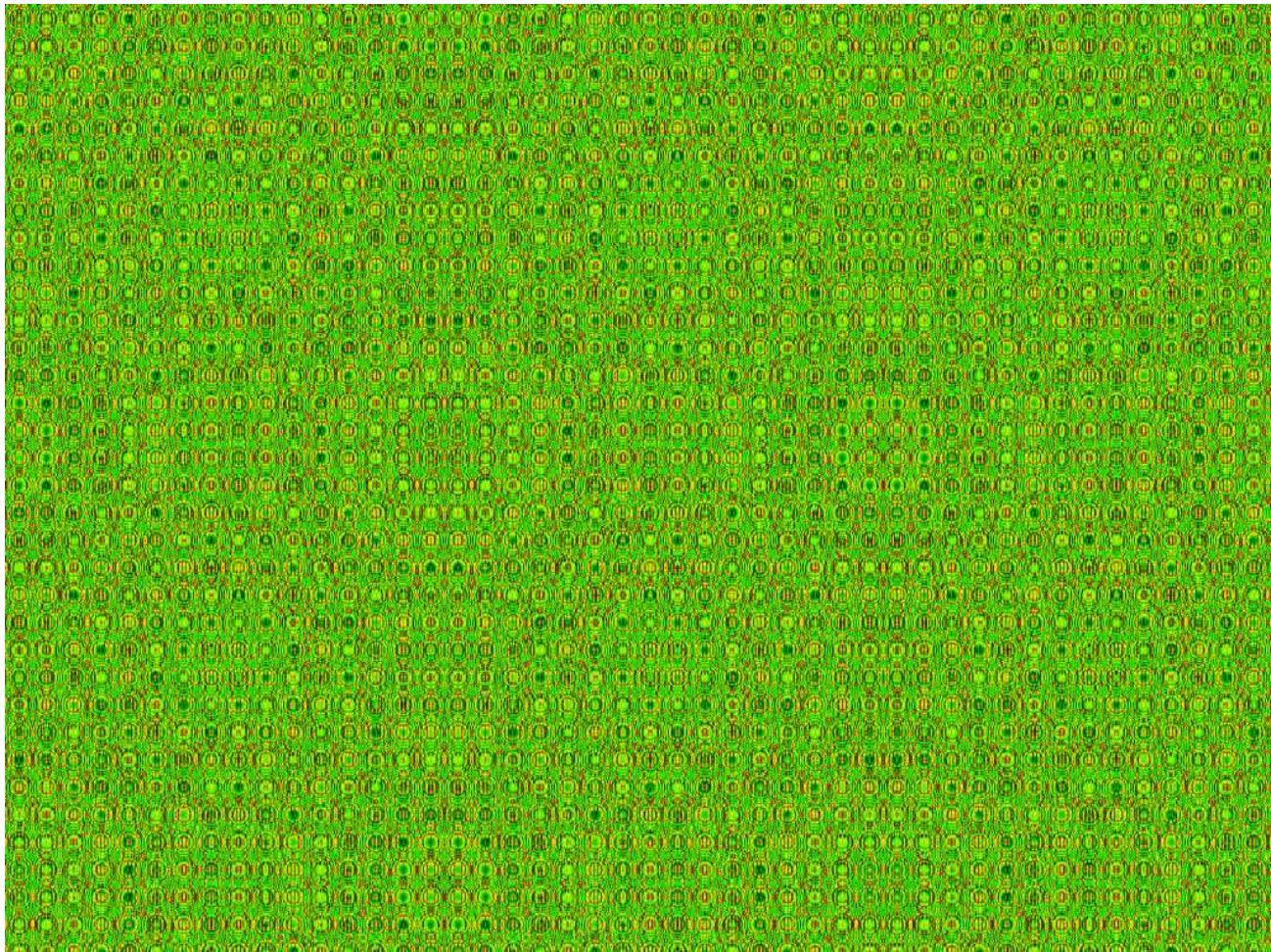


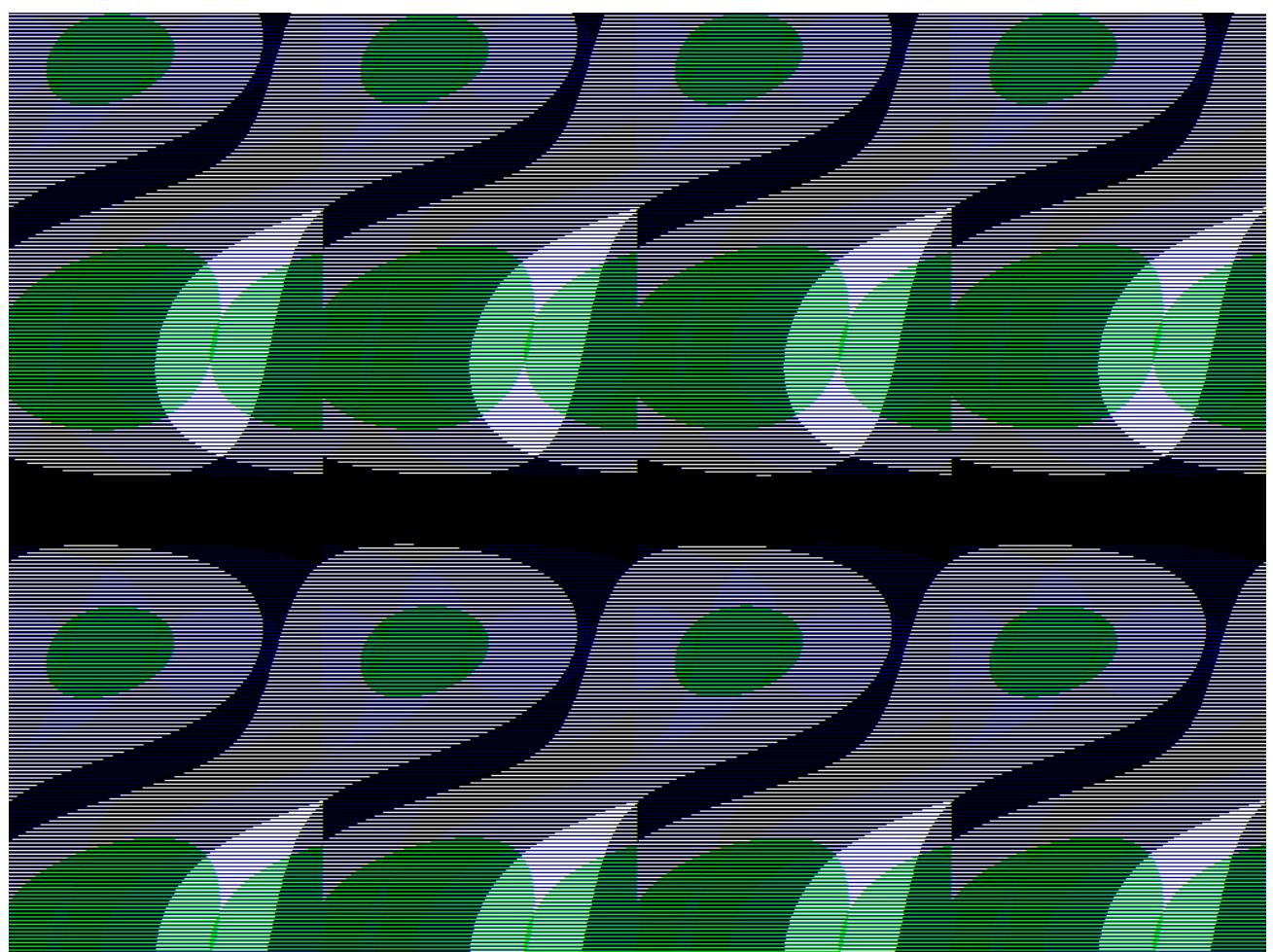
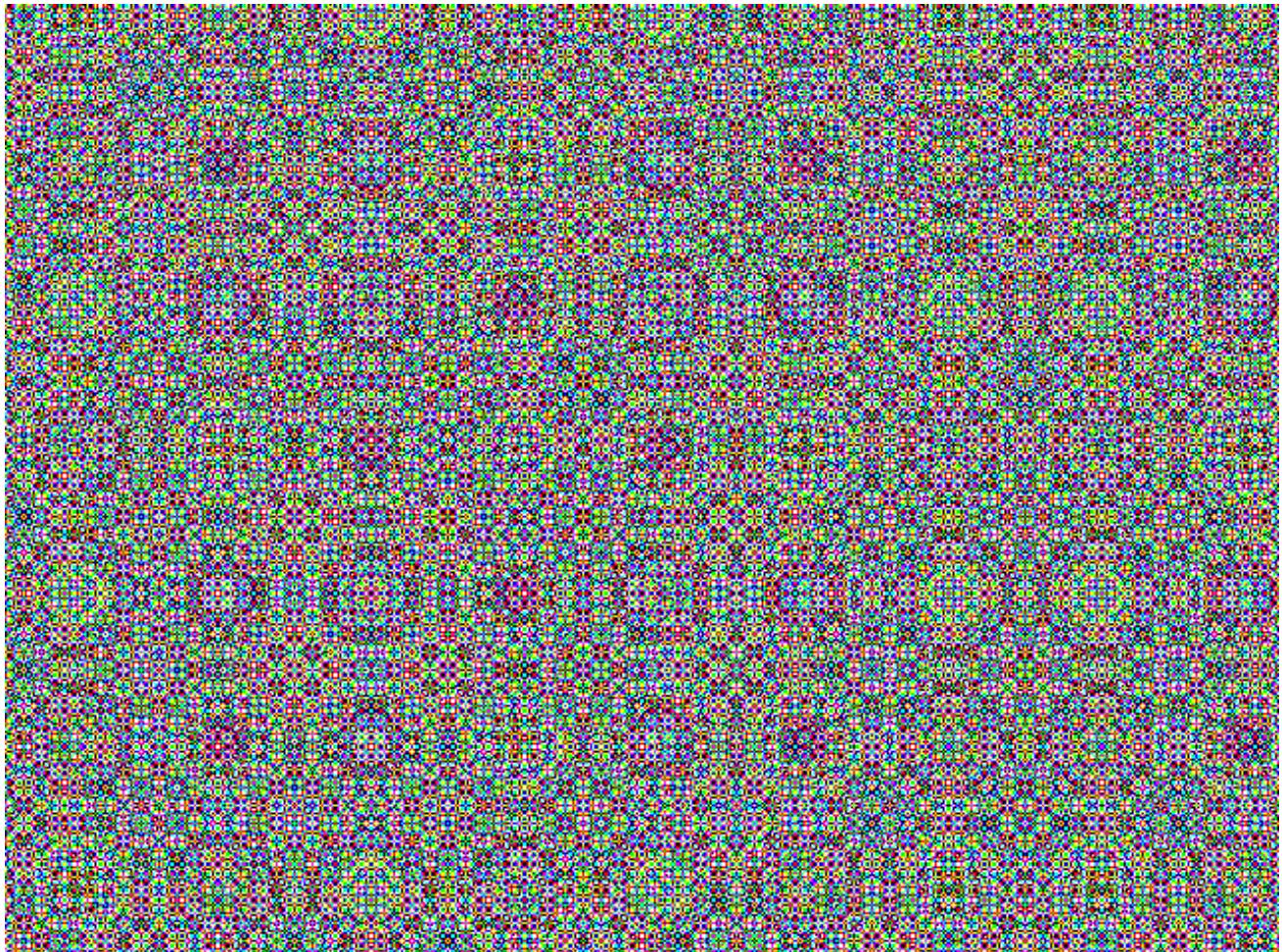


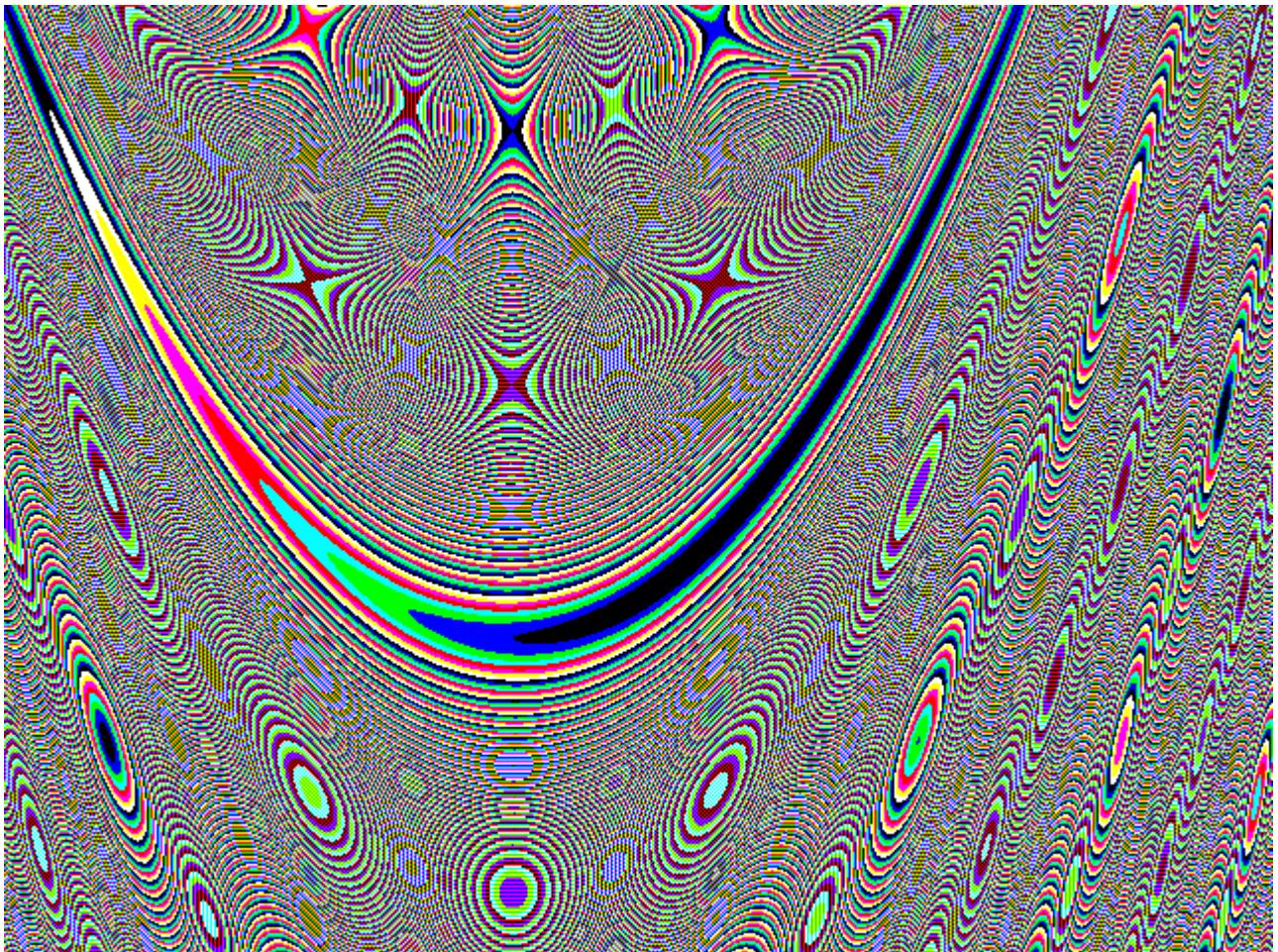










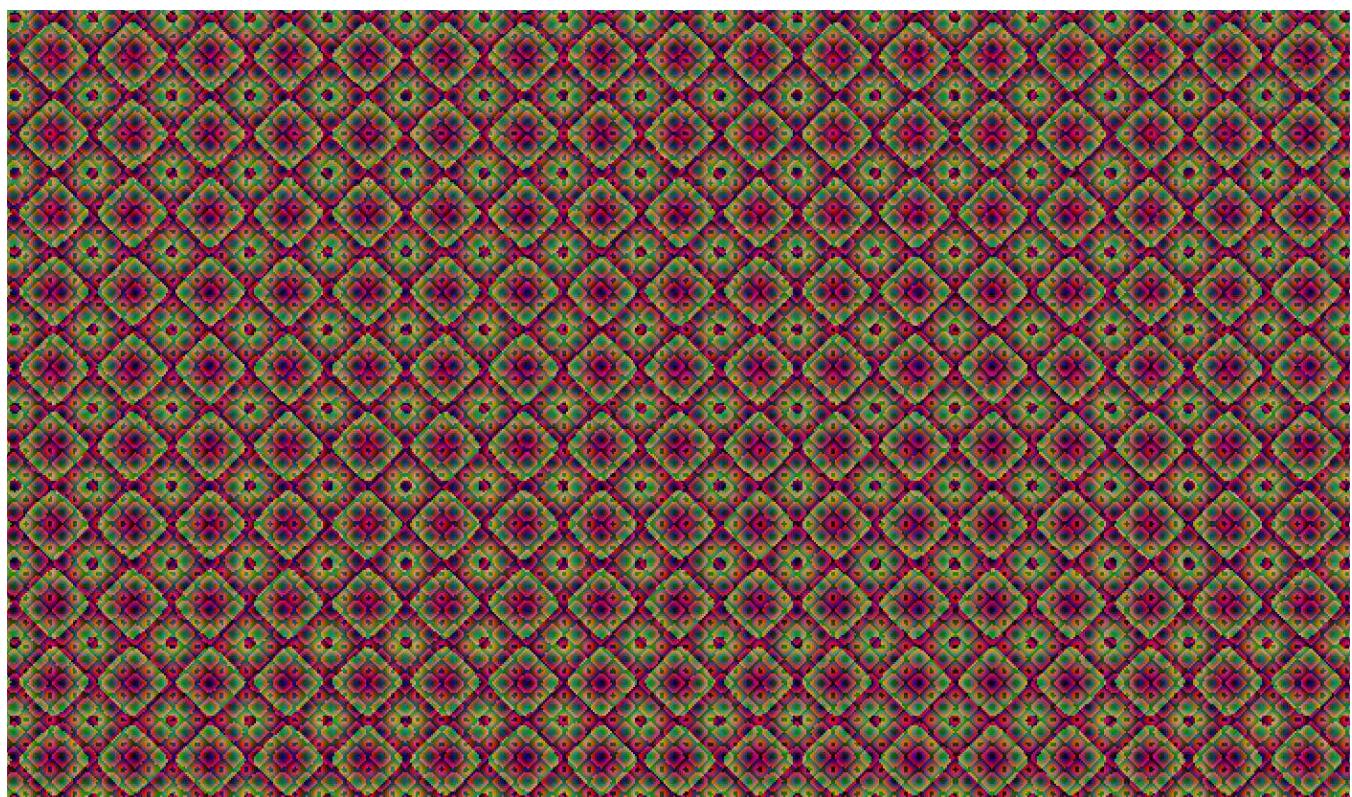
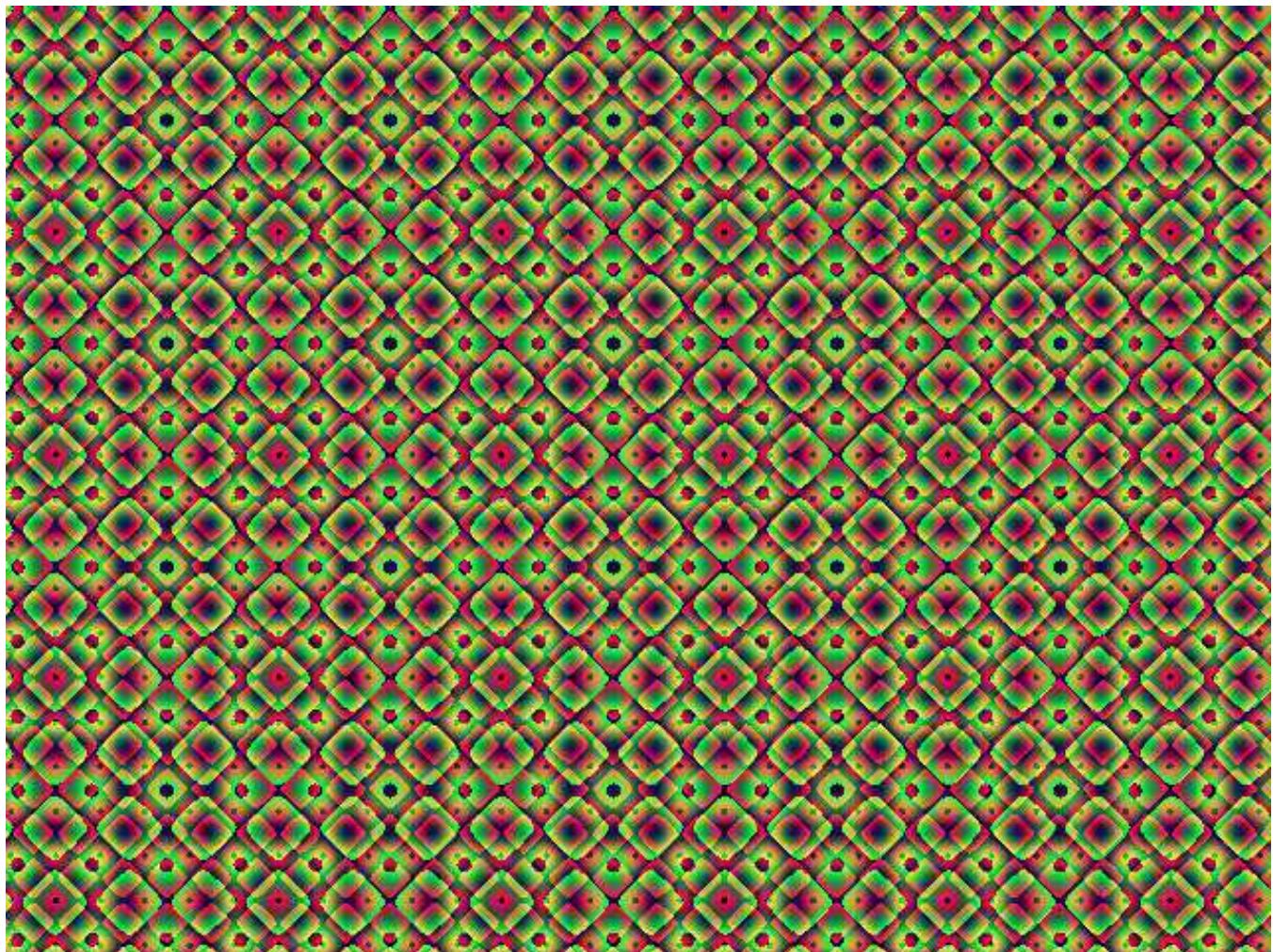


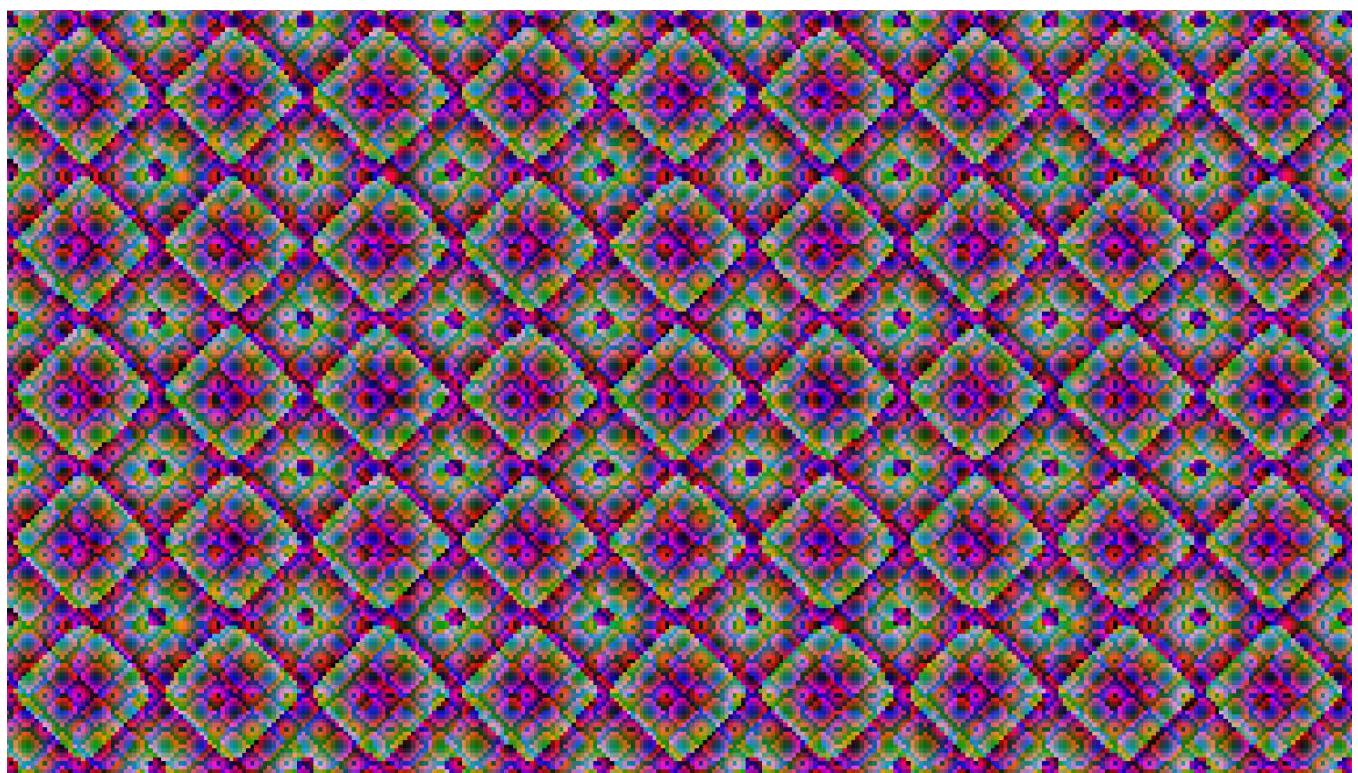
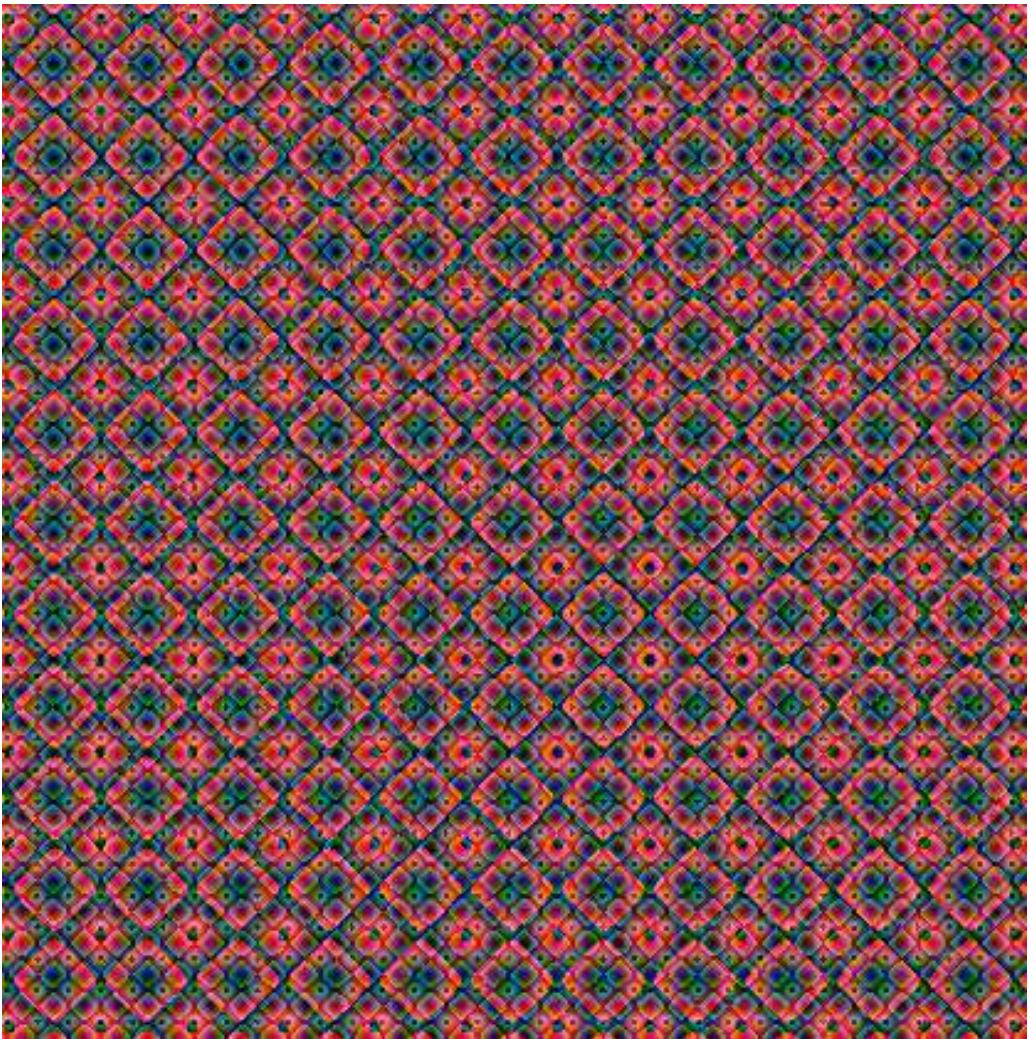
Per ottenere l'immagine riportata qui sopra, è stata ripresa la famosa *Rosenbrock's "banana" function* (1960), che spesso fu usata come funzione di test per algoritmi di minimizzazione:

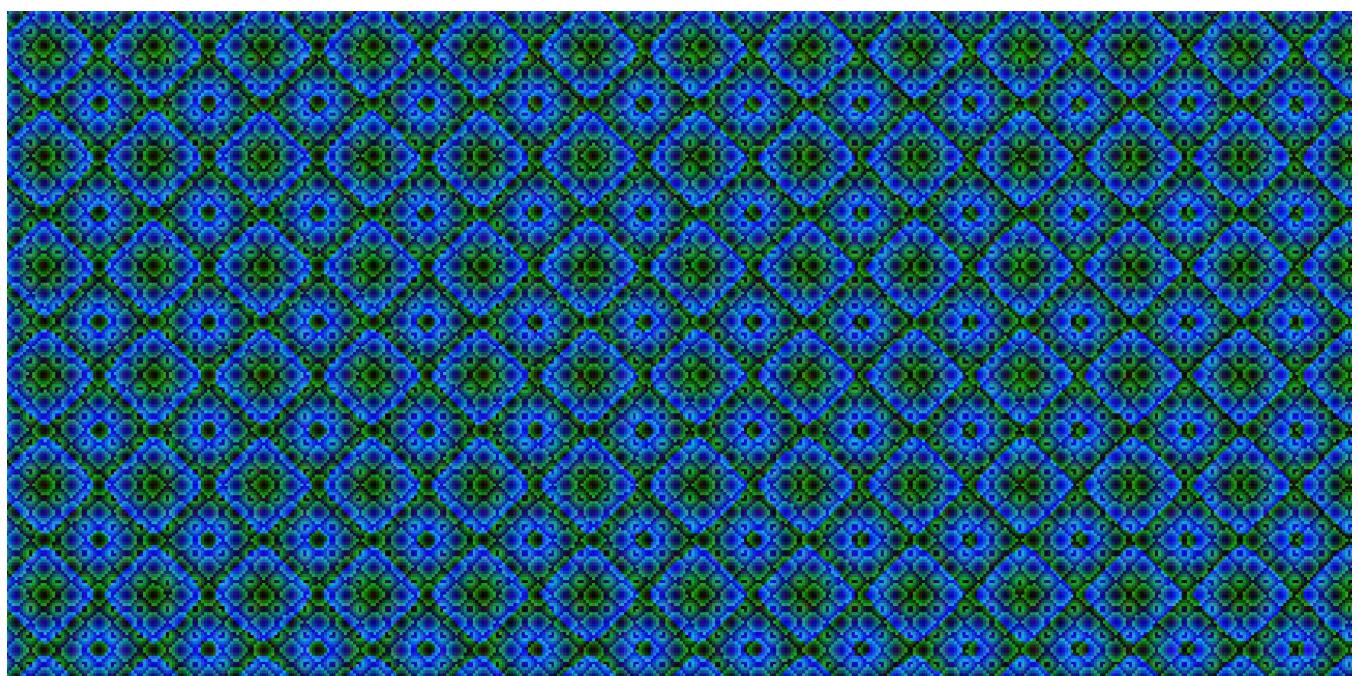
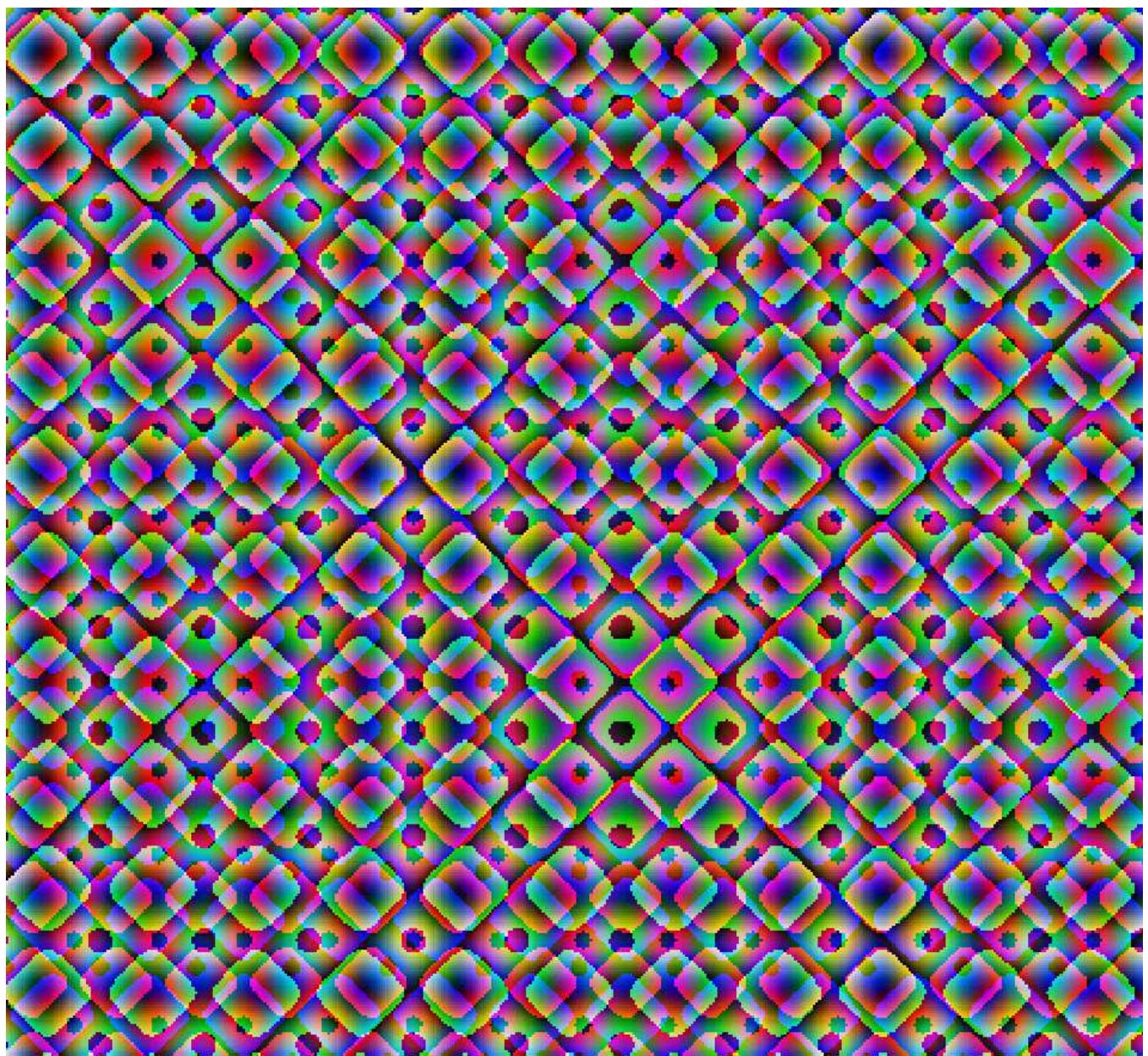
$$F(x, y) = 100 \cdot (y - x^2)^2 + (1 - x)^2$$

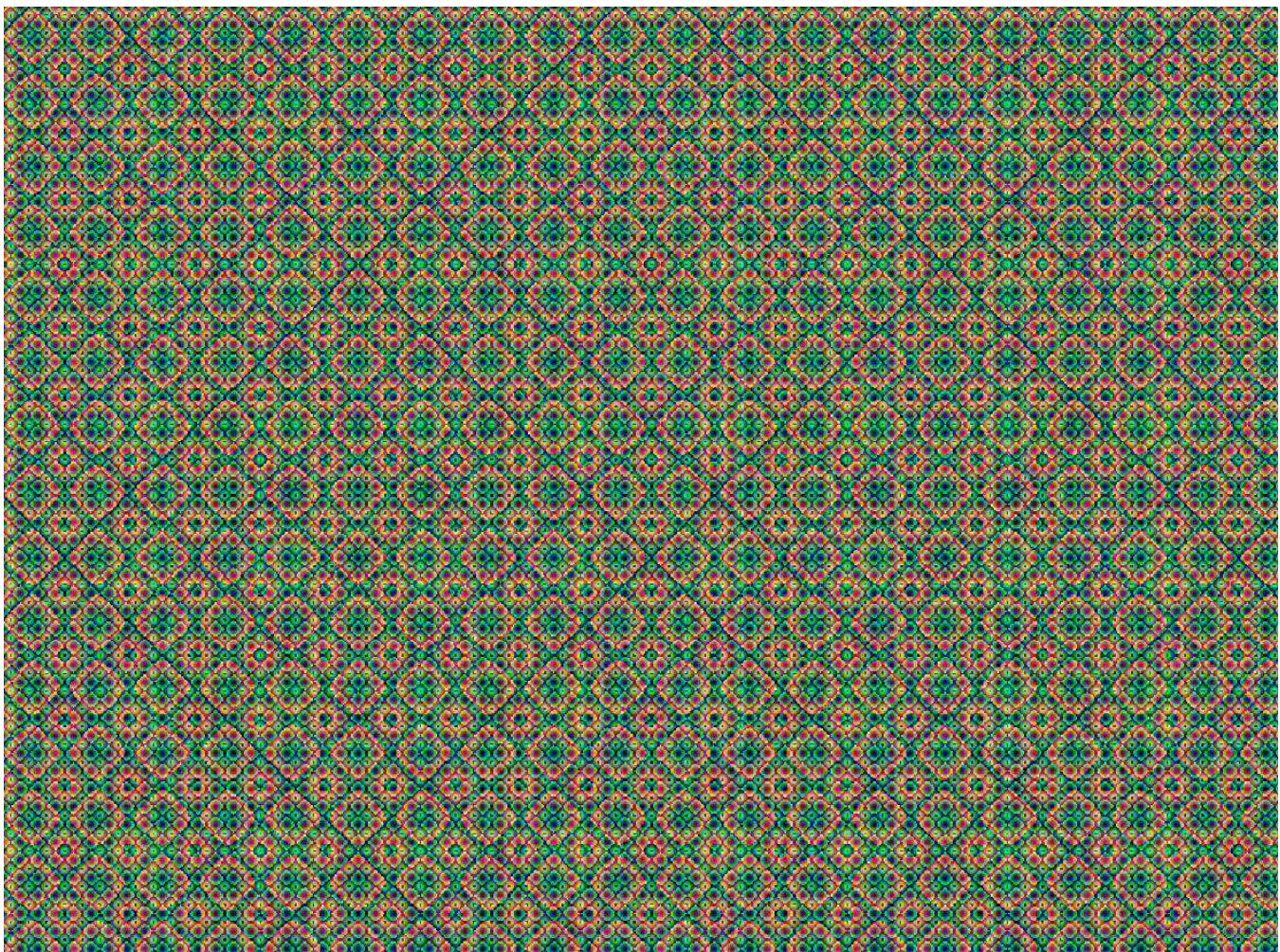
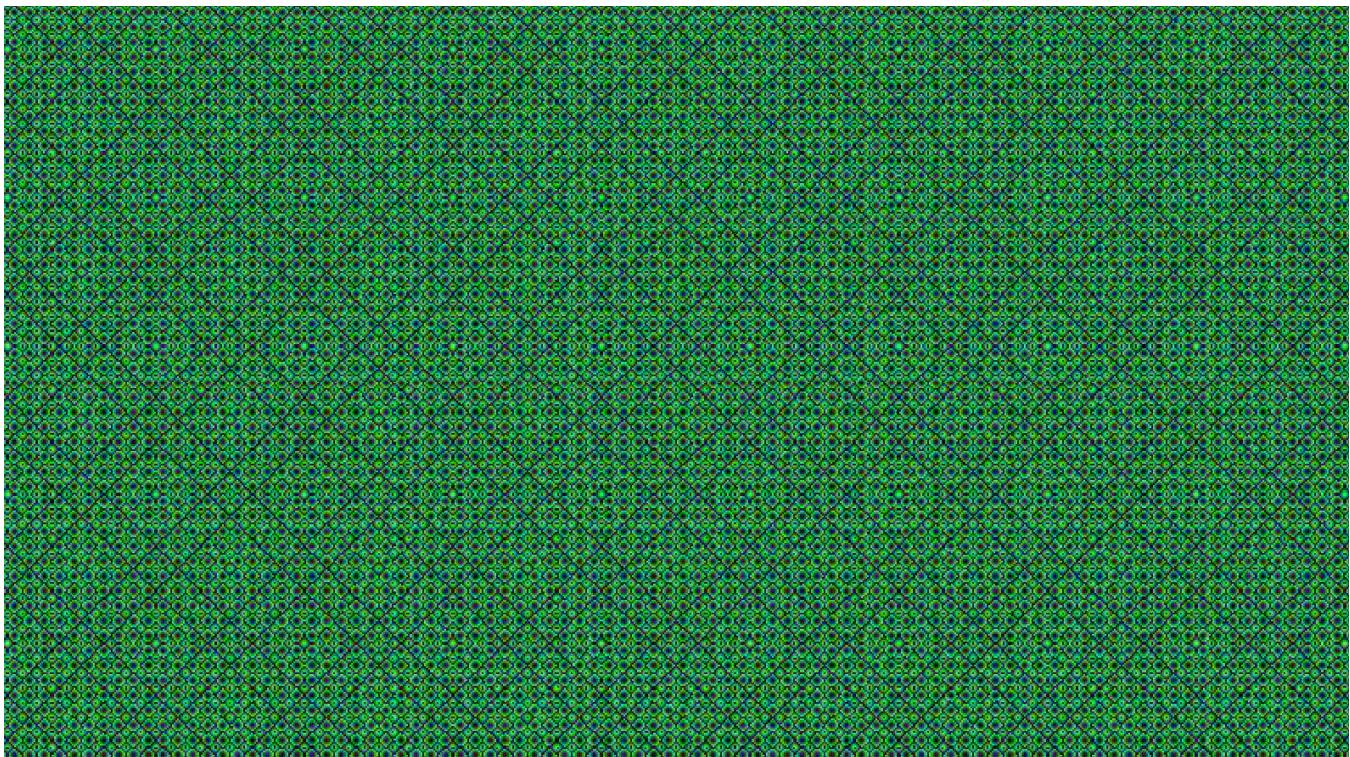
la quale presenta un minimo (assoluto) nel punto  $(1, 1)$ . La porzione rettangolare del piano cartesiano qui considerata ha vertice in basso a sinistra in  $(-2, -2)$ , larghezza 6 e lunghezza 5, non è stata usata una scala monometrica (infatti la matrice di pixel originale è  $480 \times 640$ ), la tavolozza consta di soli 8 colori; si notino le asimmetrie.

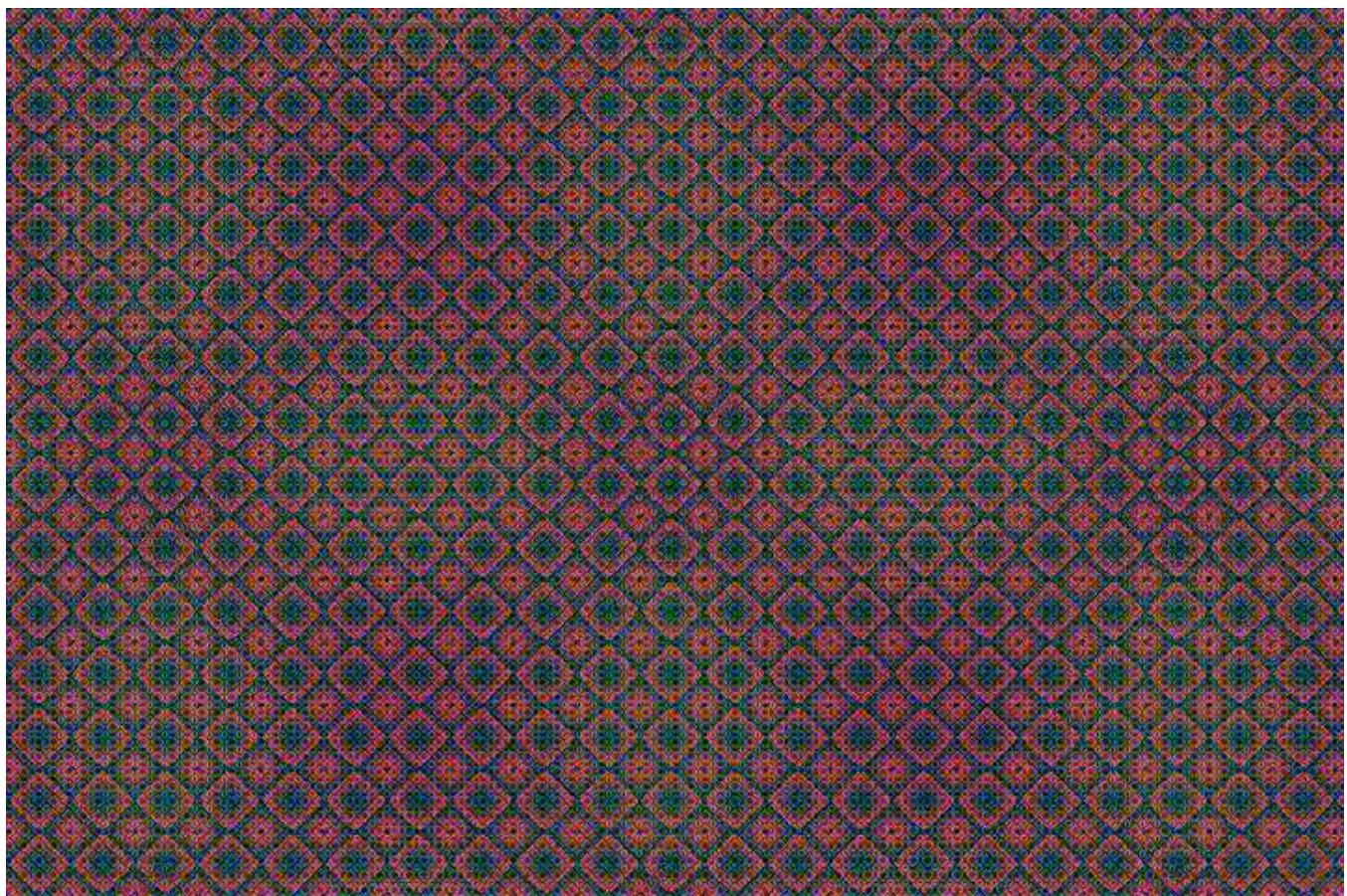
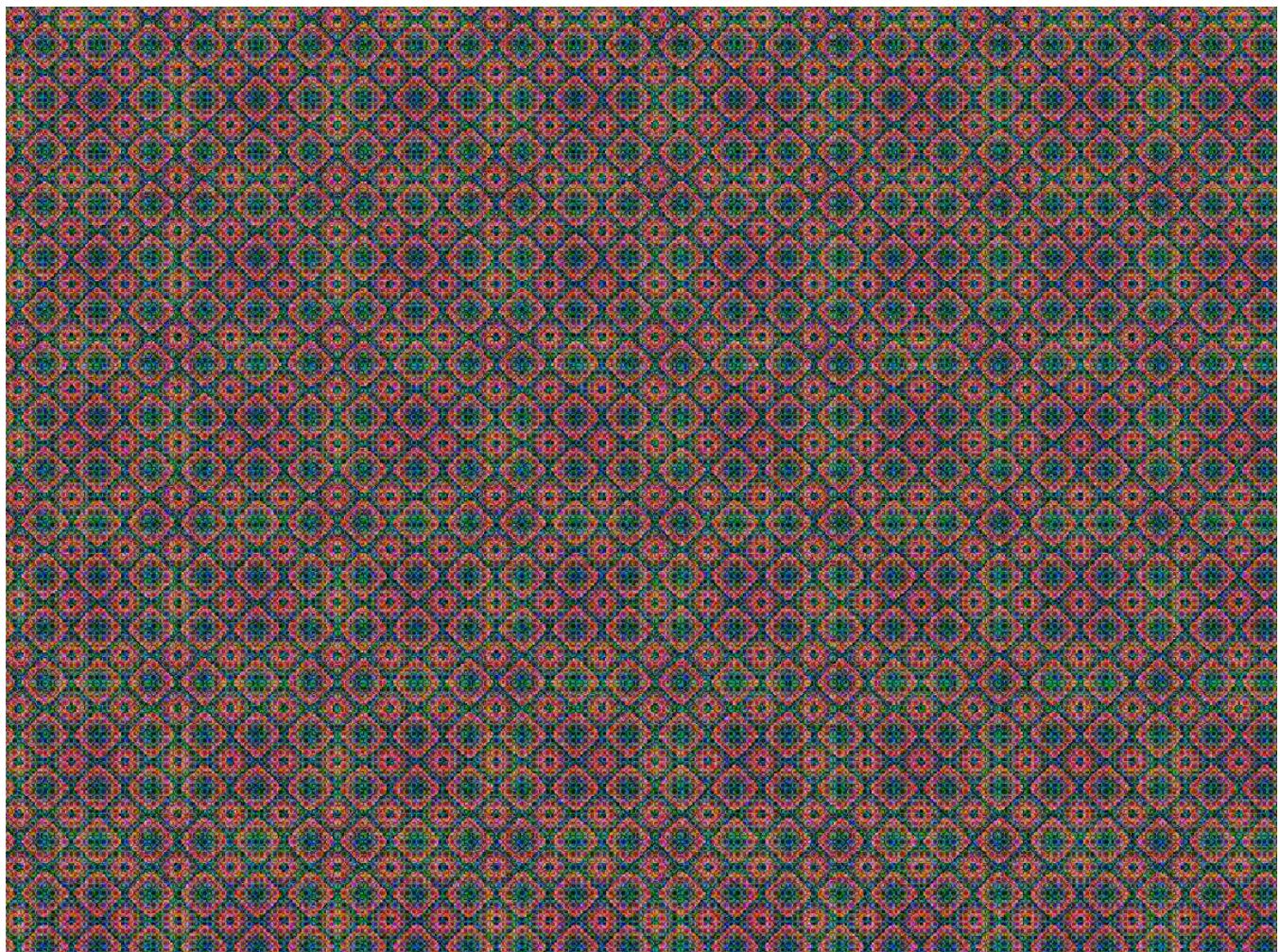
## **Sezione II – Uova Fabergé**

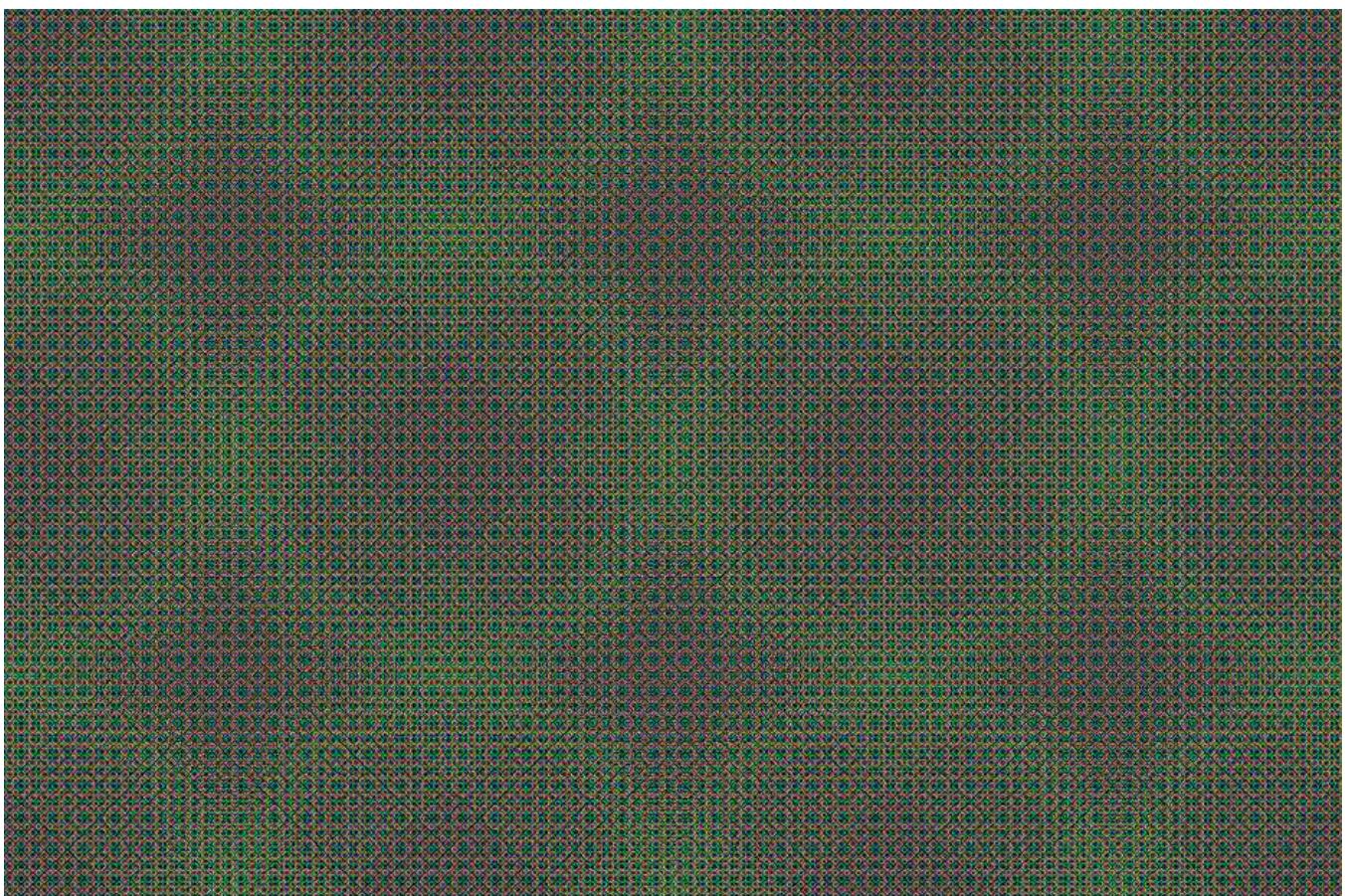
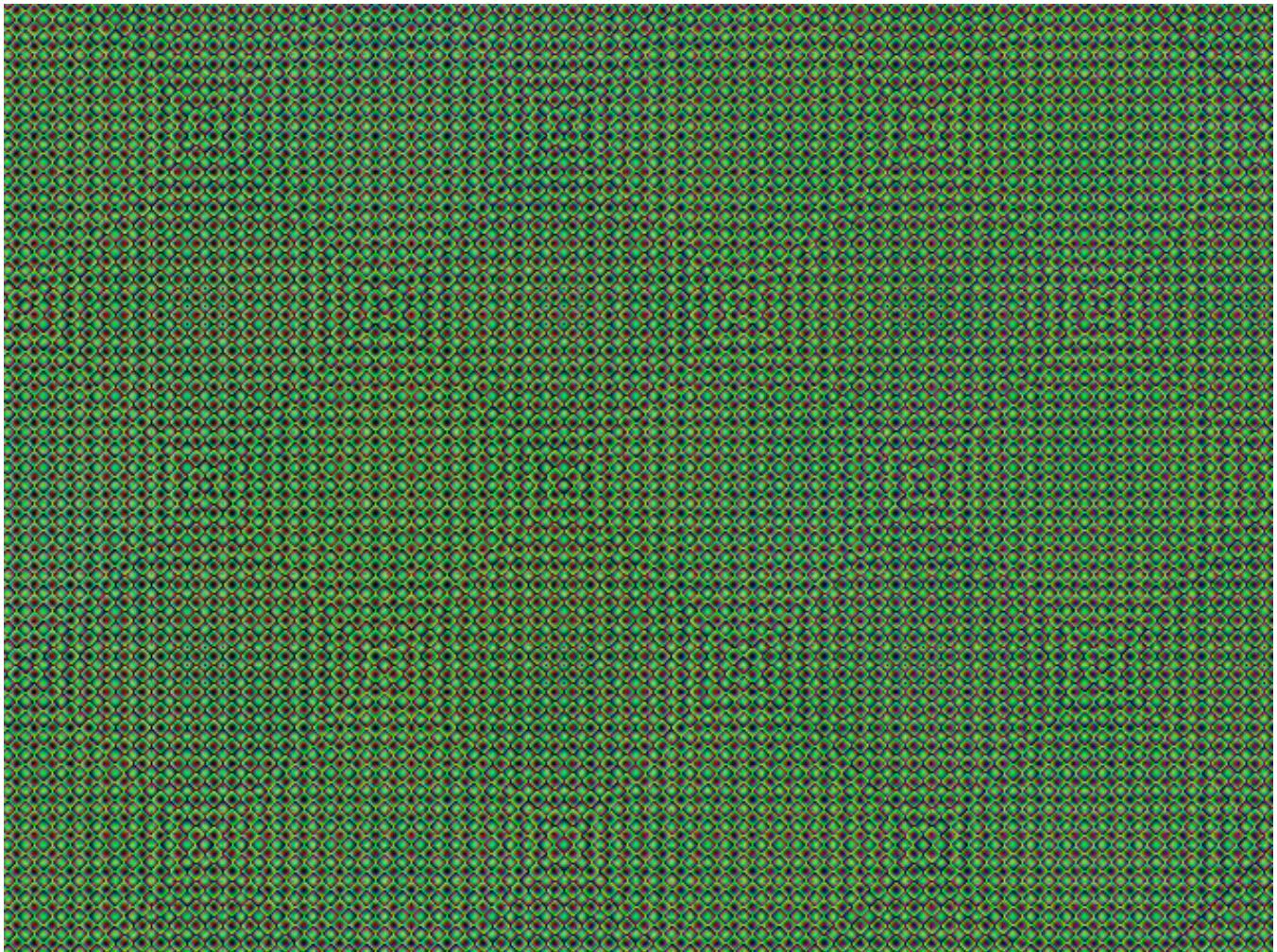


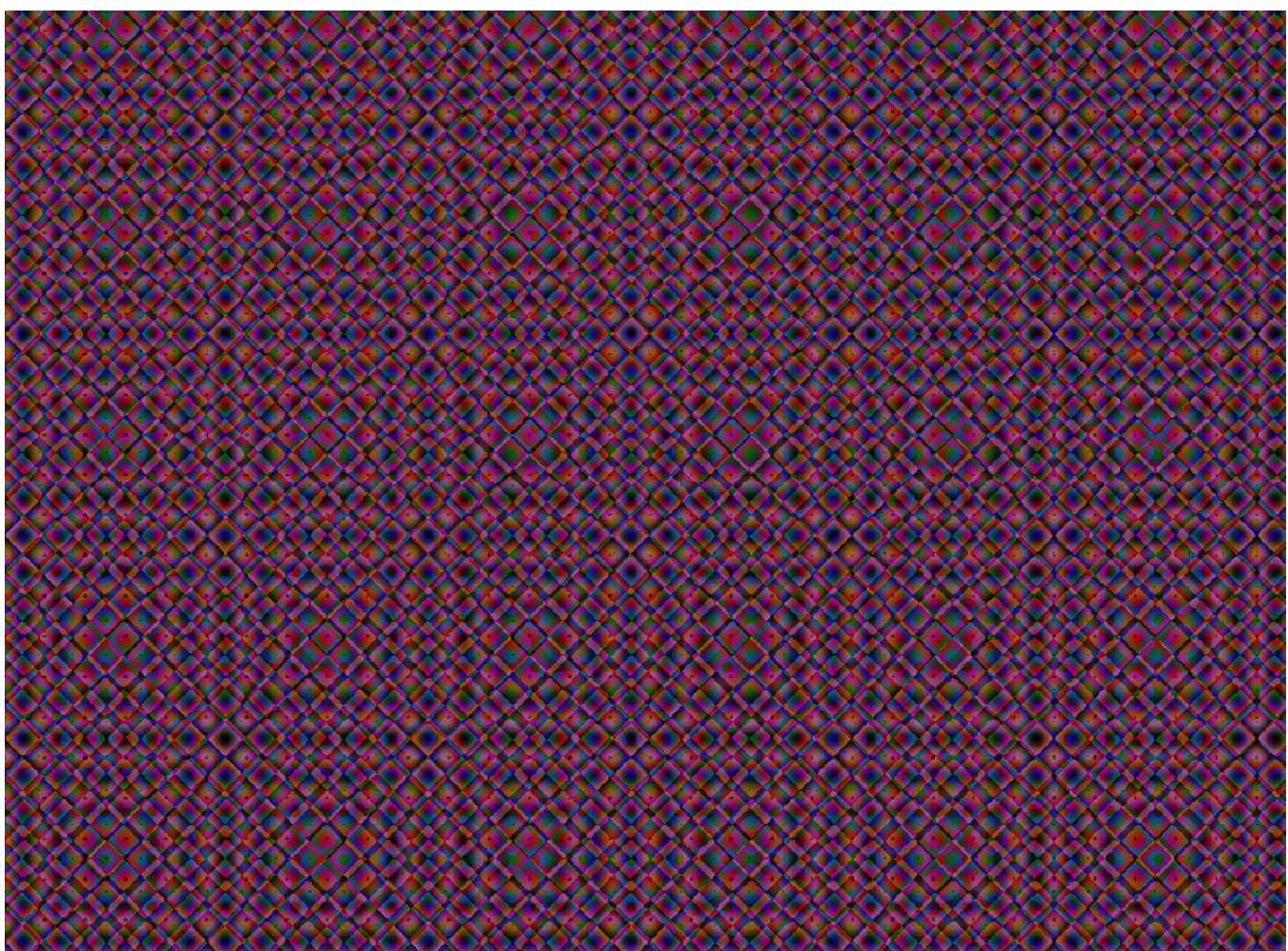
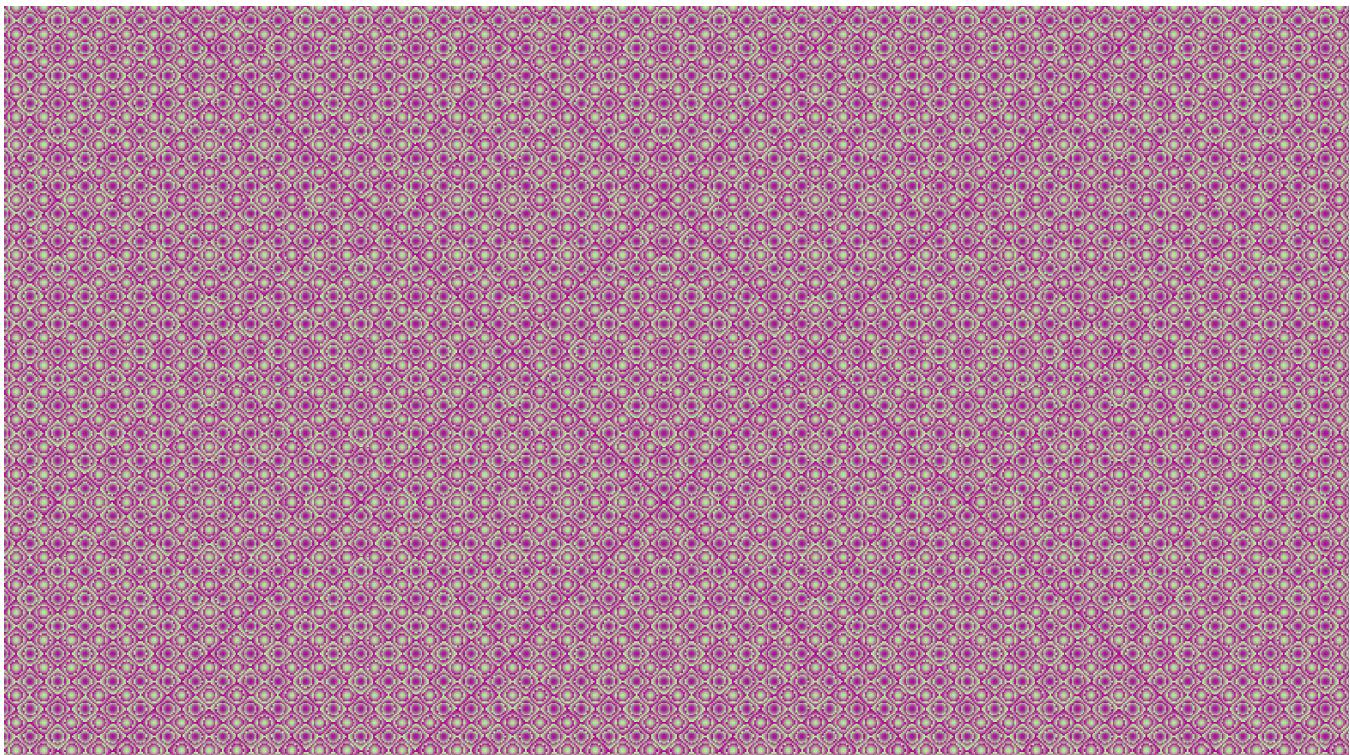


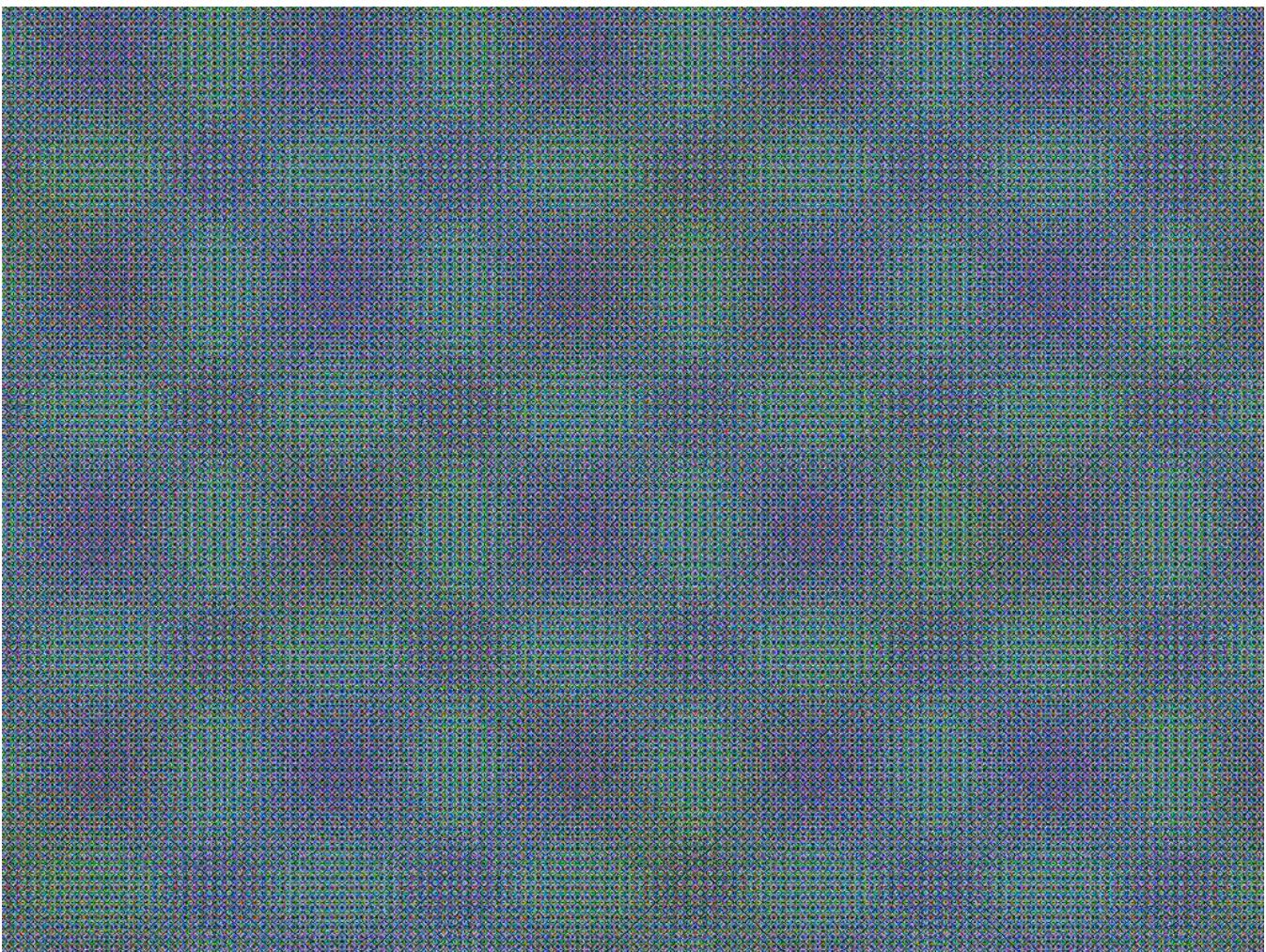
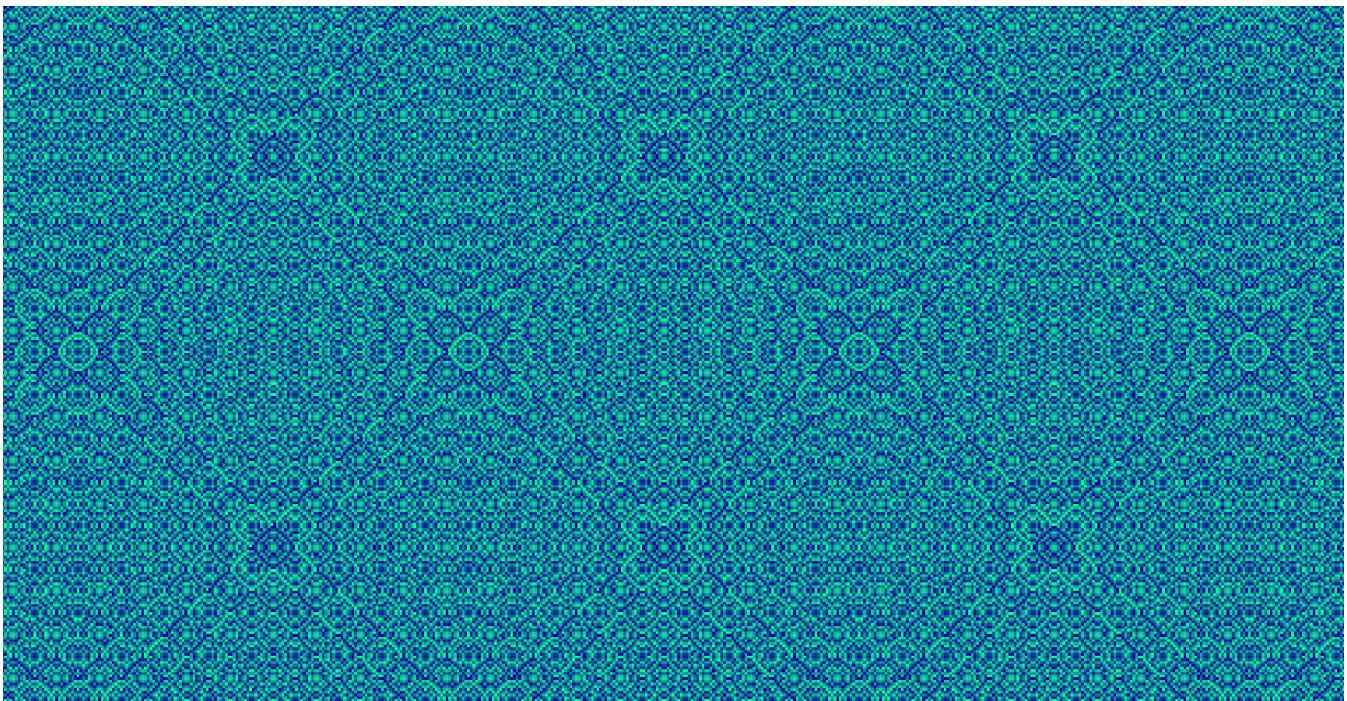


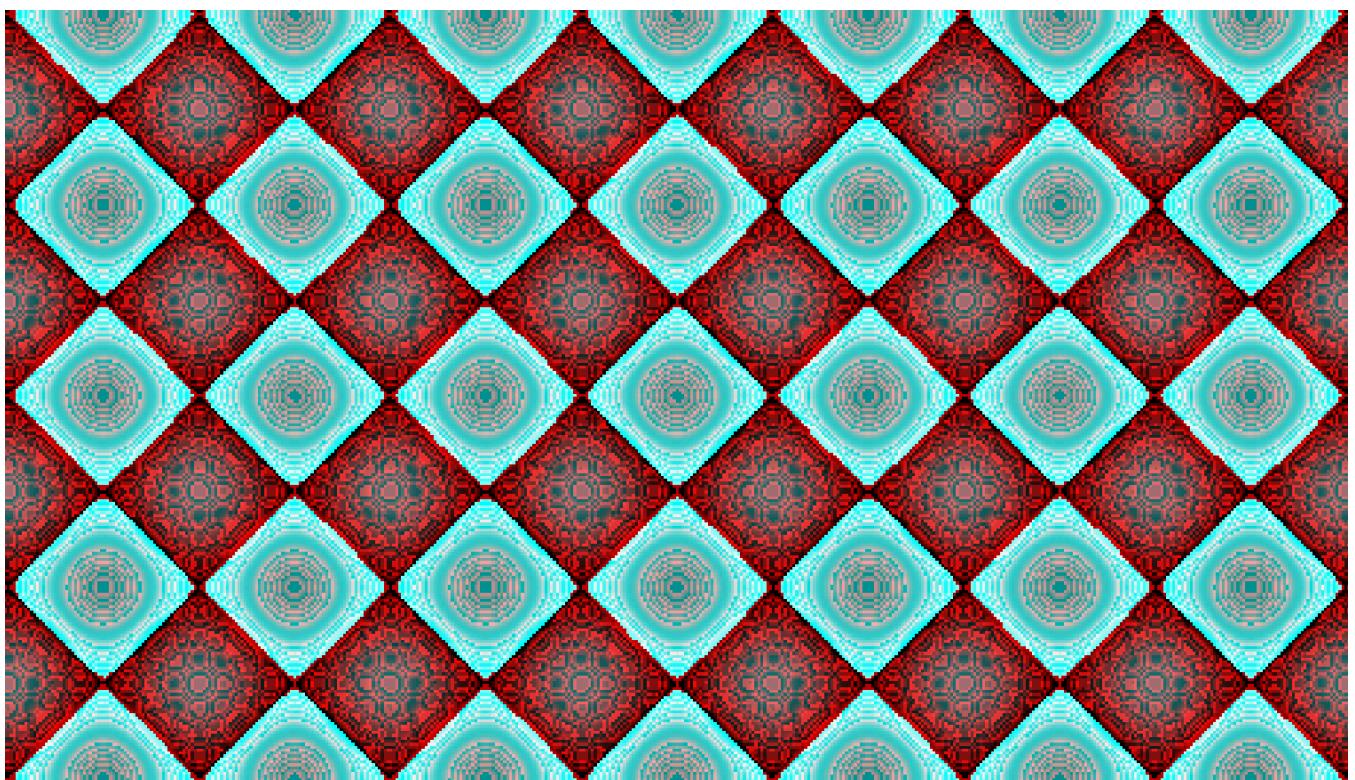
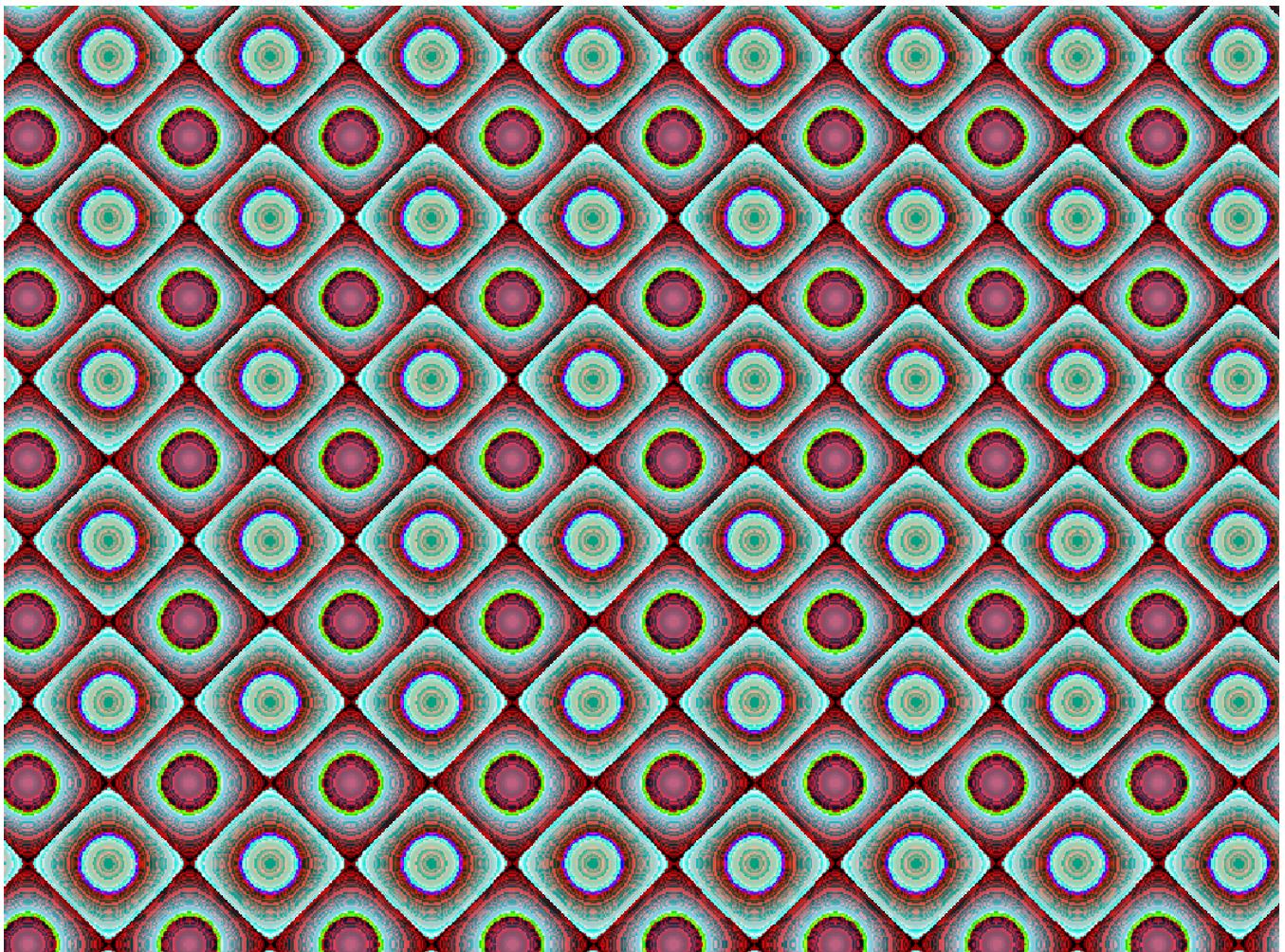


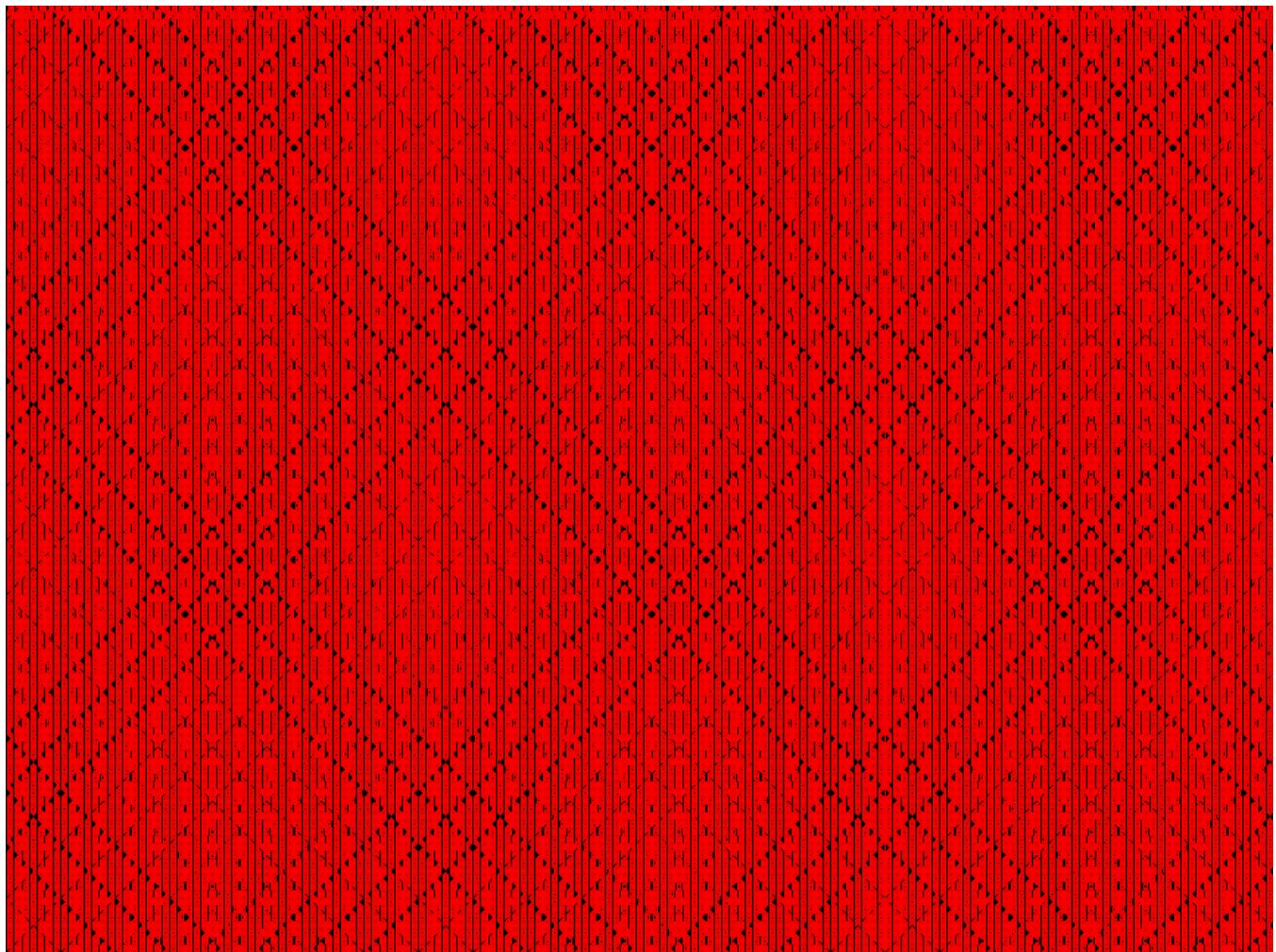
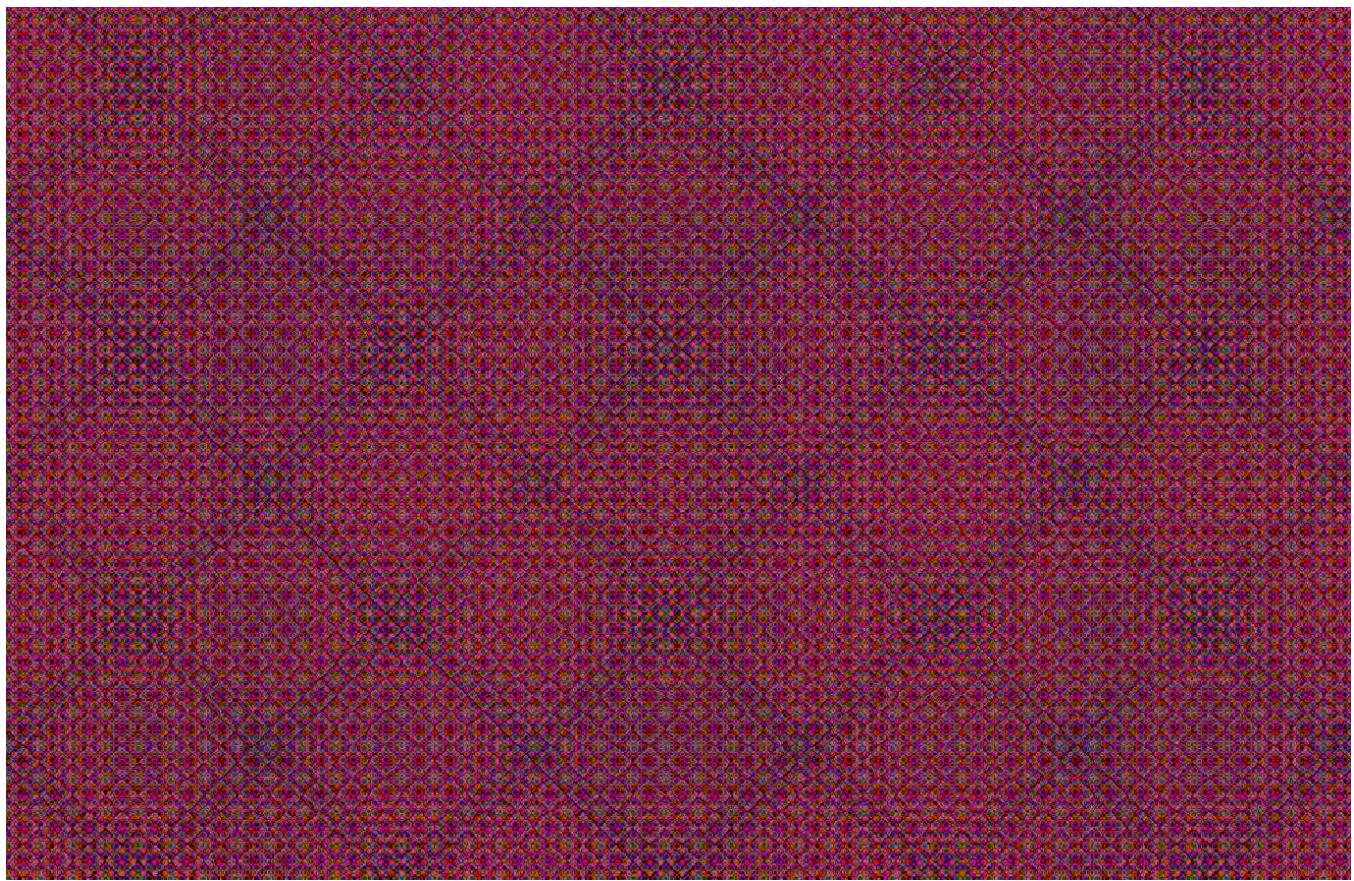


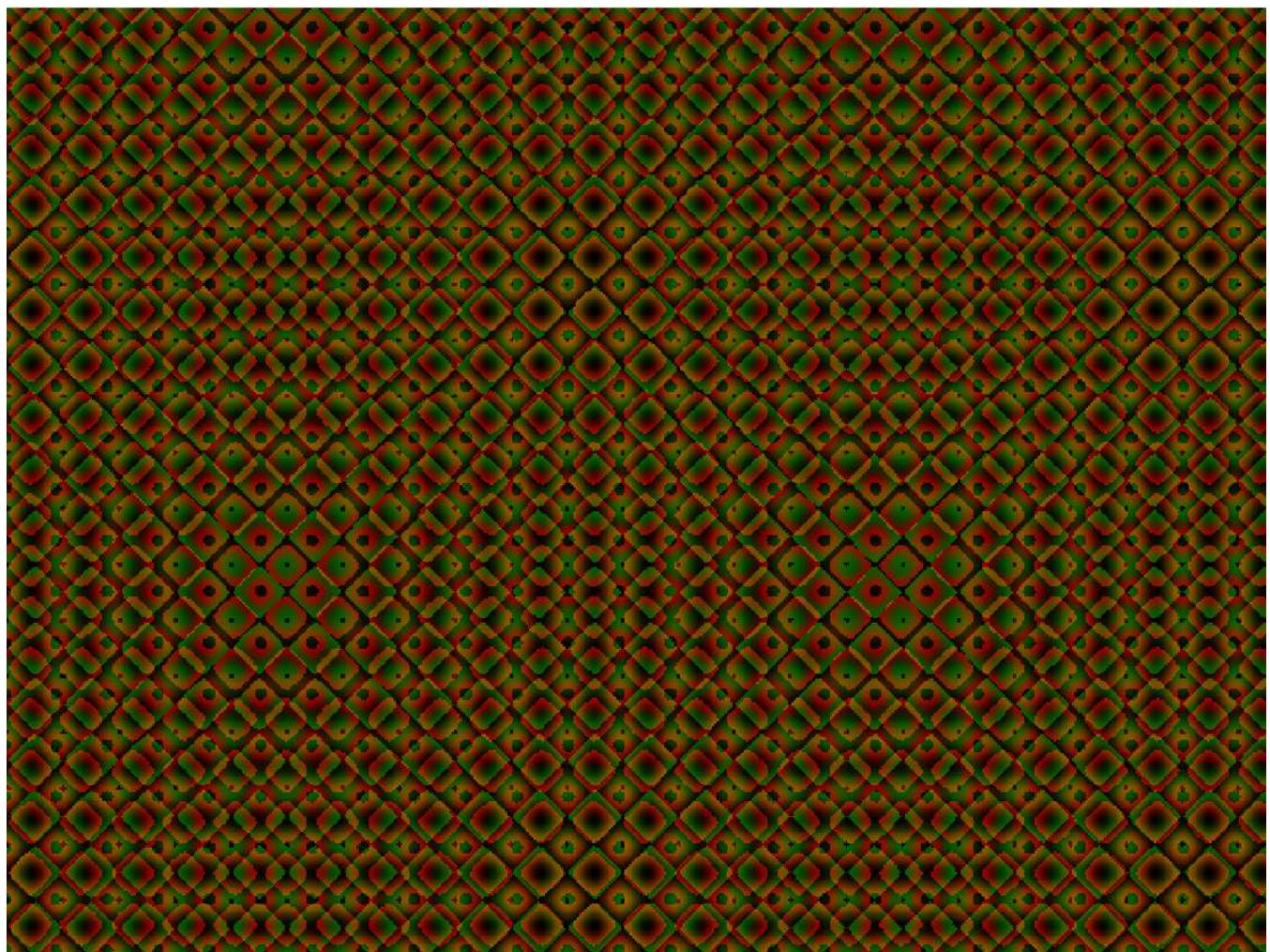
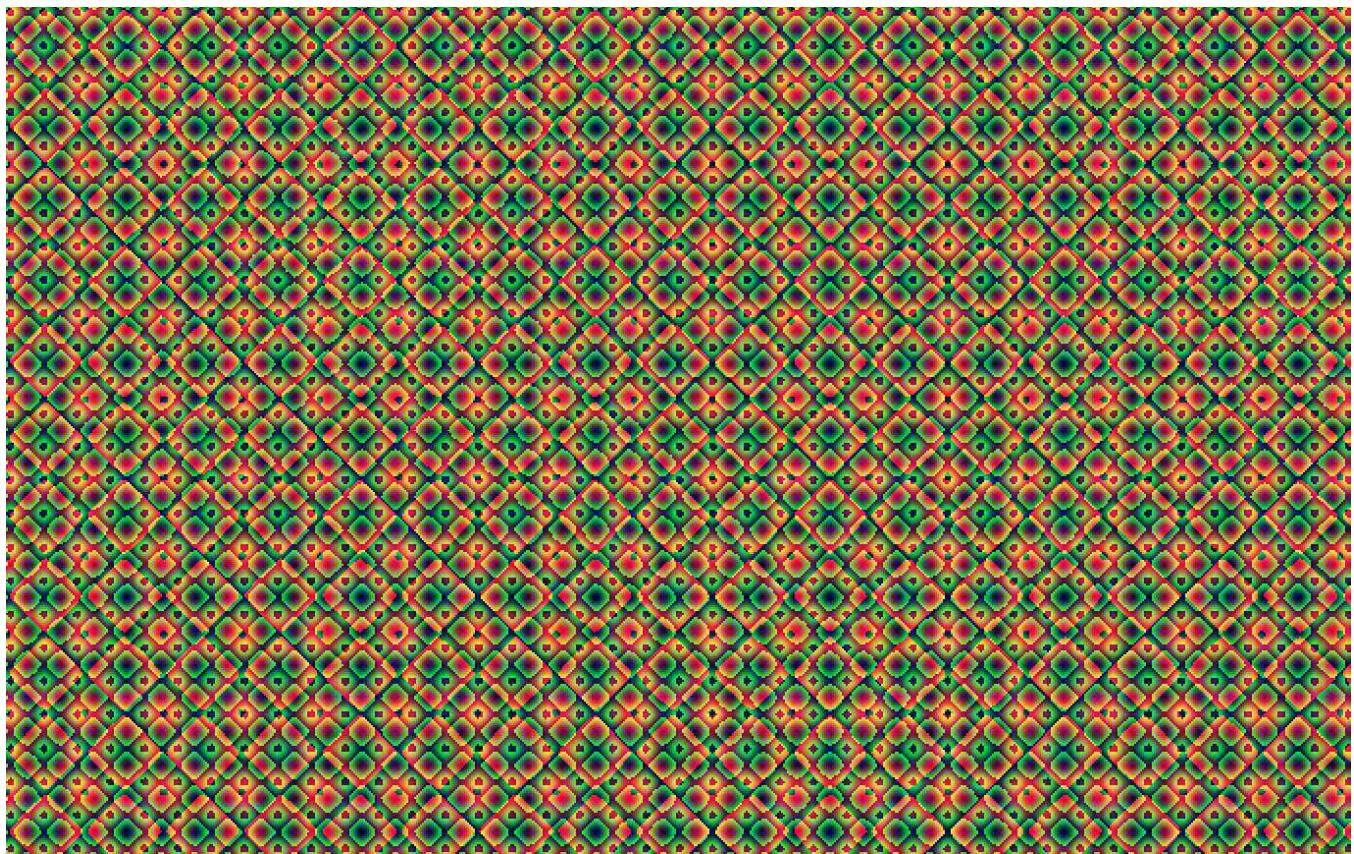


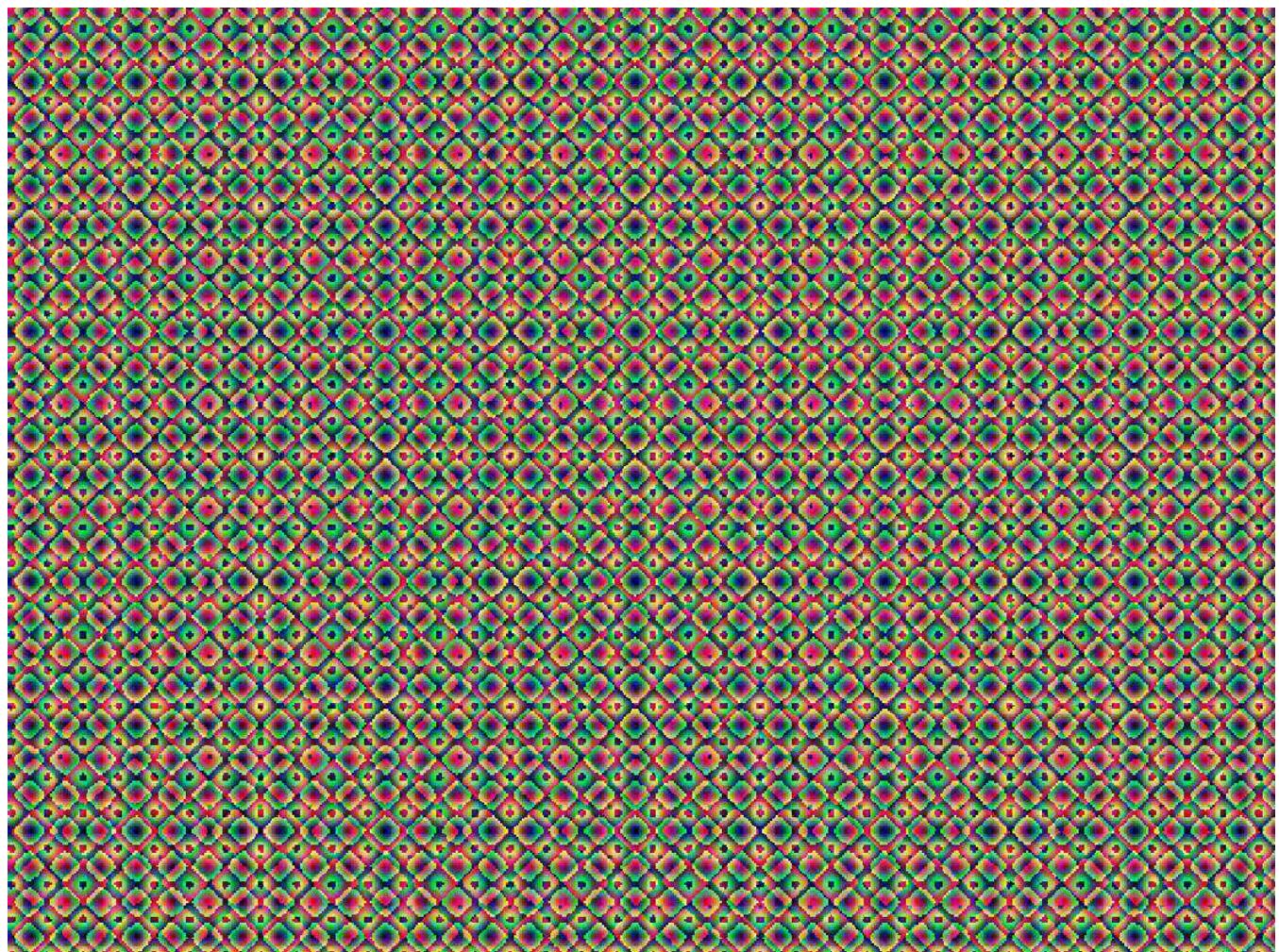
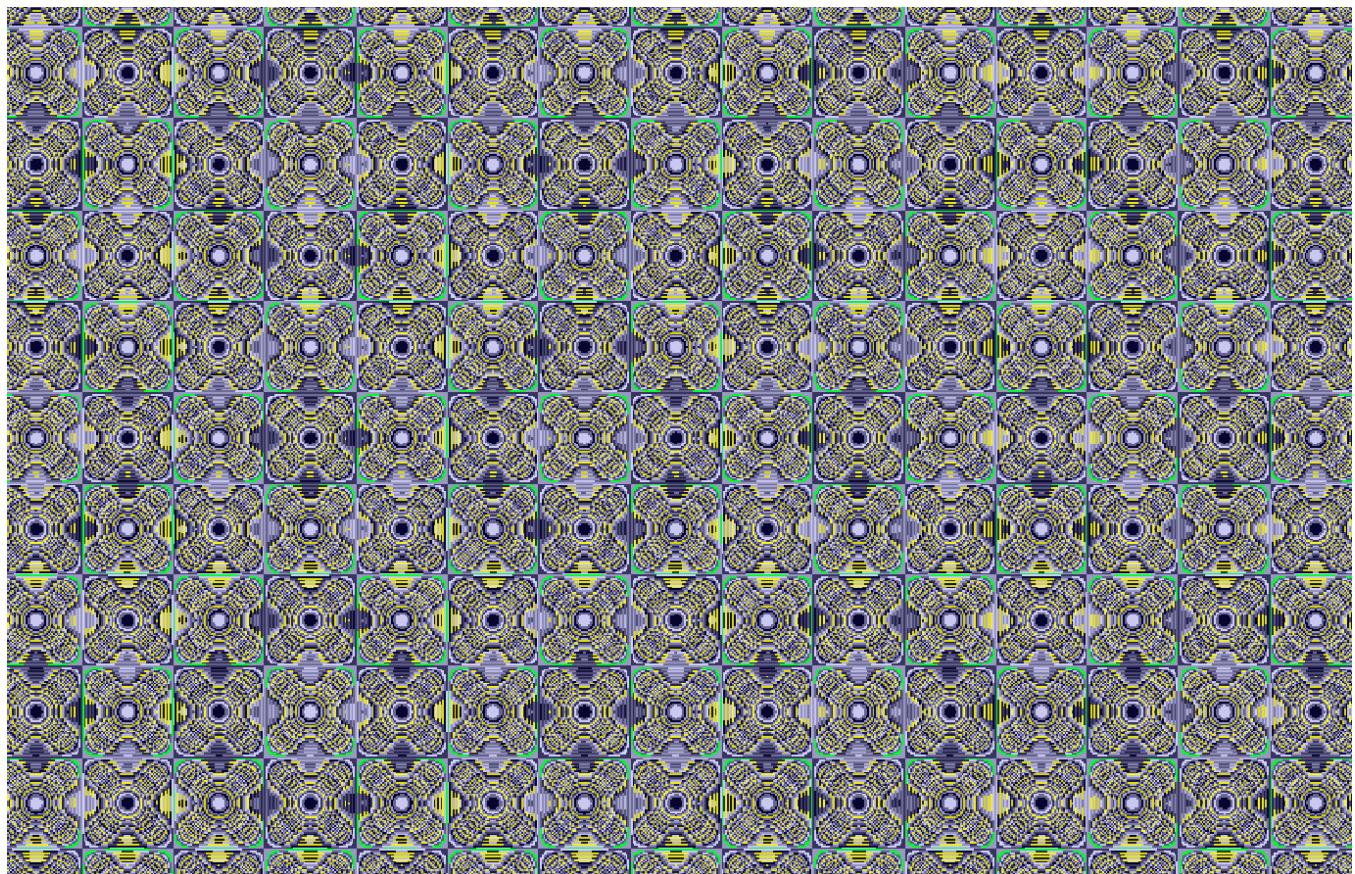


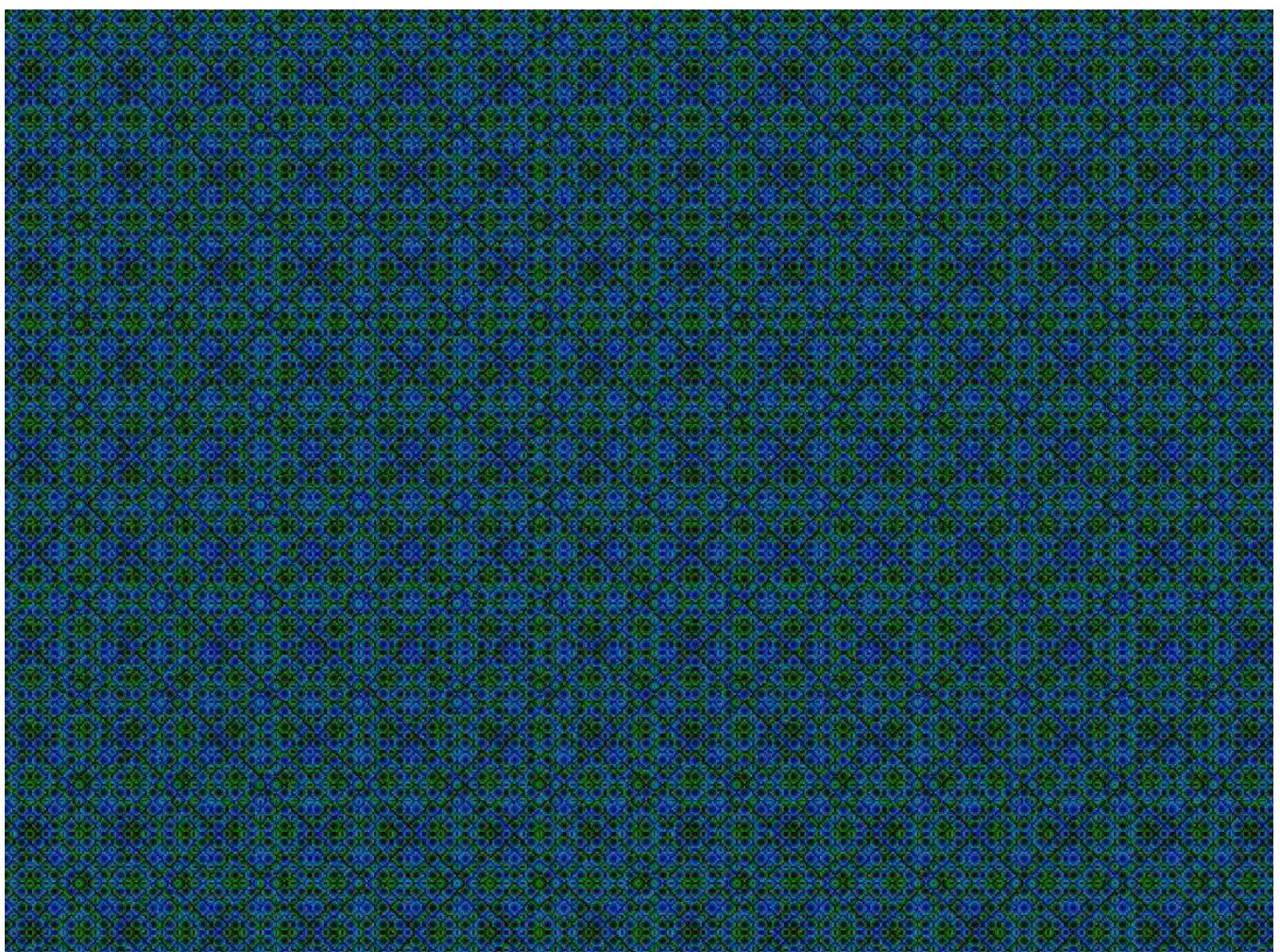
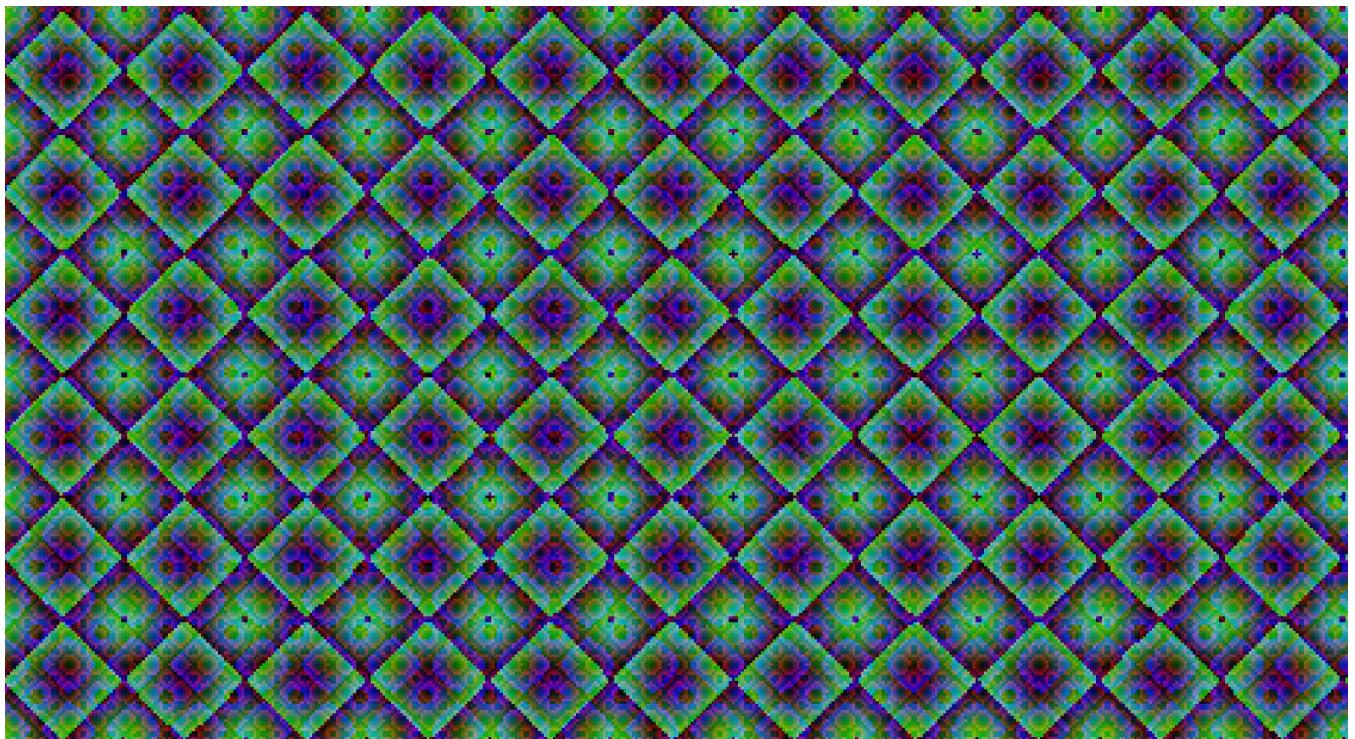


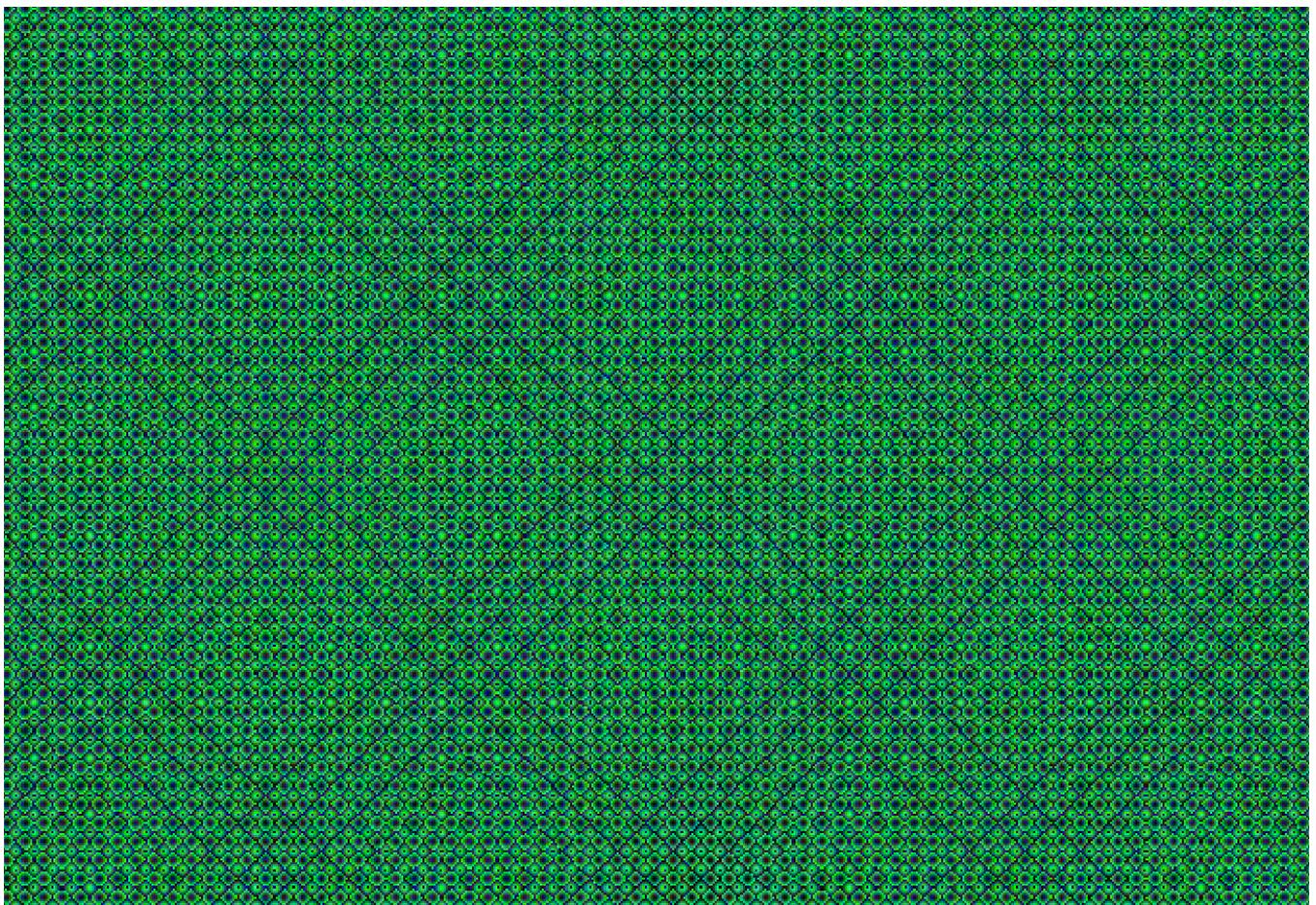
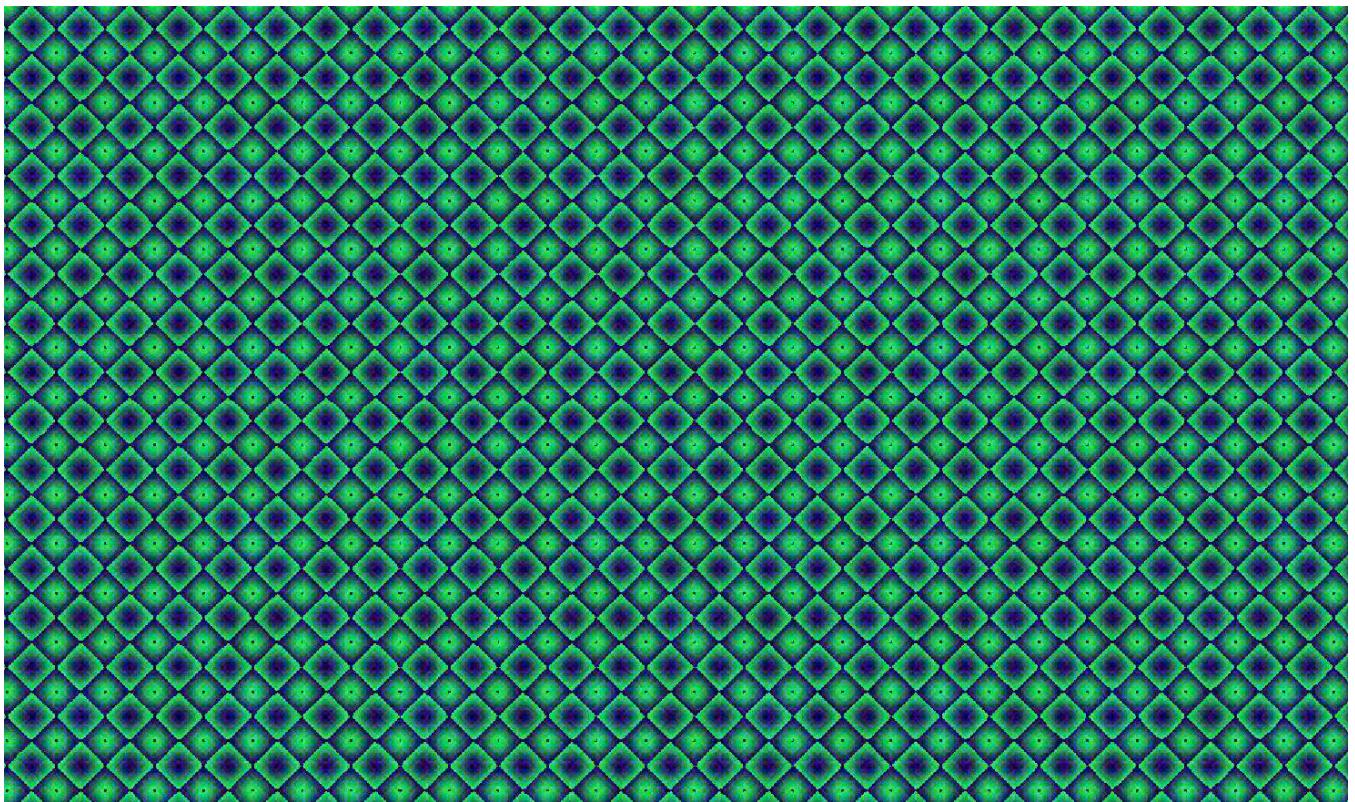


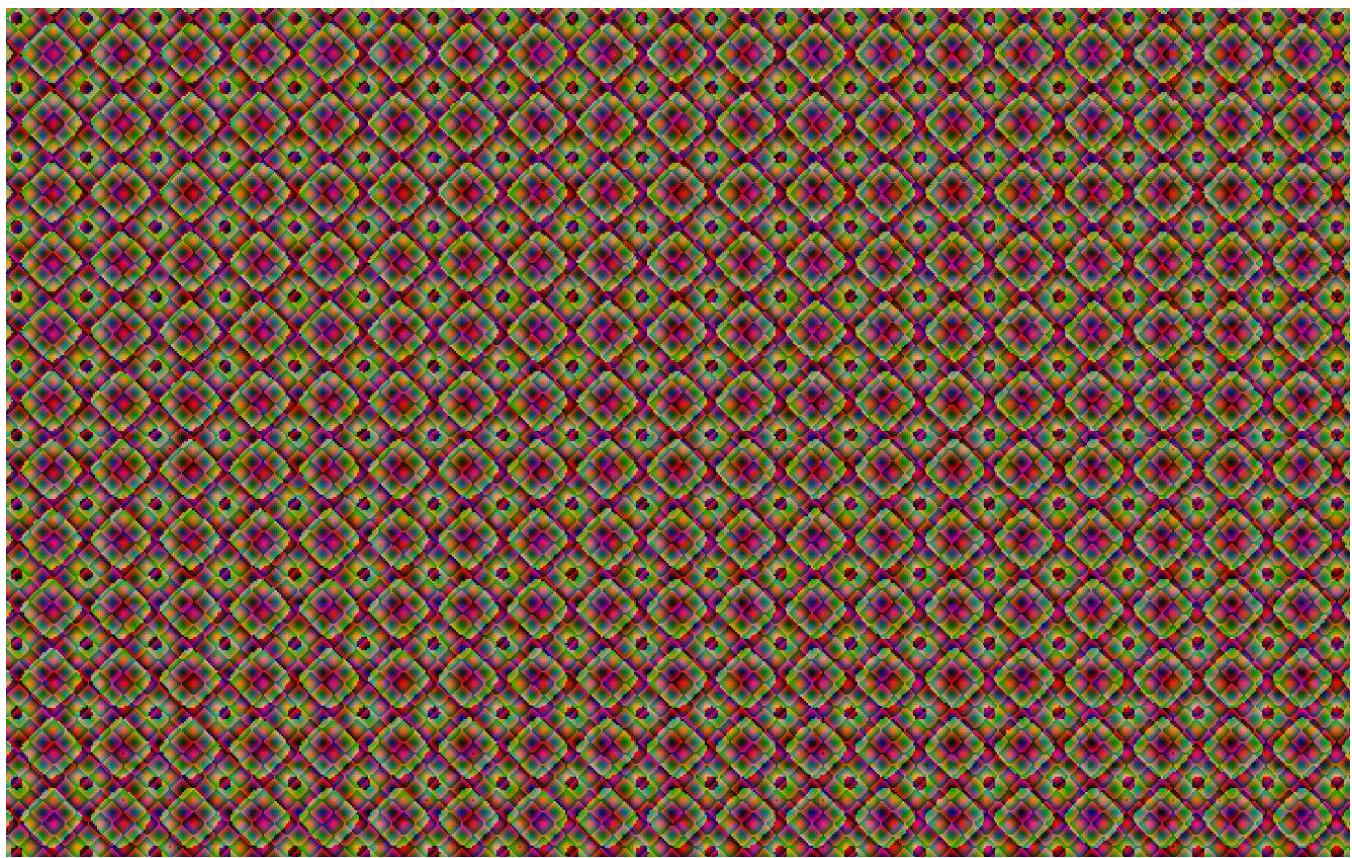
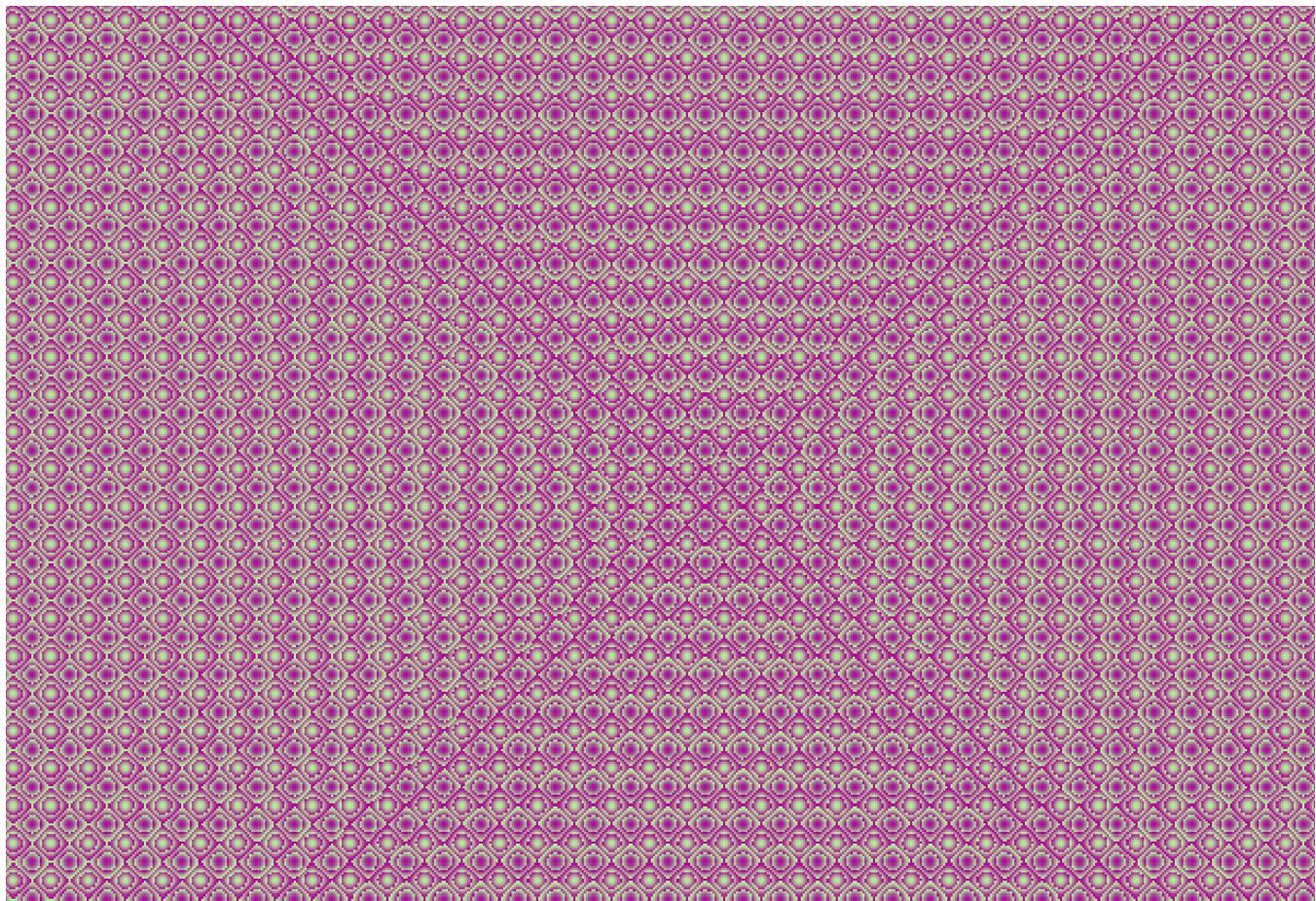


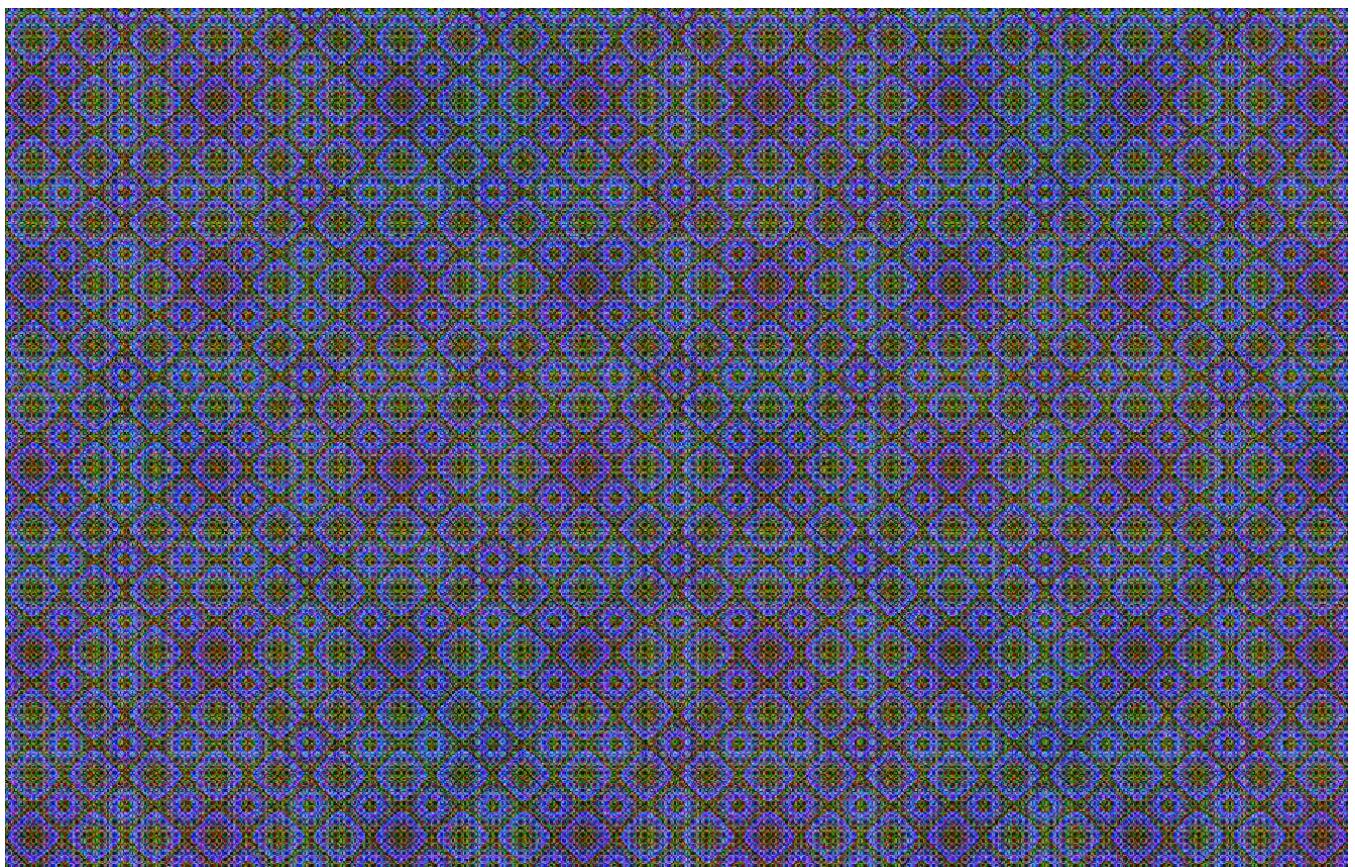
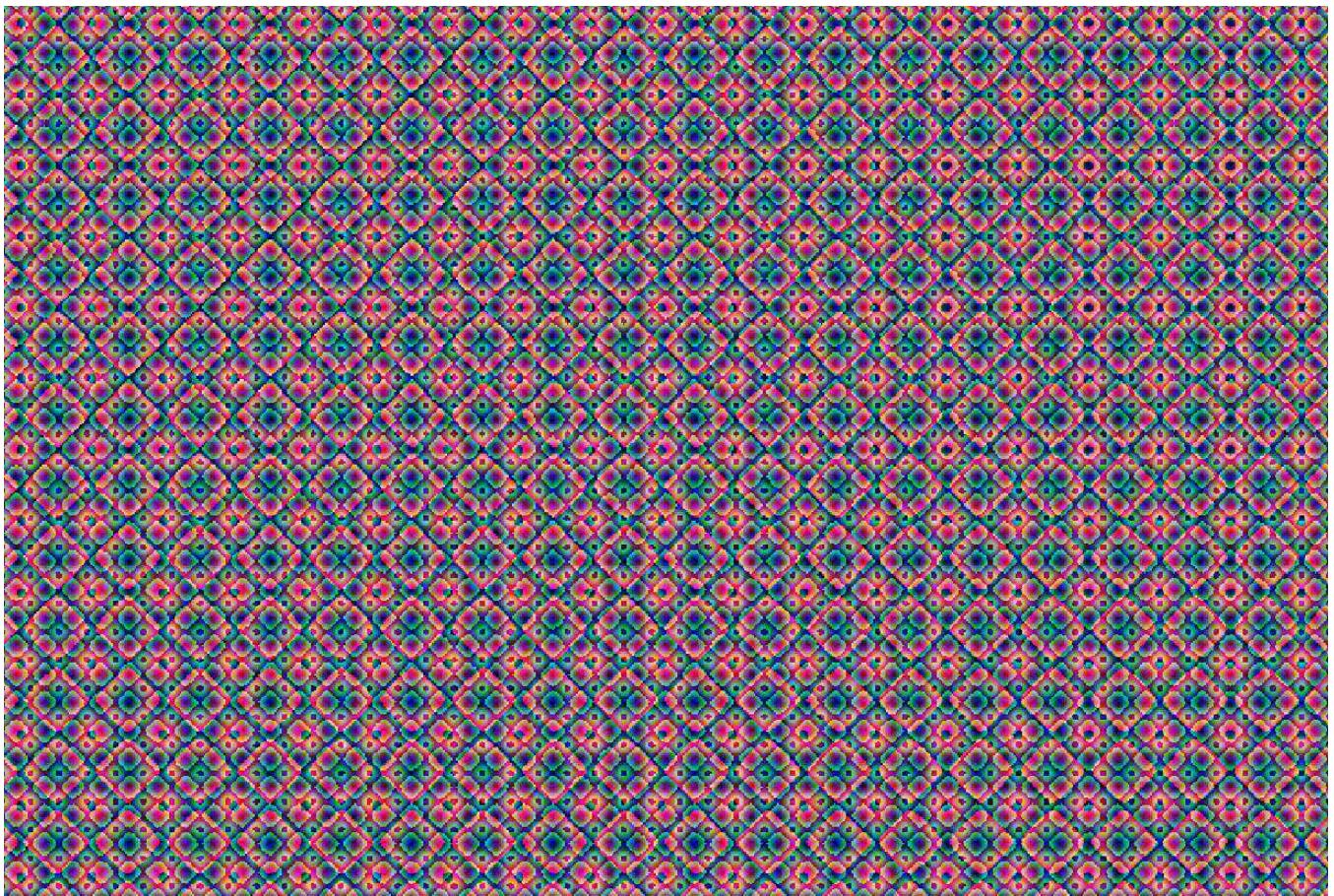




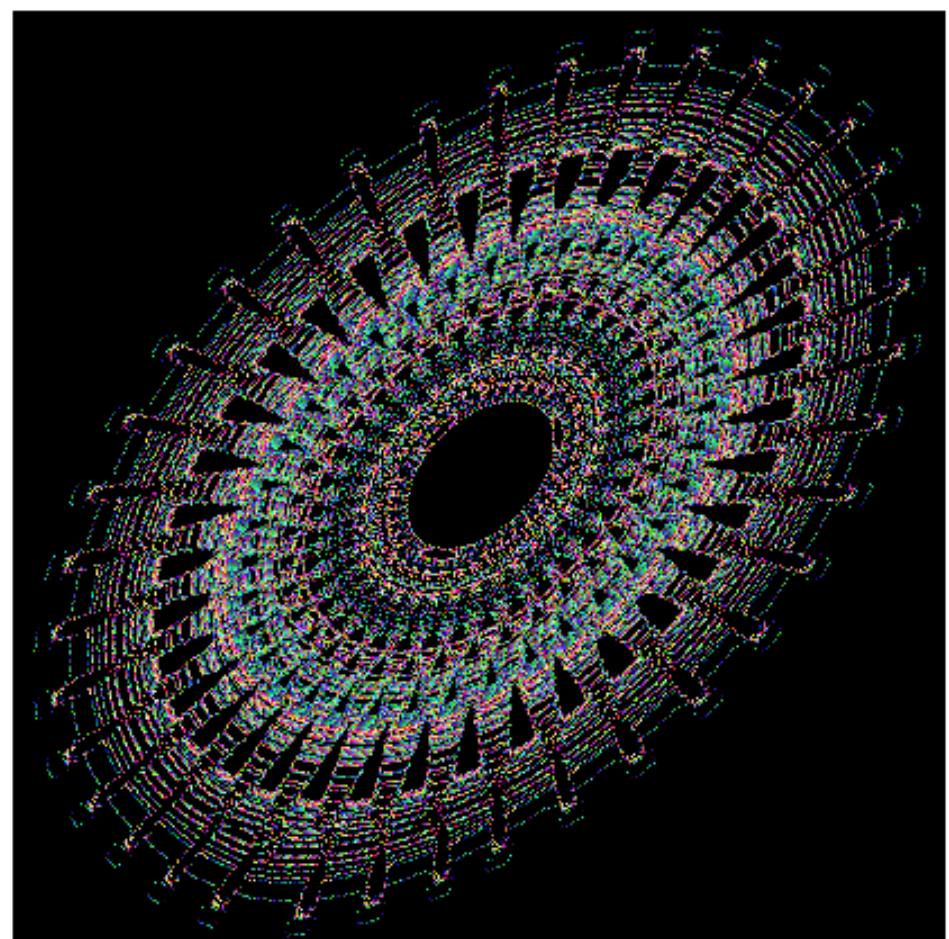
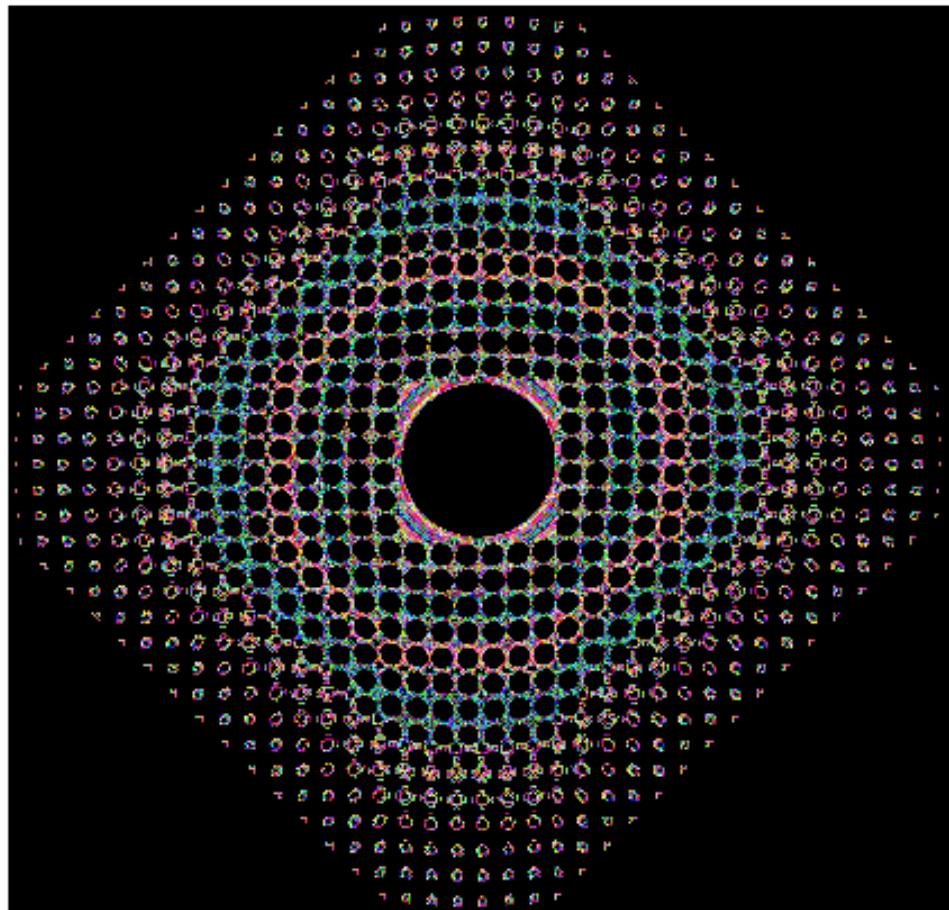


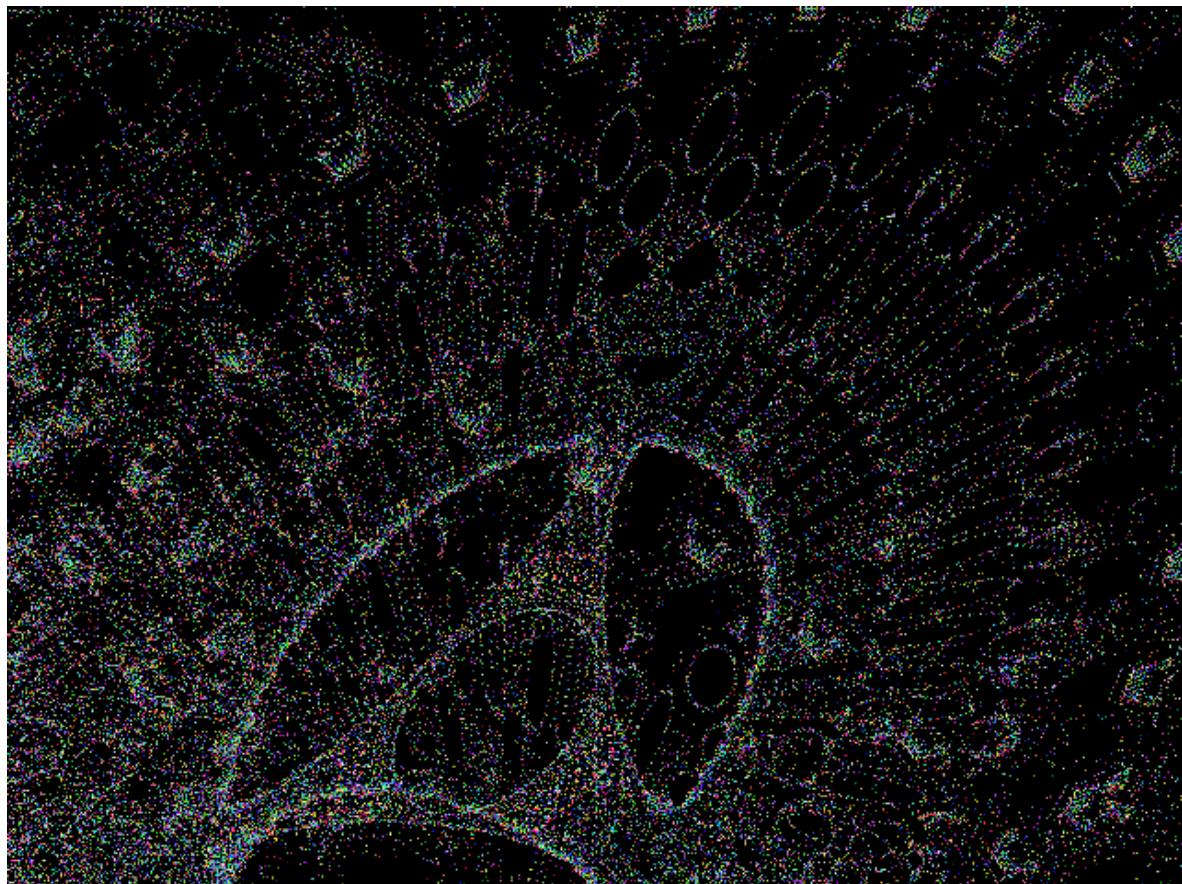
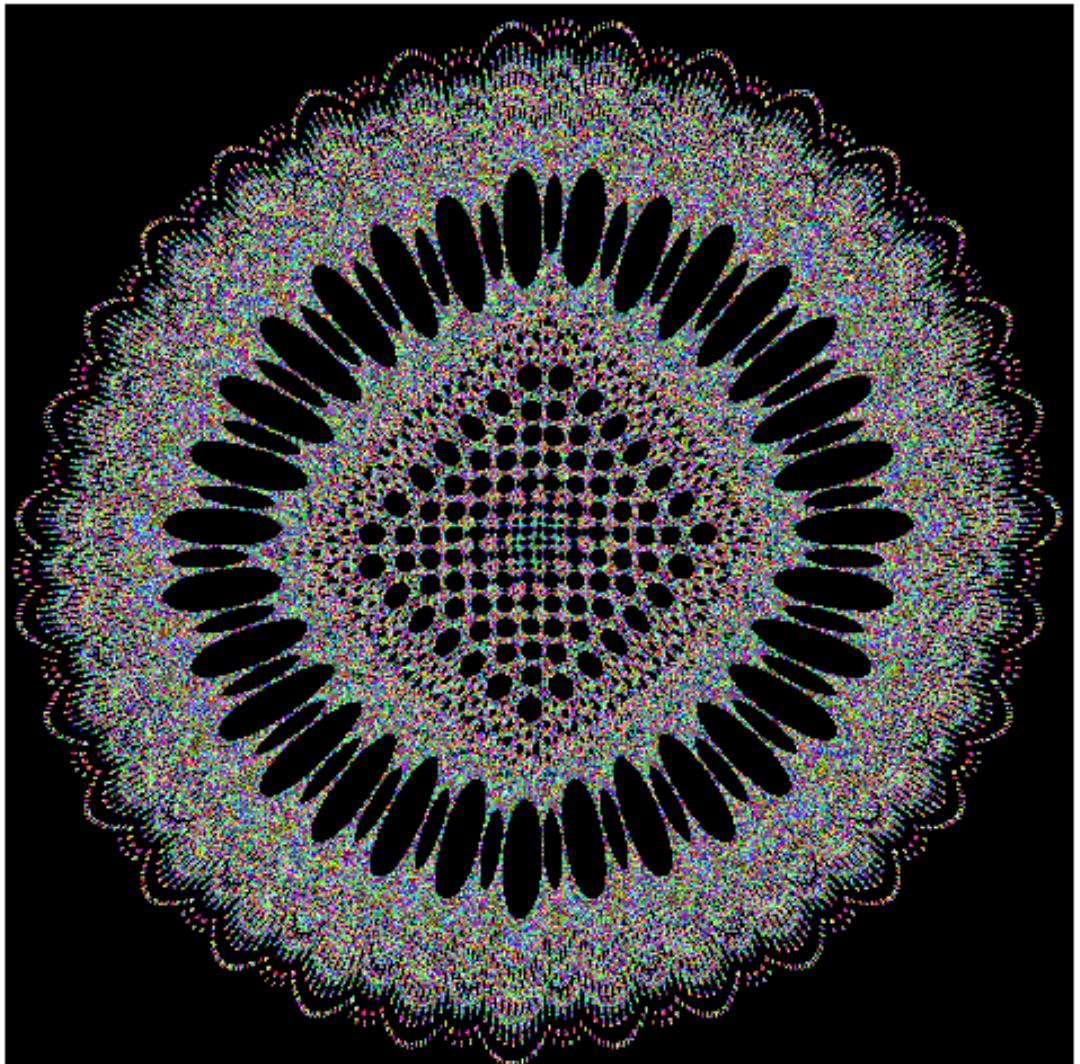




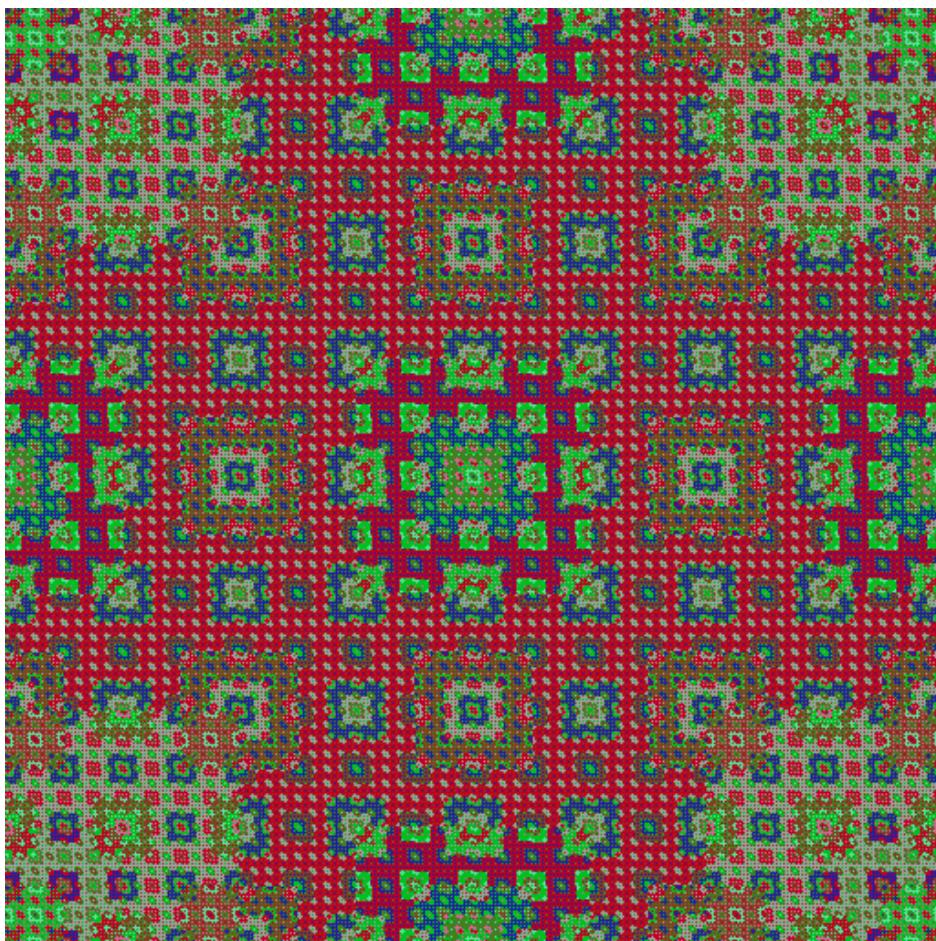
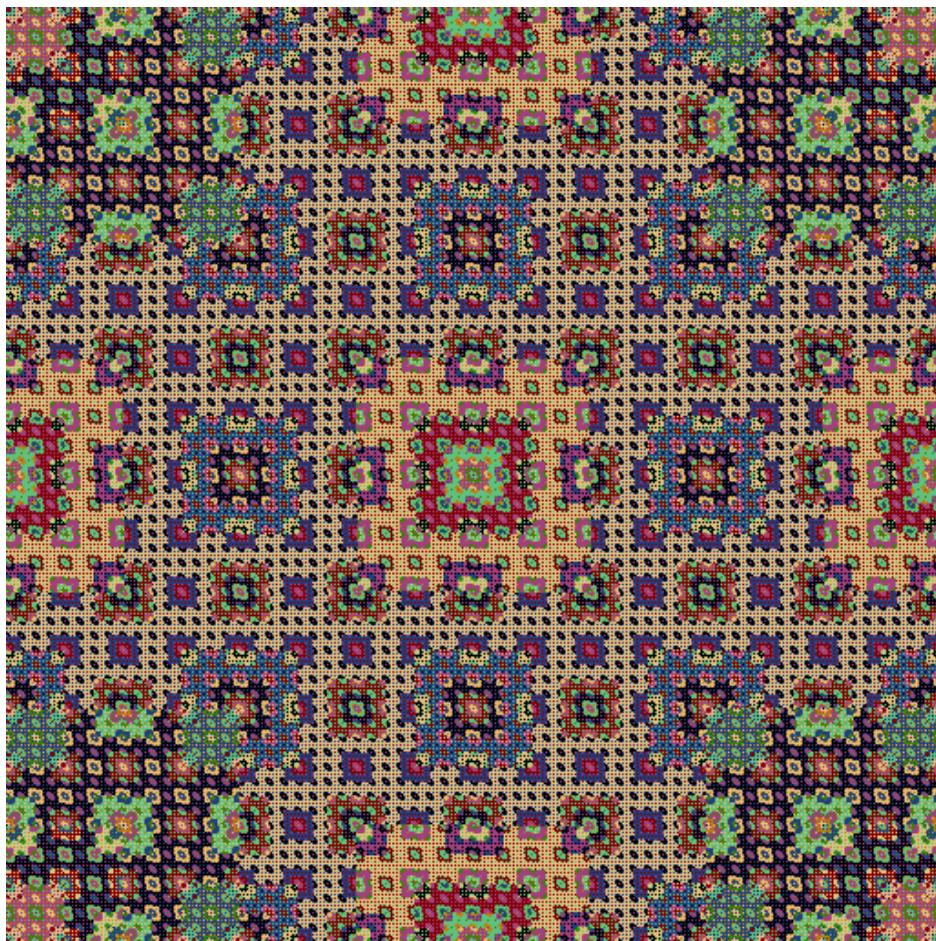


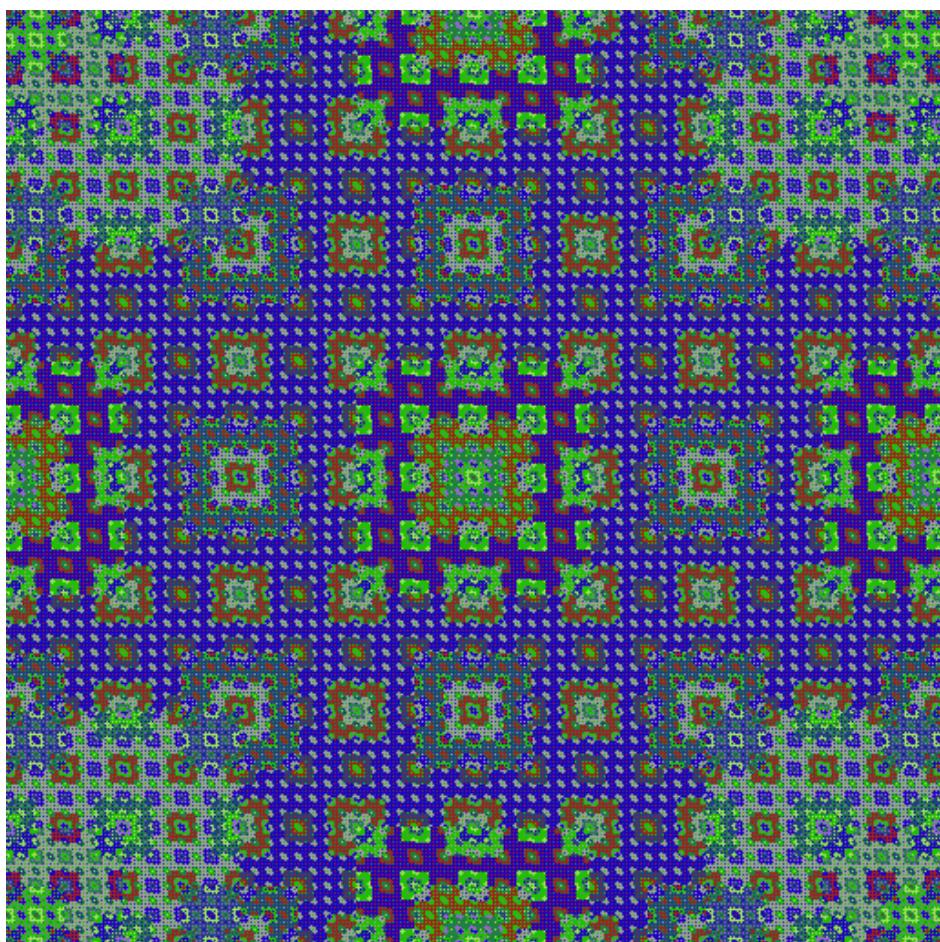
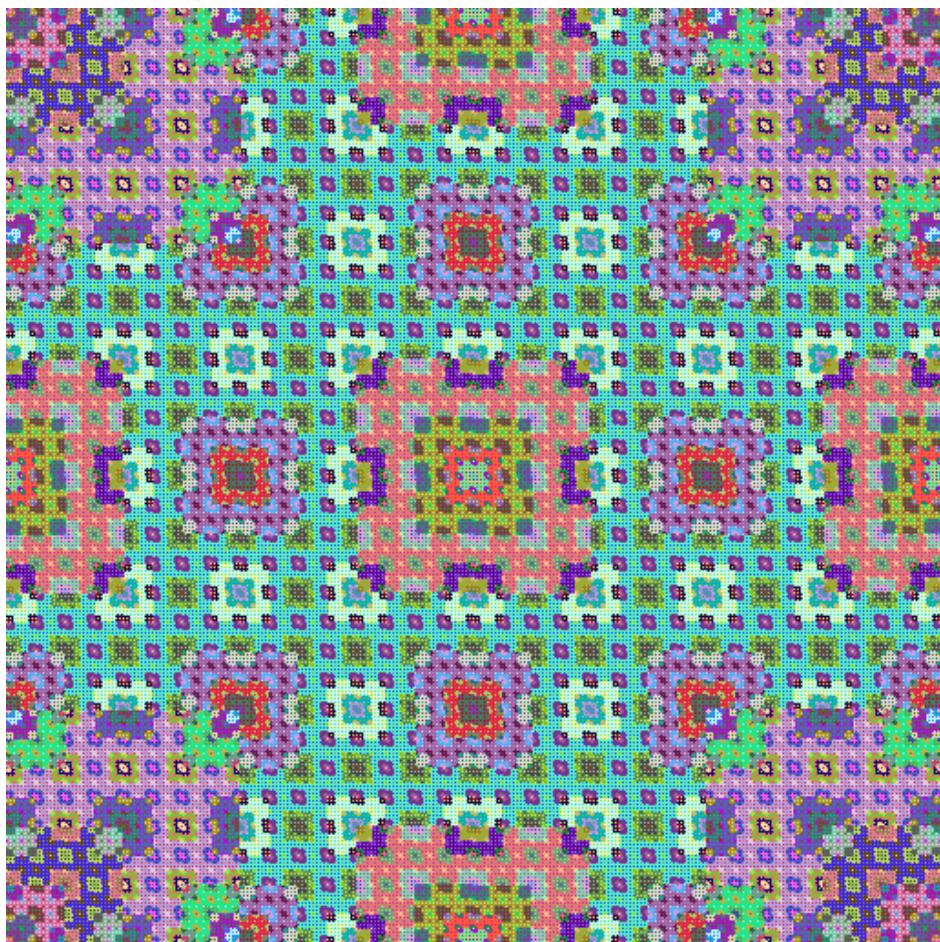
### Sezione III – Formule ricorrenti





## Sezione IV – Tappeti persiani





ISBN 978-88-89249-62-8



9 788889 249628 >

Lorenzo Repetto

# Dai giochi agli algoritmi

## **Parte prima. Con carta, matita, gomma ... e computer**

1. Un inizio “problematico”
2. Omaggio a Smullyan
3. Piaceri e limiti del calcolo
4. Problemi intrattabili o quasi
5. Tappezzerie, decorazioni e altre cose

## **Parte seconda. Da zero a due giocatori: divertimenti con il computer**

6. Automi cellulari e macchine di Turing
7. La corsa del cavallo sulla scacchiera
8. La soluzione sta nella traccia!
9. Una partita a tris
10. Awari... ma con un occhio agli scacchi!
11. Uno dei due vince sempre!

Questo libro vuol essere, da un lato, una rivisitazione e un arricchimento di alcune lezioni – non soltanto di argomento ludico – raccolte nel corso degli anni dall’autore, insegnante di informatica e studioso di lunga esperienza; dall’altro, un approfondimento delle problematiche proposte nelle gare di informatica organizzate nel mondo della scuola.

I destinatari naturali di questo lavoro sono dunque gli studenti, sebbene il volume sia strutturato in modo da poter essere consultato da tutti i cultori di “passatempi informatici” o della programmazione di giochi al computer, qualunque sia il loro livello di preparazione tecnica.

