Le torri di… “ahinoi!”

Sommario.

Una brevissima digressione informatica sul famoso problema delle “Torri di Hanoi” proposto da Édouard Lucas (nel dagherrotipo qui a fianco) nel suo ponderoso *“Récréations mathématiques*” (edito in quattro volumi tra il 1882 e il 1894): un problema ludomatematico divenuto suo malgrado punto di passaggio pressoché obbligatorio da decenni nella didattica informatica come forzoso esempio d'uso della ricorsione. Poiché l'argomento periodicamente riemerge nei contesti più disparati, qui si mostra in modo molto succinto ma – sperabilmente – ultimativo che la ricorsione non fa che complicare inutilmente la soluzione (e la comprensione) di quello che è un banalissimo problema di enumerazione combinatoria.

1 Introduzione.

Scopo delle presenti note è mostrare il problema delle “Torri di Hanoi” nella sua basilare natura di **problema di enumerazione** e, di conseguenza, l'ottimalità della sua soluzione in termini iterativi, che risultano di semplicissima comprensione e di efficientissima implementazione anche su sistemi dotati di risorse limitatissime. Il tutto *senza l'uso di ricorsione*, che invece una insistente e persistente tradizione didattica tende ahinoi (donde il calembour del titolo) a presentare come “assolutamente inevitabile” in simili frangenti, creando una vera e propria distorsione mentale in eserciti di studenti.

2 Preliminari e definizioni.

Come nella schiacciante maggioranza degli articoli divulgativi destinati ad un vasto pubblico, anche qui rinunciamo a priori ad elevati livelli di rigore formale e indulgiamo in alcuni comuni abusi di notazione, seguendo il grande filone. Quindi non definiremo minuziosamente a priori la notazione impiegata (comunque intuitiva e universalmente diffusa) e non partiremo da una complessa definizione formale di «insieme» o di «appartenenza».

Di certo, per default, gli unici numeri che qui ci interessano sono naturali, ossia **interi non negativi** $\mathbb{N} = \{0,1,2,3,\cdots\}$, gli insiemi sono rigorosamente **finiti** senza eccezione e tutti gli indici sono implicitamente **opportuni**. Per contro useremo invece in modo molto libero e informale le nozioni di *insieme*, *sottoinsieme* (proprio e non), *lista ordinata*, *intervallo* di numeri naturali, secondo ciò che in quel momento ci fa comodo considerare, dando per assodata la corrispondenza stabilita tra i vari enti matematici[1]. Si vuole solamente sottolineare che, in questo particolare caso, l'ordine degli elementi ha sempre rilevanza ed è imprescindibile, pertanto a rigore dovremmo parlare unicamente ed esplicitamente di liste ordinate e/o intervalli.

Seguendo questo approccio decisamente minimalista, possiamo affrontare la risoluzione iterativa del problema ricordando velocemente poche, semplicissime nozioni.

1. Funzione caratteristica;
2. Codice Gray binario riflesso;
3. Ruler sequence (o Gray delta sequence).

Definizione 1: Funzione caratteristica (o indicatrice).

Dato un insieme A non vuoto e un suo sottoinsieme $S \subseteq A$, la *funzione caratteristica* (o funzione *indicatrice*) associata al sottoinsieme S è la funzione $f_S : A \to \{0,1\}$ definita come segue per ogni $a \in A$:

$$f_S(a) := \begin{cases} 1 & \text{se } a \in S \\ 0 & \text{se } a \notin S \end{cases} \tag{1}$$

[1] Per un arcinoto lemma, ogni insieme finito di cardinalità *n* può essere mappato biunivocamente sull'insieme dei primi *n* numeri naturali, il quale (usando l'intrinseco ordinamento naturale crescente) allo stesso tempo può essere visto e utilizzato computazionalmente anche come una lista ordinata, una catena e un intervallo finito: $[0, n-1]$. Pertanto, dato l'insieme arbitrario finito $A = \{a_0, a_1, \ldots, a_{n-1}\}$, una volta fissata un'applicazione biunivoca arbitraria (di norma immediata) tale per cui $a_0 \leftrightarrow 0$, $a_1 \leftrightarrow 1, \cdots$, esso è isomorfo all'insieme $A_N = \{0,1,2,\ldots,n-1\}$ e quindi anche all'intervallo di interi $[0, n-1]$, con tutto ciò che ne consegue. Ciò si applica, a fortiori, quando possiamo scegliere direttamente di usare l'insieme A_N.

In letteratura la funzione caratteristica è sovente indicata anche come $1_S, I_S$ o χ_S. Si noti, per inciso, che la cardinalità del sottoinsieme *S* è data dalla somma dei valori della sua funzione caratteristica (a rigore, la somma dei soli valori unitari):

$$|S| = \sum_{a \in A} f_S(a) \tag{2}$$

Tale funzione assume un ruolo *determinante* dal punto di vista computazionale, individuando in modo estremamente diretto ed intuitivo la più efficiente struttura dati per la rappresentazione di sottoinsiemi: un *array booleano*.

L'esperienza pratica, tuttavia, conferma che l'importanza cruciale e la centralità di tale funzione e del suo utilizzo non sono mai sottolineate a sufficienza. Si consideri ad esempio l'insieme $A = \{a, b, c\}, |A| = 3$ e il suo insieme potenza $\wp(A)$, valendo come noto $|\wp(A)| = 2^{|A|}$: per definizione, quest'ultimo è l'insieme di tutti i possibili sottoinsiemi di A (inclusi l'insieme vuoto e l'insieme stesso, visto come sottoinsieme improprio), come visibile in figura 1 qui a lato.

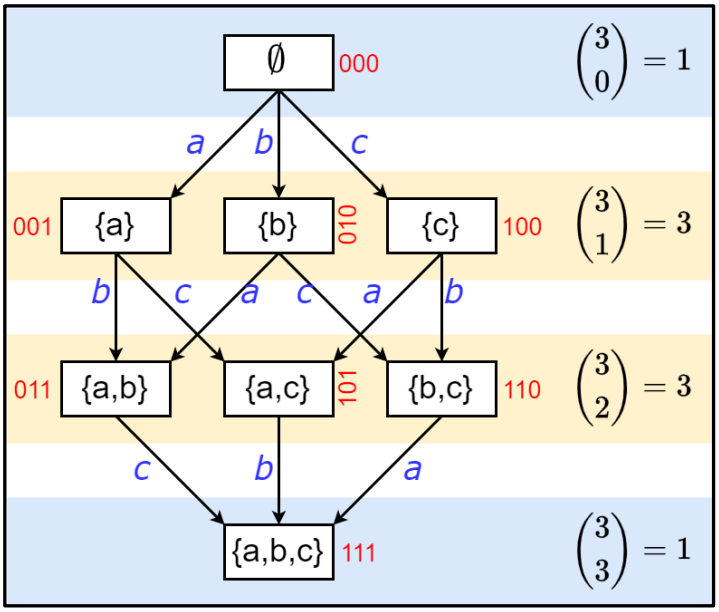


Figura 1: diagramma di Hasse arricchito dell'insieme potenza per A={a,b,c}.

Conveniamo di costruire per ogni sottoinsieme $S \subseteq A$ il relativo vettore binario caratteristico così definito:

$$V_S = f_S(c) \cdot 2^2 + f_S(b) \cdot 2^1 + f_S(a) \cdot 2^0$$

che abbiamo riportato in figura in colore rosso, a lato del sottoinsieme a cui si riferisce. Ne risulta in modo immediato che *ciascun sottoinsieme nell'insieme potenza è descritto in modo univoco da una stringa binaria*, ovvero che esiste una **biezione** tra i sottoinsiemi e i primi $2^{|A|}$ numeri binari: in ultima analisi, una volta determinata una applicazione biunivoca come quella sopra definita implicitamente tramite le funzioni caratteristiche, ciascun sottoinsieme in $\wp(A)$ è **descritto da un piccolo numero naturale**, la cui rappresentazione binaria è appunto data dal vettore caratteristico. Questo significa, d'altro canto, che generare tutti i sottoinsiemi di A si riduce[2] a null'altro che un banalissimo **conteggio incrementale**, da zero a $2^{|A|} - 1$. Se questa banalissima sequenza deduttiva fosse maggiormente diffusa, spiegata e compresa avremmo ridotto di moltissimo l'entropia su mailing list, forum, gruppi d'interesse ingolfati da richieste confuse e soluzioni farraginose per la generazione dei più diffusi oggetti combinatori elementari, praticamente ubiqui: sottoinsiemi di cardinalità k e combinazioni di n oggetti in classe k, che sono tra loro isomorfi (come evidenziato in figura dalle bande di sfondo colorate e dalle formule binomiali sulla destra), nonché numerosi altri oggetti combinatori più sofisticati, isomorfi o comunque riconducibili ai due summenzionati.

La figura 1 fornisce in realtà numerose altre informazioni: in altro contesto dovremmo dilungarci per molte pagine per illustrare le caratteristiche e le proprietà di questo **anello d'insiemi** (a partire dalla corretta duplice definizione), nella sua natura di ordine parziale e anello booleano. Ci limitiamo ad affidare all'osservazione e all'intuito del lettore le innumerevoli proprietà e relazioni tra gli enti coinvolti, fornendo solo alcuni telegrafici spunti di riflessione tra i tanti possibili:

- La cardinalità (arietà) di ciascun sottoinsieme è pari al **numero di bit alti** nel relativo vettore che lo descrive, come già implicitamente indicato nella formula (2);
- Al complemento di un dato sottoinsieme $A \setminus S$ corrisponde la **negazione** (NOT) del relativo vettore V_S, quindi ad esempio $A \setminus \{b\} = \{a, c\} \Rightarrow \sim 001 = 110$;
- L'unione di due sottoinsiemi e l'OR dei relativi vettori binari coincidono: $\{a\} \cup \{b\} \Rightarrow 001 \vee 010 = 011$.

…e questa è solamente la punta dell'iceberg. Fedeli all'intento di mantenere estremamente succinta la trattazione, completiamo l'esempio con la più classica e intuitiva delle costruzioni ricorsive manuali dell'insieme potenza (o *power set*)

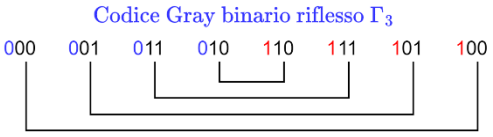
[2] A patto, ovviamente, che non vogliamo complicarci "inutilmente" la vita con ordinamenti particolari, restrizioni sulla dimensione dei subset, algoritmo del banchiere e altre amenità, comunque secondarie rispetto al concetto qui espresso.

$\wp(A)$ in $2^{|A|-1}$ passaggi, sfruttando intrinsecamente la *partizione* creata da un sottoinsieme e dal suo complemento, senza ulteriori spiegazioni:

V_S	S	$A \setminus S$	$V_{A\setminus S}$
000	∅	{a,b,c}	111
001	{a}	{b,c}	110
010	{b}	{a,c}	101
100	{c}	{a,b}	011

Definizione 2: Codice Gray binario riflesso.

Il codice Gray (OEIS: A014550) è un **codice a minima variazione**, studiatissimo e con centinaia di applicazioni discretistiche e combinatorie: il più recente survey disponibile (Mütze 2024) consta di oltre 90 pagine dense di riferimenti. Nella sua versione binaria, la sequenza è tale che tra un elemento ed il suo successore si ha sempre la variazione di **uno e un solo bit**.



Il codice Gray binario è definito in modo ricorsivo, laddove il pedice indica la lunghezza della stringa di bit mentre ϵ rappresenta la stringa binaria vuota, di lunghezza nulla:

$$\begin{cases} \Gamma_0 & = \epsilon \\ \Gamma_{n+1} & = 0\Gamma_n, 1\Gamma_n^R \end{cases} \tag{3}$$

La definizione induttiva, semplificando, indica di riscrivere l'intera sequenza precedente Γ_n anteponendo a ciascuna stringa binaria uno zero, raddoppiandone poi la lunghezza con la medesima sequenza, ma *letta al contrario* e anteponendo in questo caso un uno. L'immagine a fianco e un semplice esempio tabulare per i primi valori di n chiariranno definitivamente tale modalità costruttiva.

Γ_0 = “”
Γ_1 = 0,1
Γ_2 = 00,01,11,10
Γ_3 = 000,001,011,010,110,111,101,100

La tipica sequenza «specchiata», a meno dei prefissi, è il lapalissiano motivo per cui si parla di codice «riflesso». Esiste una giustamente famosa forma chiusa per la generazione della sequenza Gray. Nella sua versione più diffusa, fa uso dello XOR e dello scorrimento a destra (divisione binaria per due) per calcolare $\Gamma[n]$, ovvero l'n-esimo codice Gray:

$$\Gamma[n] = n \oplus (n \gg 1) \tag{4}$$

La tabella seguente riassume i primi otto valori binari per i due operandi dello XOR e il codice Gray corrispondente:

n	$n \gg 1$	$\Gamma[n]$
000	000	**000**
001	000	**001**
010	001	**011**
011	001	**010**
100	010	**110**
101	010	**111**
110	011	**101**
111	011	**100**

Definizione 3: Ruler sequence.

Una sequenza strettamente correlata al codice Gray e **fondamentale** nella risoluzione iterativa del problema proposto è la cosiddetta sequenza del regolo (*ruler sequence*) o 2-adic, OEIS: A001511, definita per ogni $n > 0$:

$$R(n) = 1,2,1,3,1,2,1,4,1,2,1,3,1,2,1,5,\ldots \quad (5)$$

$$R(n) := \begin{cases} 1 & \text{se } n \text{ è dispari} \\ R(n/2)+1 & \text{se } n \text{ è pari} \end{cases}$$

Tra i numerosi significati combinatori noti in letteratura, possiamo evidenziare quelli di nostro immediato interesse:

1. La **posizione** (aumentata di una unità) dell'unico bit che per definizione *si inverte* nel passare dal codice Gray $\Gamma[n-1]$ al suo successore: pertanto, ad esempio, $Ruler(8) = 4$ indica che l'unico bit variato tra $\Gamma[7] = 0100_\Gamma$ e $\Gamma[8] = 1100_\Gamma$ è il 4, di posto 2^3.
2. La **potenza del due** (di nuovo, aumentata di una unità) corrispondente al *bit più a destra* che passa da 0 a 1 tra $n-1$ e n nel conteggio binario naturale: quindi, ad esempio, $Ruler(5) = 1$ indica che (essendo $5_{10} = 0101_2$ e $4_{10} = 0100_2$) il bit 1 (in realtà di posto 2^0, dato l'offset unitario) ha compiuto la transizione $0 \to 1$, sovente indicata con il simbolo ↑ in vari contesti applicativi;
3. Il **numero di bit** che si *invertono* complessivamente (indipendentemente dal verso della transizione: $0 \to 1$ o $1 \to 0$) nel passaggio tra $n-1$ e n nel conteggio binario: per esempio, $Ruler(4) = 3$ indica che – essendo $4_{10} = 0100_2$ e $3_{10} = 0011_2$ – si ha la variazione di tre bit, quelli adiacenti di posto $2^0, 2^1, 2^2$;

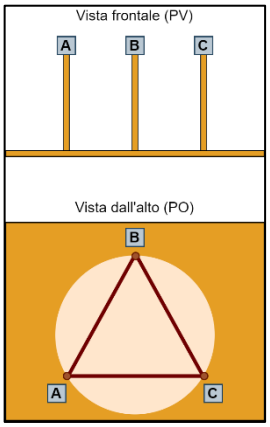
n_{10}	n_2	$Ruler(n)$	$\Gamma[n]$
0	0000	-	0000
1	0001	1	0001
2	0010	2	0011
3	0011	1	0010
4	0100	3	0110
5	0101	1	0111
6	0110	2	0101
7	0111	1	0100
8	1000	4	1100
9	1001	1	1101

Si noti che di tale sequenza esiste anche una versione equivalente (OEIS: A007814), nella quale ciascun elemento è semplicemente **decrementato** di una unità. La sequenza in questione, oltre ad esporre ancora i significati combinatori dettagliati poco sopra ai punti 1 e 2 (a meno di un ovvio off-by-one) indica anche esattamente **il numero di zeri finali** nell'espansione binaria di ciascun naturale $1 \ldots n$:

$$Ruler_{-1}(n) := 0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,4,0,1,0,2,0,1,0,3,\ldots \quad (6)$$

3 Le regole del gioco.

Le regole del gioco firmato dall'inesistente *N. Claus de Siam* (anagramma di Lucas d'Amiens) sono squadernate in un numero enorme di testi, articoli, siti web, blog, portali, etc.; d'altro canto, l'Autore ha già proposto da lungo tempo brevi note biografiche su Édouard Lucas, nell'articolo «Le problème des ménages» pubblicato originariamente qualche decennio fa sul blog «Titolo: provvisorio» e successivamente confluito nella raccolta https://tinyurl.com/56kfmb46.



Nel gioco delle torri di Hanoi si hanno a disposizione tre pioli (peg) e una pila di dischi forati o ciambelle (toroidi) di diametro crescente, predisposta ordinatamente dal più grande (in basso) al più piccolo su uno di detti pioli, che diviene così quello di partenza. I dischi sono contraddistinti univocamente, per fissare le idee, tramite un naturale non nullo, in modo tale che il disco più piccolo sia il numero 1 e tutti gli altri risultino numerati in ordine crescente per diametro crescente, avendo così $\phi_1 < \phi_2 < \cdots < \phi_n$, con ovvio significato dei simboli e dei pedici; i pioli possono essere denominati A, B, C come in figura, che ne rende molto evidente la ciclicità. Si assume, per fissare le idee, che sia A il piolo di partenza.

Il gioco consiste nello spostare e ricostruire l'intera pila ordinata su uno degli altri due pioli, arbitrariamente scelto, osservando due semplicissime regole:

1. Si può spostare uno e un solo disco per ogni mossa.
2. Si può appoggiare su un disco dato solo e unicamente un disco di diametro minore, i.e. con ordinale minore.

Il numero totale di mosse $M(n)$ in funzione del numero n di dischi presenti è pari a:

$$M(n) = 2^n - 1 \tag{7}$$

4 Sequenze risolutive.

Il gioco, nella sua estrema semplicità, si presta ad illustrare innumerevoli tecniche risolutive: funzioni ricorsive, algebre di processo, automi rule-based (es. DFA), constraint programming... Tuttavia, analizzando le sequenze di mosse riportate negli esempi, possiamo notare una forte regolarità nelle sequenze stesse, che consente di fare alcune osservazioni e di desumere le regole generali, utilizzando formule elementari che sono **unicamente funzione dell'ordinale della mossa corrente** per generare la sequenza risolutiva, senza fare ricorso a strutture di controllo e virtualmente senza occupare spazio in memoria che ecceda un eventuale $O(n)$ destinato alla mera memorizzazione dello stato del sistema.

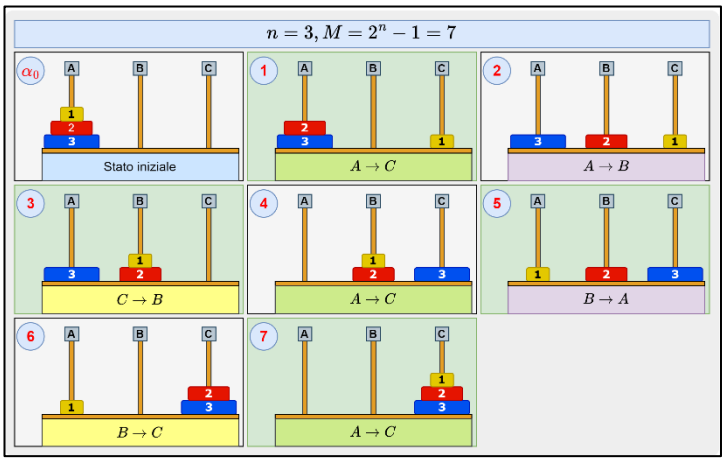


Figura 2: Sequenza risolutiva per n=3.

Le figure 2 e 3 rappresentano le sequenze minime per i casi $n = 3$ e $n = 4$ rispettivamente.

Sia $D = \{1,2,3,\dots,n\}$ l'insieme dei dischi. Conveniamo di indicare nel seguito i dischi direttamente tramite il valore naturale della loro etichetta, come $(1), (2), (3), \dots$ oppure simbolicamente come d_i dove il pedice rappresenta la medesima etichetta $i \in [1, n]$.

La strategia risolutiva manuale più efficace prevede le sequenze di mosse riportate di seguito, indicando come da intestazione tabella:

- Il numero della mossa $[1, 2^n - 1]$;
- Il disco (i);
- Il piolo di origine "da";
- Il piolo di destinazione "a";
- Lo stato dopo ogni mossa dei tre pioli visti come sottoinsiemi tramite il relativo vettore caratteristico binario $V_p = \sum_i d_i \cdot 2^{d_i - 1}$, come indicato più avanti (al punto 1 nell'elenco delle strutture dati);
- L'array D[] che associa ciascun disco al piolo su cui è attualmente inserito.

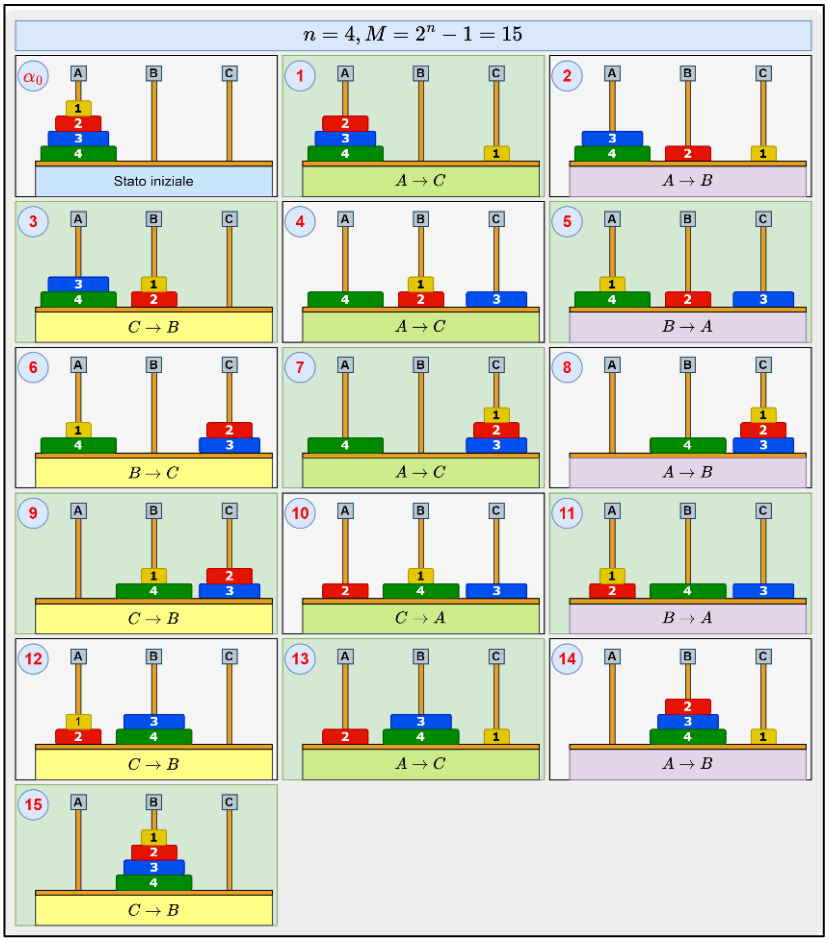


Figura 3: Sequenza risolutiva per n=4.

Esempio 1: n = 4, mosse 15 (vedi fig. 3)

mossa	disco	da	a	V_A	V_B	V_C	D[1;2;3;4]
1	**(1)**	**A**	**C**	**1110**	**0000**	**0001**	**{C,A,A,A}**
2	(2)	A	B	1100	0010	0001	{C,B,A,A}
3	**(1)**	**C**	**B**	**1100**	**0011**	**0000**	**{B,B,A,A}**
4	(3)	A	C	1000	0011	0100	{B,B,C,A}
5	**(1)**	**B**	**A**	**1001**	**0010**	**0100**	**{A,B,C,A}**
6	(2)	B	C	1001	0000	0110	{A,C,C,A}
7	**(1)**	**A**	**C**	**1000**	**0000**	**0111**	**{C,C,C,A}**
8	(4)	A	B	0000	1000	0111	{C,C,C,B}
9	**(1)**	**C**	**B**	**0000**	**1001**	**0110**	**{B,C,C,B}**
10	(2)	C	A	0010	1001	0100	{B,A,C,B}
11	**(1)**	**B**	**A**	**0011**	**1000**	**0100**	**{A,A,C,B}**
12	(3)	C	B	0011	1100	0000	{A,A,B,B}
13	**(1)**	**A**	**C**	**0010**	**1100**	**0001**	**{C,A,B,B}**
14	(2)	A	B	0000	1110	0001	{C,B,B,B}
15	**(1)**	**C**	**B**	**0000**	**1111**	**0000**	**{B,B,B,B}**

Esempio 2: n = 3, mosse 7 (vedi fig. 2)

mossa	disco	da	a	V_A	V_B	V_C	D[1;2;3]
1	**(1)**	**A**	**C**	**110**	**000**	**001**	**{C,A,A}**
2	(2)	A	B	100	010	001	{C,B,A}
3	**(1)**	**C**	**B**	**100**	**011**	**000**	**{B,B,A}**
4	(3)	A	C	000	011	100	{B,B,C}
5	**(1)**	**B**	**A**	**001**	**010**	**100**	**{A,B,C}**
6	(2)	B	C	100	000	110	{A,C,C}
7	**(1)**	**A**	**C**	**000**	**000**	**111**	**{C,C,C}**

Poniamo innanzi tutto attenzione alla sequenza dei dischi movimentati: $1, 2, 1, 3, 1, 2, 1, 4, \cdots$ certamente appare familiare, perché l'abbiamo vista poco sopra. Si tratta proprio della *ruler sequence*!

Si nota anche che per n dispari il piolo d'arrivo finale risulta essere C, mentre se n è pari la pila viene ricostruita su B. Non si tratta di una occasionale casualità: la regola si estende sistematicamente a qualsiasi valore di n, e caratterizza la formulazione originale del gioco. Appare anche immediatamente evidente, osservando le righe a sfondo grigio, che la metà più una delle mosse (in particolare quelle *dispari*, secondo la notazione qui utilizzata) coinvolgono il disco (1) con una sequenza chiaramente ciclica, caratterizzata da un pattern fisso regolare che segue una rotazione antioraria secondo la nostra convenzione: C, B, A, C, B, A etc.

Estendendo e generalizzando l'osservazione appena esposta, appare intuitivamente evidente che la tabella delle frequenze delle movimentazioni per ordinale disco è inversamente proporzionale al "peso" del disco e si applica in realtà a qualsiasi valore di n:

1	2	3	4
8	**4**	**2**	**1**

Si nota in particolare che il disco con ordinale maggiore verrà movimentato **una sola volta**, dal piolo di origine a quello di destinazione, peraltro esattamente a metà della sequenza. Più in generale, avremo $Freq = 2^{n-d_i}$ per ogni disco $d_i \in [1, n]$.

Tornando al disco (1), se indichiamo con $om = 2q + 1$ (dove il naturale $q \in [0, 2^{n-1} - 1]$) l'ordinale della mossa dispari corrente (*odd move*), usando l'intuitiva corrispondenza $A = 0, B = 1, C = 2$ avremo:

$$Dest_{odd} = (om - 2)(mod\ 3) \equiv (om + 1)(mod\ 3) \qquad (8)$$

Quindi, nel caso $n = 4$ si avrà:

q	om	$dest_{odd}$	peg
0	1	2	C
1	3	1	B
2	5	0	A
3	7	2	C
4	9	1	B
5	11	0	A
6	13	2	C
7	15	1	B

Senza dilungarci in ulteriori esempi e passaggi algebrici fondamentalmente banali, con analoghi ragionamenti (anche sulla forma binaria dell'ordinale di mossa) si giunge con facilità ad esprimere deterministicamente **in forma chiusa** ciascuna mossa in termini di piolo di partenza e di arrivo, *in funzione del solo ordinale naturale della mossa corrente*. Se infatti

indichiamo con $m \in [1, 2^n - 1]$ tale ordinale, abbiamo le seguenti comodissime formule modulari, che fanno uso anche di AND e OR bitwise:

$$Src = (m \wedge (m - 1))(mod\ 3) \tag{9}$$

$$Dest = (m \vee (m - 1) + 1)(mod\ 3) \tag{10}$$

Queste semplici e intuitive deduzioni, qui illustrate nel modo più informale, sono suffragate da una mole di letteratura – tra cui (Gardner 2008) e i noti (Scorer 1944), (Smillie 1973), (Allouche e Shallit 2009) – che ha comprovato l'isomorfismo del problema con una semplice sequenza a minima variazione come la (5), correlata al codice Gray binario. Tale sequenza, a sua volta, equivale alla ricerca di un cammino hamiltoniano (un percorso sequenziale che tocca una e una sola volta ciascun vertice) in un grafo: vedi fig. 4.

Ricapitolando in estrema sintesi i risultati esposti sopra:

A. La **sequenza dei dischi da spostare** è esprimibile in forma chiusa in funzione del solo ordinale della mossa corrente $m \in [1, 2^n - 1]$ tramite la *ruler sequence* (5), talora indicata anche come *Gray delta sequence*;

B. I pioli di **origine e destinazione** possono parimenti essere individuati tramite le semplici formule chiuse (9) e (10), conoscendo solamente l'ordinale della mossa corrente.

A rigore, queste due informazioni sono tutto ciò che è necessario per ricostruire l'intera sequenza risolutiva come esposta nelle prime colonne delle tabelle d'esempio 1 e 2 in esattamente $2^n - 1$ passi: il tutto **senza utilizzo di ulteriore memoria** e, come già accennato varie volte, senza necessità alcuna di ricorsione.

Tuttavia, a fini informazionali possiamo aggiungere ulteriori dati con minimo aggravio di costi, ad esempio per una visualizzazione (grafica o testuale) dello stato del sistema ad ogni mossa, o per realizzare soluzioni ibride (es. stack simulato con push e pop fittizie, implementando il solo calcolo dei pioli di origine e destinazione).

Lo stato del sistema ad ogni step, come esemplificato nelle figure 2 e 3, può essere rappresentato esplicitamente in numerosi modi e con varie strutture dati.

Le più utili per quanto ci riguarda, dal punto di vista informazionale, sono certamente:

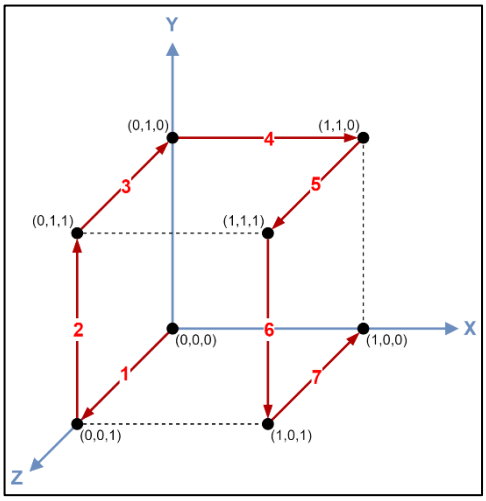


Figura 4: cammino hamiltoniano in un cubo booleano, si notino le coordinate binarie.

1. Tre **vettori booleani** (legati alle funzioni caratteristiche) di dimensione pari a n, ovviamente implementabili come interi o come un singolo bit array di dimensione $3 \cdot n$ con appropriata aritmetica degli indici. Si vedano V_A, V_B, V_C nelle tabelle d'esempio. Soluzione ovviamente ridondante, ma di grande efficacia in numerosi contesti;

2. Un **compatto array D[]** di n piccoli interi, nel quale l'elemento i-esimo indica su quale dei tre pioli è attualmente inserito il disco (i) – si vedano le colonne all'estrema destra nelle tabelle degli esempi 1 e 2. Tale approccio risulta di particolare utilità solo quando si conosce il numero del disco da spostare e il piolo di destinazione;

3. In modo più scolastico e ridondante, una **matrice di interi** $3 \times n$ con funzione di stack LIFO, nella quale l'elemento $E_{r,c}$ contiene un valore non nullo se e solo se il disco (i) è attualmente inserito sul piolo r e la posizione c è il numero di dischi che precedono quello correntemente in cima allo stack (i tre puntatori di stack/contatori di elementi necessari possono ovviamente essere inseriti in una colonna aggiuntiva della medesima matrice); anche in questo caso, la matrice può essere validamente implementata con un singolo array di dimensione $3 \cdot (n + 1)$. Ad esempio, considerando la mossa $m = 5$ dell'esempio 1 (fig. 3), avremmo la seguente situazione nella matrice ipotizzata (i cui contenuti sono rappresentati dalle sole celle a sfondo grigio):

Piolo	Livello 0	Livello 1	Livello 2	Livello 3	Puntatore alla cima
A	4	1			**2**
B	2				**1**
C	3				**1**

4. Infine, lo stato del sistema può essere sintetizzato anche in **una singola stringa binaria** di dimensione n, in modo certamente meno intuitivo, che qui non approfondiremo ma che comunque fa «rientrare dalla finestra» il

codice Gray e la sequenza ruler, in questo caso nella versione (6) che equivale al conteggio degli zeri finali nella predetta stringa binaria che rappresenta lo stato.

Riguardo al punto 1, vale anche la pena di notare che questi tre array booleani, visti come i sottoinsiemi che rappresentano, espongono una serie di proprietà certamente molto intuitive, ma non banali.

a) Ad ogni step, ciascun elemento d_i dell'insieme D dei dischi è contenuto in esattamente uno dei tre sottoinsiemi;
b) Di conseguenza, i sottoinsiemi sono **disgiunti a coppie**: $S_1 \cap S_2 = \emptyset$ per qualsiasi coppia di sottoinsiemi;
c) L'unione dei tre sottoinsiemi è sempre uguale a D: $A \cup B \cup C = D$ (i tre sottoinsiemi **coprono** l'insieme dei dischi).

In altri termini, la condizione a) ci dice che scrivendo sotto forma di matrice binaria i tre vettori (riga), si avrà che per qualsiasi mossa ciascuna colonna contiene esattamente un valore pari a 1. Ad esempio, consideriamo di nuovo la mossa $m = 5$ dell'esempio 1 (fig. 3) e costruiamo la relativa matrice booleana dai tre vettori riga:

1	0	0	**1**
0	0	**1**	0
0	**1**	0	0

Esiste però la possibilità che fino a due di tali sottoinsiemi siano vuoti in un dato momento, e questo ci impedisce di applicare pienamente la definizione di **partizione dell'insieme** D. Tuttavia, abbiamo quella che viene detta partizione debole (*weak partition*) che ammette un numero massimo limitato a priori di insiemi vuoti. Tornando alla rappresentazione booleana, in ogni momento vale la relazione $V_A \oplus V_B \oplus V_C = V_1$ dove l'ultimo elemento è il vettore caratteristico di dimensione n popolato di soli valori unitari.

5 Implementazioni di esempio.

Dopo ben sette pagine di definizioni, formule, tabelle, illustrazioni e spiegazioni, dovrebbe risultare intuitivamente chiaro anche al lettore più distratto che il problema delle torri di Hanoi di ordine n si risolve banalmente in tempo $O(2^n)$ in modo iterativo, con un singolo loop e pochissime operazioni elementari, senza alcun bisogno di ricorsione e con un uso di memoria *worst case* pari a $3 \cdot (n + 1)$.

Questo implica, tra l'altro, che l'unico oggettivo limite al numero di dischi è dato… dalla pazienza dell'utente, in proporzione diretta al tempo di elaborazione. Soprattutto parlando di retrocomputing!

Supponendo di usare un home computer con clock a 1 MHz, lavorando in Assembly possiamo generare in memoria (senza stampa né animazioni grafiche) almeno 20k mosse al secondo, un valore assolutamente realistico per l'epoca, come testimonia la diretta esperienza dell'Autore. In tali condizioni, l'elaborazione completa dei circa 4,3 miliardi (4.294.967.295 per gli affetti dall'inguaribile morbo della pedanteria) di mosse per un ipotetico gioco con 32 dischi richiederebbe circa 214.748", ossia poco meno di 2 giorni, 11 ore e 40', che ovviamente aumentano notevolmente in caso di stampa a video, salvataggio su disco o REU, etc.[3]

La *ruler sequence*, seppure non banalmente esprimibile in forma chiusa usando solo operazioni elementari, risulta generabile tramite banalissimi algoritmi, basati su poche assegnazioni in memoria e altre operazioni elementari in tempo costante $O(1)$. Per l'occasione useremo (per giunta solo in parte) un elegantissimo algoritmo loopless per il calcolo della sequenza Gray, esposto in (Knuth 2005): per l'esattezza l'algoritmo **L** 7.2.1.1. Tale algoritmo è quintessenziale nella sua geniale semplicità: calcola ad ogni passo quale bit della parola Gray generata deve variare rispetto al predecessore, usando solamente delle assegnazioni in un array ausiliario per gestire una variabile il cui valore corrisponde all'n-esimo elemento della ruler sequence – in questo caso nella versione diminuita di una unità, la (6), tra i cui significati combinatori c'è appunto la **posizione del bit** che ci interessa, espressa direttamente come **esponente del due**.

Riportiamo qui di seguito la sola parte di tale algoritmo che ci riguarda, usando la medesima notazione di Knuth: $(f_n, \cdots, f_0)$ è un array ausiliario di dimensione $n + 1$ che l'autore denomina "focus pointer".

Algoritmo L di Knuth (7.2.1.1) semplificato per generare la sola Ruler sequence (Gray delta).

L1. [Initialize.] Imposta $f_j \leftarrow j$ per $0 \leq j \leq n$.
L2. [Visit.] Stampa l'elemento j-esimo della ruler sequence.
L3. [Choose j.] $j \leftarrow f_0$, $f_0 \leftarrow 0$, $f_j \leftarrow f_{j+1}$, $f_{j+1} \leftarrow j + 1$; Termina se $j = n$, altrimenti torna a **L2**.

[3] Sconsigliato agli impazienti, ai quali si raccomanda vivamente l'uso di un simulatore in modalità *warp* e, soprattutto, di accontentarsi di un numero di dischi inferiore: prendendo atto del fatto che superare anche di poco i canonici otto dischi massimi consentiti da una scolastica soluzione ricorsiva su un Commodore 64 è già una vittoria enorme e lasciando i valori da primato alle moderne CPU superscalari multicore e multithread ed alle GPU che ormai da decenni ne superano le prestazioni (se non altro si tratta di un modo costruttivo per sfruttarne l'esorbitante potenza di calcolo).

Come si vede, seppure in mancanza di una forma chiusa elementare, la generazione iterativa della sequenza è decisamente banale e richiede in pratica solo quattro assegnazioni. Nulla vieta di implementarla anche sulle retropiattaforme più minimaliste, realizzando un felice connubio tra retrocomputing e moderni progressi algoritmici.

Il primo sorgente d'esempio qui proposto deriva da una famiglia di soluzioni modulari e grafiche elaborate dall'Autore in COMAL 80 su Commodore 64 tra il 1984 e il 1986 circa.

Vale la pena incidentalmente di sottolineare per l'ennesima volta che COMAL è un linguaggio straordinariamente moderno, completo, potente e non teme confronti con gli ambienti di sviluppo coevi per HLL imperativi e modulari maggiormente "famosi". Inoltre è indiscutibile la sua superiorità rispetto a quasi tutti i BASIC (termine diretto di paragone, che ha spinto gli ideatori danesi a creare il nuovo linguaggio) circolanti all'epoca, sotto ogni aspetto oggettivo di ingegneria e classificazione dei linguaggi di programmazione.

La versione qui presentata risulta fortemente semplificata e ridotta ad una mera *proof of concept*, pensando anche alla portabilità:

- Si è rinunciato all'output delle sequenze risolutive in background su file sequenziale o su stampante (comunque facilmente implementabile come esercizio);
- Sono state sostituite le originali POKE in area display con le inerentemente più lente PRINT AT;
- Sono state espunte le numerose procedure di libreria create dall'Autore in COMAL e Assembly richiamate dinamicamente da disco.

Nonostante le numerose modifiche e la drastica riduzione delle features per renderne possibile la presentazione nel presente contesto come semplice esempio standalone, rimane tangibile il vantaggio nell'uso di tale linguaggio sulle piattaforme d'epoca: al punto che, con un intervento davvero minimale, è stato possibile variare il codice della procedura `hanoi'gray` per poter includere il modernissimo algorimo loopless L di Knuth in luogo dell'originario algoritmo di convoluzione usato all'epoca, apparso in un articolo di BYTE a fine anni '70.

Con pochissime ulteriori modifiche il lettore potrà divertirsi a rendere più o meno interattiva l'applicazione, in modo da poter seguire più facilmente a video la successione delle mosse risolutive. Il tutto, ovviamente, senza dimenticare che lo scopo ultimo di questo codice è solo quello di **dimostrare la natura computazionale del problema** e delle principali soluzioni iterative basate su banalissime formule dirette, assolutamente elementari e implementabili senza problemi anche sugli home computer più limitati.

```
 1. //********************
 2. // SAVE "hanoi"
 3. //********************
 4.
 5. // Stack simulato, 3xMAX
 6. DIM stack(3,18)
 7. li#:=5
 8. hanoi'banner
 9.
10. //FOR n:=3 TO 11 DO
11. INPUT AT li#+15,1: "Numero di dischi (3-16)? ": n
12. IF n<3 OR n>16 THEN END "Ho detto da tre a sedici, Einstein!"
13. moves:=2^n-1
14. PRINT "Mosse totali necessarie: ";moves
15. hanoi'stack(n)
16. clr'stack
17. hanoi'gray(n)
18. //ENDFOR n
19. CURSOR 25,1
20. END "Fine lavoro."
21. //*****************************
22. //** Subroutines
23. //*****************************
24.
25. //*****************************
26. //** Risoluzione tramite stack
27. //*****************************
28. PROC hanoi'stack(n)
29.   PRINT AT 24,1: "** stack simulato"
30.   //** Inizializzazioni
31.   stack(1,18):=n
32.   stack(2,18):=0
33.   stack(3,18):=0
34.   FOR i:=1 TO n DO
35.     stack(1,i):=n-i+1
36.     stack(2,i):=0
```

```
 37.     stack(3,i):=0
 38.   ENDFOR i
 39.   CURSOR li#+5,2
 40.   FOR i:=1 TO n DO
 41.     ch$:=" "
 42.     IF stack(1,i)>9 THEN ch$:="1"
 43.     PRINT AT li#+2,2+i: ch$
 44.     PRINT AT li#+3,2+i: stack(1,i) MOD 10
 45.   ENDFOR i
 46.   //*****************************
 47.   //** Loop soluzione con stack
 48.   //*****************************
 49.   FOR i:=1 TO moves DO
 50.     from:=src'peg#(i)
 51.     dest:=dest'peg#(i)
 52.     disk:=stack(from,stack(from,18))
 53.     PRINT AT li#+10,12: i;
 54.     stack(dest,18):+1
 55.     stack(dest,stack(dest,18)):=disk
 56.     stack(from,stack(from,18)):=0
 57.     stack(from,18):-1
 58.     hanoi'update(n,disk,from,dest)
 59.   ENDFOR i
 60. ENDPROC hanoi'stack
 61.
 62. //*****************************
 63. //** Ruler sequence
 64. //** Algoritmo L di Knuth 7.2.1.1
 65. //*****************************
 66. PROC hanoi'gray(n)
 67.   DIM focus(18)
 68.   PRINT AT 24,1: "** ruler sequence"
 69.   //L1. Init.
 70.   FOR i:=0 TO 17 DO
 71.     focus(i+1):=i
 72.   ENDFOR i
 73.   i:=1
 74.   REPEAT
 75.     PRINT AT li#+10,12: i
 76.     from:=src'peg#(i)
 77.     dest:=dest'peg#(i)
 78.     i:+1
 79.     //L2. Visit.
 80.     hanoi'display(focus(1)+1,from,dest)
 81.     //L3. Choose j.
 82.     j:=focus(1)
 83.     focus(1):=0
 84.     focus(j+1):=focus(j+2)
 85.     focus(j+2):=j+1
 86.   UNTIL focus(1)=n
 87. ENDPROC hanoi'gray
 88.
 89. //*****************************
 90. //** Visualizzazione stack
 91. //*****************************
 92. PROC hanoi'update(n,dsk,src,dst)
 93.   sp1:=stack(src,18)+1
 94.   sp2:=stack(dst,18)
 95.   PRINT AT li#+2*src,2+sp1: " ";
 96.   PRINT AT li#+1+2*src,2+sp1: " ";
 97.   PRINT AT li#+1+2*src,4+n: USING "(##)": sp1-1;
 98.   IF dsk>9 THEN PRINT AT li#+2*dst,2+sp2: "1";
 99.   PRINT AT li#+1+2*dst,2+sp2: dsk MOD 10;
100.   PRINT AT li#+1+2*dst,4+n: USING "(##)": sp2;
101.   hanoi'display(dsk,src,dst)
102. ENDPROC hanoi'update
103.
104. //*****************************
105. //** Visualizzazione mossa
106. //*****************************
107. PROC hanoi'display(dsk,src,dst)
```

```
108.   PRINT AT li#+11,12: USING "##": dsk
109.   PRINT AT li#+12,12: CHR$(96+src)
110.   PRINT AT li#+13,12: CHR$(96+dst)
111.   //** Pausa human-readable
112.   pause(8)
113. ENDPROC hanoi'display
114.
115. //*****************************
116. //** Preparazione schermo
117. //*****************************
118. PROC hanoi'banner
119.   PAGE
120.   PRINT "************************"
121.   PRINT "** Le torri di hanoi   **"
122.   PRINT "**                     **"
123.   PRINT "** SOLUZIONI ITERATIVE **"
124.   PRINT "************************"
125.   PRINT AT li#+2,1: "A"
126.   PRINT "A"
127.   PRINT "B"
128.   PRINT "B"
129.   PRINT "C"
130.   PRINT "C"
131.   PRINT AT li#+10,1: "Mossa....."
132.   PRINT "Disco....."
133.   PRINT "Sorg....."
134.   PRINT "Dest....."
135. ENDPROC hanoi'banner
136.
137. //*****************************
138. //** Pulizia video area stack
139. //*****************************
140. PROC clr'stack
141.   PRINT AT li#+2,1: "A"+SPC$(30)
142.   PRINT "A"+SPC$(30)
143.   PRINT "B"+SPC$(30)
144.   PRINT "B"+SPC$(30)
145.   PRINT "C"+SPC$(30)
146.   PRINT "C"+SPC$(30)
147.   PRINT AT li#+10,12: SPC$(10)
148. ENDPROC clr'stack
149.
150. //*****************************
151. //** Pausa arbitraria
152. //*****************************
153. PROC pause(duration) CLOSED
154.   FOR t:=1 TO 75*duration DO NULL
155. ENDPROC pause
156.
157. //*****************************
158. //** Attesa tasto
159. //*****************************
160. PROC waitkey CLOSED
161.   WHILE KEY$=CHR$(0) DO NULL
162. ENDPROC waitkey
163.
164. //*****************************
165. //** Funzioni accessorie
166. //*****************************
167. FUNC src'peg#(mv)
168.   //** Calcolo piolo origine
169.   RETURN ((mv BITAND mv-1) MOD 3)+1
170. ENDFUNC src'peg#
171.
172. FUNC dest'peg#(mv)
173.   //** Calcolo piolo destinazione
174.   RETURN (((mv BITOR mv-1)+1) MOD 3)+1
175. ENDFUNC dest'peg#
```

Il codice è decisamente elementare e la sua organizzazione risulta del tutto intuitiva. Si chiede all'utente il numero di dischi (ma il codice è predisposto anche per mostrare in sequenza le soluzioni per varie dimensioni del problema) e si presentano due tipologie di risoluzione, basate sulle formule e sugli approcci già ampiamente illustrati. Tra la visualizzazione di una mossa e la successiva viene inserito un ritardo (vedi linea 112 e la procedura `pause` definita a partire dalla linea 153) in modo da consentire la lettura: ovviamente è possibile sostituire tale chiamata con la `waitkey` appositamente predisposta, in modo da rendere ancora più interattivo (e potenzialmente più noioso) l'output delle sequenze risolutive, che in questo approccio screen-oriented è pensato esplicitamente per guidare l'utente nella risoluzione del gioco nella sua versione fisica, suggerendo ogni singola mossa.

Come si nota immediatamente, il codice è estremamente compatto e buona parte di esso è dedicata alla visualizzazione, sia pure semplificata al massimo e privata degli immancabili "effetti speciali" dell'epoca con colori e pseudoanimazioni testuali, per tacere del potente sottosistema grafico di COMAL.

L'esempio seguente, sempre rigorosamente "retro", è stato creato al volo dall'Autore in C89 nei primi anni Novanta *ad usum delphini* per un thread su FIDOnet come immediato controesempio alla fastidiosamente falsa affermazione che fosse impossibile risolvere il problema senza ricorsione.

Per l'occasione corrente il codice è stato adeguatamente aggiornato all'edizione più recente di Visual Studio e dello standard C con un minimo di belletto sintattico (dopo sei lustri anche la più meravigliosa delle attrici ha bisogno di incipriarsi un po' di più il naso…) e integrato con una modernissima versione dell'algoritmo **L** sopra riportato, per mostrare nuovamente due distinti approcci razionali loopless/branchless tra i tanti possibili alla soluzione iterativa del problema delle torri di Hanoi.

```
 1. #include <limits.h>
 2. #include <math.h>
 3. #include <stdint.h>
 4. #include <stdio.h>
 5. #include <stdlib.h>
 6. void display(uint16_t **P, const size_t d) {
 7.     size_t i;
 8. #ifndef VERBOSE
 9.     return;
10. #endif
11.     for (i = 0; i < 3; ++i) {
12.         size_t j;
13.         printf("[%u] ", P[i][0]);
14.         for (j = 1; j <= d; ++j) {
15.             printf("%u ", P[i][j]);
16.         }
17.         puts("");
18.     }
19. }
20. void HanoiByRules(const size_t d) {
21.     size_t moves, i;
22.     uint16_t *p;
23.     uint16_t **Pegs;
24.     moves = pow(2, d) - 1;
25.     Pegs = malloc(3 * sizeof(uint16_t *));
26.     p = calloc(3 * (d + 1), sizeof(uint16_t));
27.     for (i = 0; i < 3; ++i) {
28.         Pegs[i] = p;
29.         p += (d + 1);
30.     }
31.     for (i = 0; i < d; ++i) {
32.         Pegs[0][i + 1] = d - i;
33.     }
34.     Pegs[0][0] = d;
35.     printf("************************************************************\n"
36.            "** Approccio con algebra delle mosse e storage simulato.\n"
37.            "** Dischi........: %zd\n** Mosse.........: %zd\n"
38.            "************************************************************\n",
39.            d, moves);
40.     display(Pegs, d);
41.     for (i = 1; i <= moves; ++i) {
42.         uint16_t from, to;
43.         from = (i & i - 1) % 3;
44.         to = ((i | i - 1) + 1) % 3;
45.         printf("%3zd: disco (%u) da %c a %c\n", i, Pegs[from][Pegs[from][0]],
46.                'A' + from, 'A' + to);
```

```
47.         Pegs[to][0] += 1;
48.         Pegs[to][Pegs[to][0]] = Pegs[from][Pegs[from][0]];
49.         Pegs[from][Pegs[from][0]] = 0;
50.         Pegs[from][0] -= 1;
51.         display(Pegs, d);
52.     }
53. }
54. /*
55. ** Implementa l'algoritmo 7.2.1.1 L di Knuth per la generazione loopless
56. ** della "ruler sequence" (OEIS: A001511) che corrisponde alla sequenza
57. ** dei dischi da muovere. Il piolo di destinazione viene determinato con
58. ** una semplicissima espressione di incremento modulare.
59. */
60. void HanoiByGray(const size_t d) {
61.     uint16_t *focus;
62.     size_t i, moves;
63. /** L1. [Initialize.] **/
64.     focus = (uint16_t *)malloc((d + 1) * sizeof(uint16_t));
65.     for (i = 0; i <= d; ++i) {
66.         focus[i] = i;
67.     }
68.     moves = pow(2, d) - 1;
69.     printf("\n************************************************************\n"
70.            "** Binary Reflected Gray Code (ruler sequence A001511)\n"
71.            "** Algoritmo di Knuth 7.2.1.1 L\n"
72.            "** Dischi........: %zd\n** Mosse.........: %zd\n"
73.            "************************************************************\n",
74.            d, moves);
75.     i = 1;
76.     do {
77.         uint16_t from, to, j;
78.         from = (i & i - 1) % 3;
79.         to = ((i | i - 1) + 1) % 3;
80. /** L2. [Visit.] **/
81.         printf("%3zd: disco (%u) da %c a %c\n", i++, focus[0] + 1, 'A' + from,
82.                'A' + to);
83. /** L3. [Choose j.] **/
84.         j = focus[0];
85.         focus[0] = 0;
86.         focus[j] = focus[j + 1];
87.         focus[j + 1] = j + 1;
88.     } while (focus[0] != d);
89. }
90. int main() {
91.     uint16_t i;
92.     puts("** Le torri di Hanoi - soluzioni iterative efficienti **");
93.     for (i = 2; i < 8; ++i) {
94.         HanoiByRules(i);
95.         HanoiByGray(i);
96.     }
97.     return EXIT_SUCCESS;
98. }
99. /** EOF: hanoi.c **/
```

Si nota facilmente che la funzione `HanoiByRules()` fa uso di uno stack simulato tramite una **matrice di interi** $3 \times n$ e usa le formule (9) e (10) per la determinazione dei pioli di partenza e arrivo, ricavando l'identificativo del disco corrente dallo stack simulato. Viceversa, la seconda funzione `HanoiByGray()` **non fa uso di storage** e calcola ad ogni iterazione tutti i parametri della mossa corrente (disco da muovere, sorgente, destinazione) esclusivamente in funzione della variabile di induzione del loop principale, ossia dell'ordinale di mossa.

5.1 Variazioni sul tema.

Dopo l'anteprima del presente articolo sul gruppo RP- RetroProgramming Italia, Alessando Scola (già autore, tra l'altro, di un interessante intervento sul medesimo problema) ha suggerito di trattare brevemente anche una variante del problema, nella quale il ruolo di **scambio** e **destinazione** è prefissato per i pioli B e C rispettivamente (per fissare le idee), a prescindere dalla parità del numero dei dischi. Tra le numerose varianti (che includono anche quelle a quattro pioli, oppure le cosiddette torri di Bucarest, e molte altre) si tratta di una delle più diffuse, spesso confusa con la formulazione originale del problema, ma in realtà ne costituisce una prima facile generalizzazione, come testimoniato uniformemente dalla letteratura.

Le modifiche necessarie a trattare questo caso, in realtà, sono pressoché insignificanti ed hanno un impatto prestazionale praticamente nullo. Tra i numerosi approcci possibili (modifica delle formule generatrici della sequenza, uso di regole, etc.) si sceglie il cammino di minima resistenza, in omaggio al fondamentale principio K.I.S.S. e in considerazione dell'assoluta arbitrarietà nell'assegnazione delle etichette ai pioli.

La soluzione più istituzionale è quindi una mappatura differenziata di A, B e C secondo il verso di rotazione – condizionato ovviamente dal bit meno significativo del numero dei dischi, ossia in ultima analisi dalla **parità** di tale valore.

Ciò si realizza con l'aggiunta di **due** semplicissime linee di codice (con riferimento al linguaggio C, esemplificando per la sola funzione `HanoiByGray()`, e pochissimo cambia anche in COMAL), immediatamente prima della linea 63:

```
63a. const char Pegs[2][3] = {{'A','C','B'}, {'A','B','C'}};
63b. size_t parity = d & 1;
63c. /** L1. [Initialize.] **/
```

Occorre poi solamente modificare la linea 81/82:

```
 81.         printf("%3zd: disco (%u) da %c a %c\n", i++, focus[0] + 1,
 82.                Pegs[parity][from], Pegs[parity][to]););
```

Aggiungiamo solo alcune brevissime considerazioni.

1. Questa generalizzazione consente in realtà di usare **una terna di simboli totalmente arbitrari** per identificare i pioli, oltre a distinguere il caso pari da quello dispari. Molte fonti, infatti, fanno riferimento ai tre pioli A, B, C come *f, r, t* rispettivamente (con audace spicco di fantasia: *f* = from, *t* = to, *r* = remaining).

2. Il codice suggerito si riferisce al caso in cui il piolo d'arrivo sia sempre C e quello di scambio o utilità il B. La regola quindi è che il piolo d'arrivo è il **secondo elemento della prima riga** $Pegs_{1,2}$ della matrice di char (valida per i valori PARI), e il terzo nella seconda riga $Pegs_{2,3}$ – come è evidente, usiamo qui la notazione dell'algebra lineare 1-based per i valori di riga e colonna, anche pensando a COMAL.

3. La variabile ausiliaria `parity` viene introdotta unicamente a titolo prudenziale, in caso di ricorso a compilatori C giurassici o scarsamente ottimizzanti. A rigore, in ambito retroprogramming, potrebbe e dovrebbe essere preceduta da `register` ovunque ciò abbia senso, e può tranquillamente essere espunta quando si ha la certezza che ottimizzazioni del tipo *remove invariant code* sono attivate e funzionanti. Nel caso di implementazione COMAL o BASIC, vale comunque l'approccio più prudenziale ed esplicito che ne garantisce il precalcolo una e una sola volta, a monte del loop principale.

4. Restando al linguaggio C, sarebbe un banale esercizio evitare il ricorso alla matrice di char (che ha comunque una sua utilità e solide motivazioni) usando invece una espressione booleana per il calcolo della label, possibilmente associato ad un operatore ternario. Tale soluzione, probabilmente più elegante e certamente più low level per taluni aspetti, non è necessariamente più efficiente e di certo perde il carattere di generalità dato dall'uso di un array di simboli arbitrari per l'etichettatura dei pioli, usabile anche in numerose altre varianti del problema.

6 Conclusioni.

L'importanza **teorica** della ricorsione è talmente ovvia da risultare completamente fuori discussione: al cuore dell'informatica teorica troviamo infatti la teoria della calcolabilità, che fino a pochi anni fa si chiamava esplicitamente **teoria della ricorsione** e che in ogni caso si occupa delle cosiddette funzioni (primitive) ricorsive, ossia tutte le funzioni $f: \mathbb{N} \mapsto \mathbb{N}$ "intuitivamente calcolabili" oggetto della tesi di Church-Turing. Inoltre le formulazioni ricorsive godono di una inerente eleganza e concisione, che le rende particolarmente amate in ambito discretistico.

Ribadita questa lapalissiana verità di fatto, deve però risultare altrettanto ovvio e incontrovertibile che nell'informatica applicativa la ricorsione è il Male con la stragrande maggioranza dei linguaggi di programmazione, degli standard interni di chiamata di funzione e passaggio di parametri, delle architetture hardware. Tutti gli standard cogenti che regolano la progettazione dei sistemi embedded ad altissima affidabilità, a partire dal basilare MISRA/C, vietano l'uso della ricorsione senza mezzi termini.

Lasciando a margine i linguaggi funzionali puri come Haskell (particolarmente caro all'Autore), che ultimamente hanno goduto di qualche attenzione in più ma che rimangono una nicchia nella nicchia, il fatto che negli ultimi tre lustri il problema prestazionale e di ingordigia di risorse nell'uso sconsiderato di ricorsione appaia **mitigato** (ma mai eliminato) grazie alla esorbitante potenza delle piattaforme mainstream ed alla sofisticazione dei compilatori di nuova generazione non cancella decenni di sforzi teorici nell'area algoritmica nota come "eliminazione della ricorsione" citata in ogni serio manuale e della

quale il notissimo Robert Sedgewick è stato un pioniere, dimostrando peraltro grazie ai suoi sforzi in tale settore il limite inferiore assoluto di complessità per un algoritmo di ordinamento arbitrario.

Con queste semplici note e i relativi esempi di codice, sia pure ridotti al minimo indispensabile, si è in ogni caso smantellata per l'ennesima volta una leggenda urbana particolarmente dura a morire, a causa di una associazione quasi pavloviana nella didattica tra il problema delle torri di Hanoi e la ricorsione, che induce in troppi programmatori la convinzione errata che tale problema non sia risolvibile senza il ricorso a tale tecnica. Nel fare ciò si è anche cercato di mantenere le necessarie spiegazioni algebriche chiare, accessibili e corrette - tenendosi equidistanti dagli eccessi di astrazione di numerosi testi e dalla cialtroneria facilona imperante sul web. L'Autore si augura che lo sforzo non sia stato vano e che il risultato sia comunque piacevole e interessante.

Riferimenti bibliografici essenziali

Allouche, Jean-Paul, and Jeffrey Shallit. *Automatic sequences: Theory, applications, generalizations.* E-book edition. Cambridge, UK: Cambridge University Press, 2009.

Gardner, Martin. *The new Martin Gardner mathematical library: Hexaflexagons, probability paradoxes, and the tower of Hanoi.* Cambridge University Press, 2008.

Knuth, Donald E. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations.* Addison Wesley, 2005.

Mütze, Torsten. "COMBINATORIAL GRAY CODES—AN UPDATED SURVEY." *arXiv.* Luglio 30, 2024. https://arxiv.org/pdf/2202.01280.

Scorer, R. S., P. M. Grundy, and C. A. B. Smith. "Some Binary Games." *The Mathematical Gazette* 28, no. 280 (1944): 96-103.

Smillie, Keith W. "Recursive and iterative algorithms for the tower of hanoi puzzle." *ACM SIGAPL APL Quote Quad* 4 (1973): 12-14.