

TITOLO: PROVVISORIO

(1998-2016)

C'era una volta, nel millennio scorso, un giovane Programmatore¹ decisamente molto appassionato. Evidentemente un programmatore piuttosto prolifico e curioso, se i suoi colleghi su FIDOnet (in un'era in cui molti, se non tutti, lavoravano in Assembly) lo avevano soprannominato Assembly Wizard, e in seguito Master Assembly Wizard dopo il battesimo della milionesima linea di codice.

Numerosi programmatori nei decenni hanno compreso e confermato che nei post FIDOnet chiusi in calce dall'acronimo M.A.W. 1968, e poi negli interventi sui principali forum tecnici firmati con tale nickname (spesso in qualità di Moderatore), c'erano contenuti interessanti, stimolanti, inusuali sia in lingua inglese che italiana. Molti di quei contenuti sono entrati di diritto a far parte delle FAQ e BOA FIDOnet, e sono poi travasati al volgere del millennio in vari blog (durati più o meno quanto le piattaforme che li ospitavano), assai prima che nascessero quelli che oggi sono i poli internazionali di aggregazione di riferimento sul web per chi si interessa di matematica e programmazione.

Uno dei blog più stabili e longevi, decisamente il più letto come testimonia il numero di visite, è stato quello creativamente denominato “Titolo provvisorio” negli anni tra il 1998 e il 2016, ospitato in successione su due diverse piattaforme.

Nonostante le ridotte possibilità di formattazione supportate, con particolare riguardo a formule LaTeX/MathML e immagini incorporate, è stato possibile pubblicare numerosi articoli su alcuni dei temi che costituiscono i cavalli di battaglia dell'Autore: logica, matematica discreta e combinatoria, algebra, programmazione avanzata in vari linguaggi, low level, retroprogramming.

Per ovviare alle limitazioni accennate sopra, alcuni anni dopo la pubblicazione sul blog vari articoli sono stati ripresi, rielaborati e riformattati, ciclicamente in diversi momenti storici e con un ampio set di strumenti eterogenei, da LaTeX a Word, mantenendo i contenuti originali ed espandendoli. Gli articoli più lunghi nascono in realtà come integrazione organica di varie blog entry che a più riprese negli anni hanno toccato il medesimo tema, spesso in occasione di nuove pubblicazioni scientifiche o domande ricorrenti sul tema apparse nei numerosi forum moderati dall'Autore.

Sulla base del massiccio feedback specialistico ricevuto, come pure delle spontanee manifestazioni di entusiasmo da parte di studenti di ogni ordine e grado, lo stile colloquiale e divulgativo che caratterizza l'esposizione dei temi anche più ostici risulta essere notevolmente apprezzato dai programmatori e aspiranti tali.

Il tono informale, latamente didascalico, nel quale risuonano le buone letture di alta divulgazione scientifica, sovente anche intriso di velata ironia tipicamente toscana, cerca di rendere comprensibile la materia di volta in volta trattata anche ad un ideale studente degli ultimi anni di scuola superiore o matricola universitaria, come pure al practitioner della programmazione applicativa ormai da molti anni lontano dai testi scolastici e dalla teoria.

Questa scelta stilistica, o piuttosto inclinazione divulgativa, caratterizza costantemente l'esposizione e sottende in modo piuttosto naturale (stanti anche le notevoli limitazioni della piattaforma originale) l'eliminazione di quasi tutte le dimostrazioni, le citazioni puntuali da testi e articoli scientifici ridotte all'essenziale, la costante prevalenza della chiarezza e dell'intuizione sulla ricerca esasperata di rigore lessicale e formale. Questi ultimi aspetti vengono comunque in parte recuperati nelle revisioni in PDF, ma confinati in note a margine e riferimenti bibliografici disseminati generosamente per ogni dove, senza appesantire eccessivamente la trattazione.

Lo scopo primario rimane sempre quello di incuriosire il lettore, stimolando approfondimenti autonomi, trasmettendo ove possibile nozioni e concetti complessi in modo semplice e lineare, sempre supportati da numerosi esempi di codice in linguaggio C (spesso affiancato da linguaggi più o meno esotici o “retro”: ad esempio Python, J, COMAL).

¹ All'epoca nessuno avrebbe mai usato il termine “sviluppatore”, se non forse per riferirsi ad una specializzazione di nicchia della già oscura figura del particolarista meccanico: un disegnatore tecnico che, ingobbito sul tecnigrafo e perennemente sporco d'inchiostro di china, “sviluppava” appunto i singoli disegni di dettaglio e gli esecutivi dai macroprogetti di ingegneria meccanica. Principalmente si occupava proprio degli “sviluppi in piano” dei particolari in lamiera piegata.

Tra i numerosissimi temi affrontati su ML e forum nei decenni figurano argomenti ormai museali (ad esempio hardware anni Ottanta, protocolli seriali e paralleli ormai desueti, sistemi embedded e RTOS hard realtime, low level DOS e brand Unix, elettronica analogica e mixed signal, logiche cablate e programmabili...) e molto altro estremamente specifico, come generazione combinatoria, topologia e geometria computazionali, calcolo matriciale, funzioni booleane: come pure argomenti di interesse più sociopsicologico e talora psichiatrico, tra i quali noti tormentoni del tipo “Ho scritto un virus per imparare”, “Dopo il primo hello world e la sommatoria di un array di interi voglio scrivere il mio sistema operativo”, “Come ottimizzare 36 loop annidati per generare le combinazioni del lotto per mia nonna?” e altre amenità di questo genere, tutte rigorosamente prese da esempi reali.

Avendo per lapalissiani motivi scartato a priori gli argomenti accennati sopra, trovano invece spazio in questa raccolta alcuni degli articoli divulgativi che hanno raccolto i maggiori consensi dei lettori e il più elevato numero di feedback:

- il problema dei **matrimoni stabili**, con un algoritmo risolutivo di tipo esaustivo (ingiustamente snobbato nella vasta area della letteratura algoritmica, forse a causa della sua relativa sofisticazione), analizzato in tutti i suoi passaggi salienti e corredata di un esempio in linguaggio C per illustrare al più basso livello possibile le peculiarità delle strutture dati utilizzate per ottenere un impressionante incremento prestazionale rispetto alle soluzioni classiche;
- il **problema dei Ménage**, per restare in tema “matrimoniale”, un classico ludomatematico proposto nel 1891 da Édouard Lucas nel suo altrettanto classico “Théorie des nombres”, diventa un eccellente pretesto per illustrare ben sei distinti algoritmi combinatori (e sette differenti implementazioni in C), di fatto i migliori esistenti, concepiti nell’arco di oltre mezzo secolo;
- le **partizioni di numeri naturali**, argomento solo apparentemente banale ma che cela numerose insidie, ripreso numerose volte sul blog, che come al solito diventa occasione per discutere alcuni dei più efficienti algoritmi generatori ad oggi noti;
- i **numeri ciclici**, argomento immancabile nella divulgazione (ludo)matematica, ripreso al volo ed espanso da un topic su un forum, con implementazioni in C e Python che ne illustrano alcune rilevanti proprietà e mostrano alcuni cardini della generazione combinatoria;
- la teoria dei **tornei round robin**, con una solida spiegazione dell’algoritmo di Berger (anzi, di Schurig...) e relativa implementazione in C;
- il problema ottocentesco delle **Torri di Hanoi**, anch’esso proposto dal già citato Édouard Lucas ed assurto (sebbene nessuno sembri conoscerne bene il motivo) al ruolo di esempio standard nella didattica caratterizzata dal feticcio della ricorsione: ovviamente, conoscendo l’inclinazione dell’Autore, qui viene risolto con varie strategie iterative, note da circa mezzo secolo, enormemente più efficienti, a bassissimo impatto computazionale e di una semplicità disarmante;

...e poi raddoppio ricorsivo, metodi formali, Mersenne twister, transputer, i linguaggi matriciali APL e J, auguri per il compleanno di Ada Lovelace e molto altro ancora.

Rileggendo questi articoli ci si rende conto che, a distanza di molti anni, rimangono ancora perfettamente attuali e godibili. L’importanza dei problemi trattati, la sostanziale assenza di rivoluzionarie novità teoriche dall’epoca della pubblicazione, la ricerca della massima chiarezza espositiva, l’amore per i dettagli e le note storiche rendono ancor oggi degno di nota lo sforzo dell’Autore di creare questi testi divulgativi, dai più brevi ai più lunghi (che oggi potrebbero tranquillamente costituire un elaborato finale in qualche corso di specializzazione, aggiungendo solo quel pizzico di formalismo in più normalmente richiesto in ambito accademico).

«Cielo... mio marito!»

Sommario

La celeberrima battuta tratta da “Tailleur pour dames” del grande Georges Feydeau, una delle più esilaranti (e copiate) *pochade* incentrate sul tema delle relazioni extraconiugali, si presta splendidamente ad introdurre il non meno famoso problema combinatorio noto come “Problema dei matrimoni stabili”. In questo articolo proponiamo come primo assaggio una succinta descrizione del problema così come posto originariamente da Gale & Shapley nel 1962 e una elegante implementazione in COMAL 80, per restare nell’ambito retroinformatico.

«*The difference between the poet and the mathematician is that the poet tries to get his head into the heavens, while the mathematician tries to get the heavens into his head.*»
(G. K. Chesterton, 1874-1936)

1 Introduzione.

Sfatiamo subito un mito: scorrendo l’ormai vastissima lista delle applicazioni nate attorno a questo problema e sue variazioni ([Man13], pagg. 30 ÷ 31), si vede come in realtà l’uso di tali algoritmi in ambito matrimoniale e simili (dating online, etc.) è sporadico, pressoché inesistente. Come spessissimo avviene, il riferimento usato nel nome del problema è volutamente metaforico e serve unicamente a creare uno spiritoso spunto mnemonico e situazionale per quello che è un serissimo problema di assegnazione di risorse in operations research, nato attorno ad una istanza importante (l’assegnazione di studenti ai college universitari e, indipendentemente, dei medici tirocinanti alle cliniche) ed evolutosi poi in una serie di generalizzazioni con un vastissimo campo applicativo: dall’assegnazione di posti istituzionali in enti pubblici e parastatali di ogni genere alla logistica, dalla pianificazione della produzione all’assegnazione di risorse ai centri di costo, alla gestione di affiancamenti nell’ambito sanitario o militare, eccetera.

2 Matrimoni stabili: un primo sguardo al problema.

In cosa consiste esattamente il problema Stable Marriage (SM)? Esistono numerose varianti, ma nella formulazione originale, dovuta a Gale e Shapley nel 1962 ([GS62]), si hanno due distinti gruppi (“uomini” e “donne”), di pari dimensione. Ciascun membro di ogni gruppo esprime in una lista completa le proprie preferenze per “l’altro sesso”, ordinandole - per fissare le idee - in modo decrescente, dalla più alla meno desiderabile. In pratica, ad ogni elemento di uno dei due insiemi si associa una particolare permutazione dell’altro insieme. Si procede quindi a formare le “coppie”, seguendo gli elenchi di preferenze: la soluzione ottenuta è detta “matrimonio stabile” se non esistono due individui, uno per insieme, i quali preferiscono reciprocamente l’altro individuo al proprio partner attuale. L’algoritmo pubblicato nel 1962 (che chiameremo GS) è basato su semplicissimi passaggi detti “proposte”, che vengono accettate o rifiutate secondo il semplice confronto posizionale tra il partner attuale e il proponente nella scala delle priorità della parte che riceve la proposta. Se la ricevente preferisce il proprio partner attuale al proponente, scatta il manzoniano “Questo matrimonio non s’ha da fare” e si passa alla successiva preferenza. L’idea tipica, nelle intenzioni di Gale & Shapley, è che l’insieme M degli “uomini” (es. studenti) si propone e quello delle “donne” W (es. college, cliniche) valuta le proposte, ma ovviamente nulla impedisce di eseguire l’algoritmo seguendo la convenzione opposta. Il principale merito di Gale & Shapley è quello di avere dimostrato che:

- 1) Qualunque istanza del problema SM ammette almeno una soluzione stabile;
- 2) Tale soluzione ha la proprietà di essere proponent-optimal, ossia garantisce a ciascun membro dell’insieme proponente il migliore accoppiamento in assoluto rispetto a qualsiasi altra eventuale soluzione stabile (e non). Questa proprietà è la cosiddetta ottimalità paretiana debole.

Il tutto usando un algoritmo di estrema semplicità che, dati due insiemi non vuoti M e W di cardinalità n e le relativi matrici di preferenze, genera la soluzione male-optimal (o female-optimal) in $O(n^2)$ al caso peggiore. L’analisi del caso medio di esecuzione avrebbe diritto ad un intero articolo dedicato, ma ci limitiamo qui a presentarne il risultato che coinvolge la definizione di numero armonico: $H_n = \sum_{k=1}^n k^{-1} = \gamma + \psi_0(n+1)$ dove H_n è appunto

l'ennesimo numero armonico¹. Detto questo, il numero medio di proposte nell'algoritmo GS è dato da $n \cdot H_n + O((\log n)^4)$. Ne consegue che la complessità media è dell'ordine di $\Theta(n \log n)$.

La soluzione ottenuta può essere ovviamente espressa in vari modi. Sostanzialmente essa consta di un elenco di n coppie (m, w) con $m \in M$ e $w \in W$. Si abbiano i seguenti elenchi di preferenze, rispettivamente maschili (a sinistra) e femminili, in cui ogni membro dei due insiemi è codificato da un piccolo intero positivo:

$$\begin{array}{ccccccccc} 1 : & 4 & 1 & 2 & 3 & 1 : & 4 & 1 & 3 & 2 \\ 2 : & 2 & 3 & 1 & 4 & 2 : & 1 & 3 & 2 & 4 \\ 3 : & 2 & 4 & 3 & 1 & 3 : & 1 & 2 & 3 & 4 \\ 4 : & 3 & 1 & 4 & 2 & 4 : & 4 & 1 & 3 & 2 \end{array}$$

Intuitivamente, la riga 1 della matrice a destra ci dice che la signora #1 preferisce nell'ordine il signor 4, poi come seconda scelta il signor 1, e via di seguito in ordine discendente di preferenza.

La soluzione stabile male-optimal generata da GS è il seguente elenco di coppie che segue la convenzione (m, w) : $\{(1, 4), (2, 3), (3, 2), (4, 1)\}$. Tale elenco può ovviamente essere espresso come una permutazione in notazione standard a due linee:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix}$$

Leggendo la notazione da sinistra a destra e dall'alto in basso, abbiamo le coppie $(1, 4), (2, 3), (3, 2), (4, 1)$. In questa notazione, la permutazione identità della prima riga rappresenta l'insieme proponente nell'ordine naturale. Ne consegue che si può usare anche la notazione posizionale 1-line, come per qualsiasi altra permutazione, senza perdita di informazione: la notazione 4321 rappresenta esattamente le medesime coppie, dove agli uomini è implicitamente assegnata la posizione 1..n crescente verso destra e le relative partner vengono specificate esplicitamente.

Tuttavia, la natura combinatoria elementare della soluzione non ci è di grande aiuto computazionalmente. Trovare tra le possibili permutazioni dell'insieme W (o M , se del caso) la soluzione ottimale per esaustione applicando il paradigma del “generate and test” è una via del tutto impraticabile anche per problemi di dimensioni assai modeste. Tale soluzione può anche essere vista nella sua natura di relazione binaria, ossia un sottoinsieme del prodotto cartesiano $M \times W$ tra i due insiemi dati, il quale prodotto in soldoni non è altro che l'elenco di tutte le possibili n^2 coppie ordinate formate prendendo un solo elemento per volta da ciascun insieme di partenza. Assolutamente banale dal punto di vista combinatorio, in ultima analisi: ma come spesso avviene, gli algoritmi necessari ad ottenere una enumerazione esaustiva di tali soluzioni, come pure quelli dedicati a trovare particolari tipologie di soluzione, erano ben lunghi dal vedere la luce ai tempi di Gale & Shapley, sono quasi tutti caratterizzati da complessità computazionale molto elevata e/o richiedono un approccio del tutto peculiare, con l'ausilio di strutture dati non banali.

3 Gli anni Settanta e Ottanta.



Figura 1: Il KDF-9 EEC a Newcastle

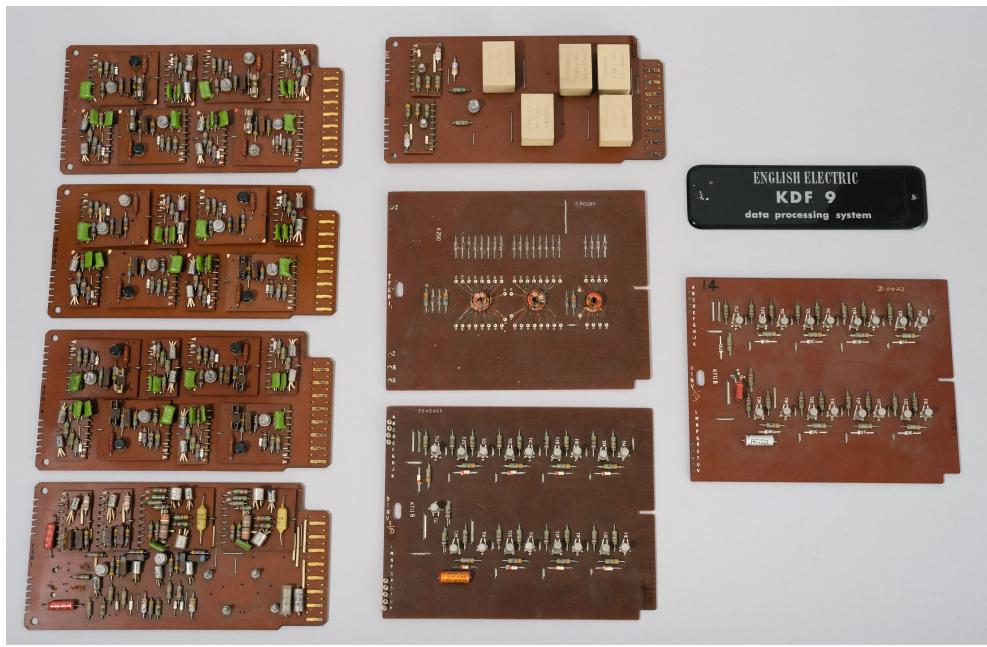
luzioni. Negli stessi anni anche D. E. Knuth aveva proposto un algoritmo sostanzialmente simile, con analoghe prestazioni.

Pochi anni dopo la pubblicazione dell'articolo di Gale & Shapley, McVitie e Wilson nel 1970/71 [MW70, MW71b, MW71a] presentano vari algoritmi in ALGOL60, il più importante dei quali consente la generazione esaustiva di tutte le soluzioni stabili ad una data istanza di SM (per la versione del problema generalizzata al caso di liste di dimensioni disuguali, che come caso particolare risolve l'istanza standard). Sfortunatamente, per motivi di copyright ancora vigente su tali articoli, non sarà possibile qui approfondire come meriterebbe l'argomento presentando il sorgente originale integrale. Tale algoritmo ricorsivo, pur avendo prestazioni decisamente non entusiasmanti, ha una sua notevole importanza storica in quanto costituisce il primo tentativo di fornire una generazione esaustiva delle so-

¹Nell'espressione analitica più a destra, γ è la costante di Euler-Mascheroni e ψ_0 è la funzione digamma, ossia la derivata logaritmica della funzione gamma, a sua volta estensione del fattoriale ai numeri reali.

Occorreranno quasi venti anni prima che tale sforzo sia superato, con la pubblicazione di alcuni articoli poi confluiti nella fondamentale monografia di Gusfield e Irving [GI89] i quali - grazie ad un attento studio della natura algebrica dello spazio delle soluzioni come reticolo e come anello d'insiemi, e suggerendo un uso accorto di strutture dati *order-preserving* in grado di garantire contemporaneamente sia l'accesso diretto indicizzato, sia cancellazioni in $O(1)$ - hanno proposto l'algoritmo ancora attualmente in uso per la generazione esaustiva anche su istanze di grandi dimensioni, con prestazioni in $O(n^2 + n \cdot |TSM|)$, dove TSM è l'insieme di tutte le soluzioni stabili: praticamente il medesimo ordine di grandezza del GS (che in quel tempo è in grado però di generare solo una singola soluzione, non tutte). Avremo modo di approfondire altrove come si possano ottenere simili incrementi prestazionali.

Altro aspetto caratteristico (e divertente!) dell'algoritmo di McVitie e Wilson, che lo rende ancora più degno di studio e di interesse, è la sua originale implementazione su un KDF-9 a transistor della English Electric Computers (poi English Electric LEO Marconi Computers), installato presso l'Università di Newcastle: questo è genuino retrocomputing, e del migliore! Di tale storico elaboratore, dotato di memorie a nuclei magnetici, esiste un emulatore scritto in Ada e dotato di una ricchissima documentazione, col quale l'Autore si è a lungo divertito.



In foto le principali schede a transistor del KDF-9.

Come già accennato, a partire dalla metà degli anni Ottanta alcune pubblicazioni (principalmente dovute a Gusfield, Irving e Leather: [IL86, ILG87, GILS87, Gus87]) avevano risvegliato l'interesse generale verso il problema, stimolando numerosi articoli divulgativi sulle principali riviste internazionali di programmazione applicativa e - come immediata conseguenza - anche diverse implementazioni di studio in vari linguaggi di alto livello (tra i quali Ada, C, Clipper, COMAL) da parte dell'Autore. In questo articolo ci concentreremo su una di tali implementazioni in COMAL 80 V. 2.01 per Commodore 64, restando nell'ambito del retrocomputing con un linguaggio che sta riscuotendo sempre maggiori attenzioni e consensi tra i lettori.

Un'ultima nota. La prima volta che l'Autore si è occupato sistematicamente del problema in contesto accademico, a fine anni Ottanta, esistevano decine di pubblicazioni al riguardo, SM era menzionato in almeno mezza dozzina di testi di algoritmica a partire dagli anni Settanta e oltre alla già citata Gusfield-Irving appena stampata vi erano altre due monografie dedicate, una delle quali [RS90] fondamentalmente incentrata solo sugli aspetti game-theoretic e l'altra, dovuta al venerabile D.E. Knuth [Knu97], edita solo in francese fino al 1997. Sei lustri dopo il numero di pubblicazioni e di testi di algoritmica che fanno menzione del problema è pressoché raddoppiato, tuttavia si deve rilevare che molte delle questioni poste nelle principali monografie storiche (alle quali se ne è aggiunta una quarta [Man13] pochi anni fa) non hanno ancora trovato soluzione e il problema rimane ricco di spunti di ricerca e applicativi. Ma avremo modo di parlare di questi aspetti in seguito.

4 L'algoritmo di Gale & Shapley.

La versione più elementare, riportata nei testi di algoritmica e in numerosi articoli divulgativi, è anche la più utilizzata per le implementazioni. Si tratta di un algoritmo iterativo di estrema semplicità, che però garantisce un risultato per nulla intuitivo leggendo il problema. Il nondeterminismo introdotto dalla mancanza di un particolare ordine dei proponenti è un aspetto che è stato dato per scontato negli articoli scientifici degli anni Sessanta e Settanta: in realtà l'ordine dei proponenti è del tutto irrilevante, perché l'algoritmo genera sempre e comunque la medesima soluzione male-optimal (o female-optimal, invertendo banalmente le tabelle di preferenze), come non mancano di notare Gusfield e Irving ([GI89], pagg. 9 ÷ 10).

```
procedure STABLE_MARRIAGE
    assign each person to be free;
    while some man  $m$  is free do
        begin
             $w := next(m)$ ;
            if  $free(w)$  then
                assign  $m$  and  $w$  to be engaged {to each other};
            else
                if  $w$  prefers  $m$  to her fiancé  $m'$  then
                    assign  $m$  and  $w$  to be engaged;
                    free  $m'$ ;
                else
                     $w$  rejects  $m$  { $m$  remains free};
        end;
        output the  $n$  engaged pairs;
```

Il funzionamento dell'algoritmo è decisamente intuitivo. La funzione di comodo $next(m)$ fornisce la donna w immediatamente successiva all'attuale partner (o la prima della lista, se siamo alla prima esecuzione) nell'elenco di preferenze dell'uomo m , il proponente. La funzione $free(w)$, in modo del tutto ovvio, controlla se w ha già un partner. In tale caso, si confrontano le posizioni del proponente m e dell'attuale partner m' nell'elenco delle preferenze di w e si agisce di conseguenza, realizzando la coppia che assegna a w il partito relativamente migliore secondo le preferenze di lei e libera il meno desiderabile, che così potrà proporsi ad altre potenziali partner nelle successive iterazioni. Importante rilevare che tali comparazioni, così come l'implementazione delle funzioni ausiliarie, possono e devono essere realizzate in $O(1)$ senza ricorso a farraginose ricerche lineari negli array principali, data anche la dimensione media dei problemi generalmente trattati con questa famiglia di algoritmi, dell'ordine di migliaia di proponenti. Altrettanto importante notare che i controlli di bound sugli indici sono sostanzialmente superflui, in quanto Gale & Shapley hanno dimostrato che l'algoritmo termina sempre senza che alcun proponente oltrepassi il termine della propria lista di preferenze, restando così celibe.

L'idea vincente dell'algoritmo si può spiegare in modo semplicissimo: ogni conflitto di preferenze tra i proponenti viene risolto in modo deterministico, scegliendo tra i due quello che risulta più vicino alla cima della lista preferenziale della donna «contesa». In questo modo uno o più proponenti dovranno «accontentarsi» di una partner che preferiscono meno, come mostra l'esito finale dell'esempio che vedremo tra breve, ma che comunque (è dimostrabile) rimane la loro migliore scelta in assoluto tra tutte le possibili soluzioni stabili.

Chiudiamo questa breve sezione con una curiosità storica: nelle implementazioni fino a tutti gli anni Settanta, veniva usato il bit del segno per indicare se un individuo m o w risultava coniugato, senza utilizzare un array ausiliario. Un tipico «trucco» di retroprogrammazione che ha caratterizzato, prima ancora dell'era dei PET e degli home computer, un'intera epoca di programmi in ALGOL e FORTRAN, condizionata dai costi delle memorie fisiche e dalla loro proverbiale scarsità.

5 L'implementazione in COMAL 80.

Si presenta una implementazione di esempio in COMAL 80 (provata con la versione 2.01 su cartridge) per Commodore 64, versione semplificata e ridotta di una delle prime implementazioni dell'Autore, datata 1984. Per verificare facilmente la correttezza dell'implementazione, si sono scelte le tabelle di preferenze dell'esempio di dimensione 8 da [GI89], pag. 12 che generano la soluzione stabile male-optimal seguente, ordinando al solito le coppie secondo la convenzione (m, w) :

$$\{(1,5), (2,3), (3,8), (4,6), (5,7), (6,1), (7,2), (8,4)\}$$

```
/******
// save stable-marriage
*****
```

```
DATA 5,7,1,2,6,8,4,3
DATA 2,3,7,5,4,1,8,6
DATA 8,5,1,4,6,2,3,7
DATA 3,2,7,4,1,6,8,5
DATA 7,2,5,1,3,6,8,4
DATA 1,6,7,5,8,4,2,3
DATA 2,5,7,6,3,4,8,1
DATA 3,8,4,5,7,2,6,1
```

```
DATA 5,3,7,6,1,2,8,4
DATA 8,6,3,5,7,2,1,4
DATA 1,5,6,2,4,8,7,3
DATA 8,7,3,2,4,1,5,6
DATA 6,4,7,3,8,1,2,5
DATA 2,8,5,3,4,6,7,1
DATA 7,5,2,1,8,6,4,3
DATA 7,4,1,5,2,3,6,8
```

```
DIM male' pref(8,8)
DIM female' pref(8,8)
DIM male' lut(8,8)
DIM m' coupled(8)
DIM w' coupled(8)
DIM stack(8)
stack' ptr:=0
```

PAGE

```
FOR i:=1 TO 8 DO
  FOR j:=1 TO 8 DO
    READ male' pref(i,j)
  ENDFOR j
  push(i)
ENDFOR i
```

```
FOR i:=1 TO 8 DO
  FOR j:=1 TO 8 DO
    READ female' pref(i,j)
    male' lut(female' pref(i,j),i):=j
  ENDFOR j
ENDFOR i
```

disp 'data

```
WHILE stack' ptr>0 DO
  pop(man)
  m' coupled(man):+1
  woman:=male' pref(man,m' coupled(man))
  IF w' coupled(woman)<1 THEN
    w' coupled(woman):=man
  ELSE
```

```

IF male' lut (man,woman)<male' lut (w' coupled (woman) ,woman) THEN
    push(w' coupled (woman))
    w' coupled (woman):=man
ELSE
    push(man)
ENDIF
ENDIF
ENDWHILE

disp ' solution

END "Fine lavoro"

PROC disp ' data
    PRINT "Uomini:           Donne: "
    FOR i:=1 TO 8 DO
        PRINT i ,": ",
        FOR j:=1 TO 8 DO
            PRINT male' pref(i,j),", "
        ENDFOR j
        PRINT AT i+1,21: i ,": ",
        FOR j:=1 TO 8 DO
            PRINT female' pref(i,j),", "
        ENDFOR j
        PRINT
    ENDFOR i
ENDPROC disp ' data

PROC disp ' solution
    PRINT AT 12,1: "Stable marriage (m,w):"
    FOR i:=1 TO 8 DO
        w:=male' pref(i,m' coupled(i))
        PRINT AT 12+i,1: "(" ,i ,",",w,")"
        PRINT AT i+1,2+2*m' coupled(i): CHR$(158),w
        PRINT AT w+1,22+2*male' lut (i,w): CHR$(159),i ,CHR$(154)
    ENDFOR i
    PRINT AT 20,1: CHR$(154)
ENDPROC disp ' solution

PROC push(d)
    stack'ptr:=1
    stack(stack'ptr):=d
ENDPROC push

PROC pop(REF d)
    d:=0
    IF stack'ptr>0 THEN
        d:=stack(stack'ptr)
        stack'ptr:=1
    ENDIF
ENDPROC pop

```

Il listato, sebbene privo di commenti, risulta del tutto autoesplicativo. Si è scelto di implementare uno stack nel modo più classico, tramite l'array denominato (con audace spicco di fantasia) `stack()` e la singola variabile `stack'ptr`. Il valore 0 è in questo caso perfetto come guard value, poiché in COMAL gli indici partono sempre da 1. Gli array `male'pref()` e `female'pref()` contengono, senza alcuna sorpresa, le liste delle preferenze dei due sessi,

mentre `male'lut()` è utilizzato per il reverse lookup in tempo costante, poiché per ogni possibile coppia (m, w) ci fornisce la posizione di m nella lista delle preferenze di w , rendendo possibile il confronto immediato tra la posizione del proponente e quella dell'attuale partner, come richiesto dall'algoritmo.

Ultima banale nota: l'array `w'coupled()` ci dice in modo immediato chi è attualmente il partner di w per ogni $w \in W$ le sue locazioni contengono ciascuna un piccolo intero $1 \leq m \leq 8$ che identifica il partner, mentre l'array `m'coupled()`, come è evidente dal codice, è utilizzato in modo leggermente diverso. Il contenuto di ciascuna cella è sempre un piccolo intero non negativo, ma usato come indice relativo alla posizione di w nella lista delle preferenze di m . In questo modo, ad esempio, considerando la coppia $(m, w) = (1, 5)$ si avrà che `m'coupled(m)=1` in quanto `male'pref(m, 1)=5`, e così via per ogni altra coppia.

A proposito di array, si ricordi solo che essi sono implicitamente inizializzati a zero, così come le variabili in genere. Si noti ancora una volta come il linguaggio è un po' proliso (come ALGOL, COBOL o FORTRAN) ma estremamente chiaro, leggibile, ordinato. Il risultato finale è visibile nello screenshot che segue, dove i partner sono elencati esplicitamente ed evidenziati con colori diversi nelle matrici di preferenze dalla procedura `disp'solution`, mostrando la collocazione relativa dei partner in ciascuna lista di preferenze per stimolare doverose riflessioni nel lettore sul significato profondo dell'ottimalità paretiana dell'algoritmo.

```

Uomini:
1: 5 7 1 2 6 8 4 3
2: 2 3 5 4 4 6 3 1
3: 8 5 1 4 6 8 5 4
4: 3 2 7 4 1 6 8 5
5: 7 2 5 1 3 6 8 4
6: 1 6 7 5 8 4 2 3
7: 2 5 6 7 5 8 4 2
8: 3 8 4 5 7 2 6 1

Donne:
1: 5 6 3 7 6 5 2 4
2: 6 6 6 5 5 2 1 3
3: 1 5 6 2 4 1 5 6
4: 8 7 3 3 4 1 5 6
5: 6 4 7 2 3 4 1 5
6: 5 5 5 4 3 4 6 7
7: 7 5 2 2 1 3 6 4
8: 7 4 1 5 2 3 6 8

Stable marriage (m,w):
(1,5)
(2,3)
(3,8)
(4,6)
(5,1)
(6,2)
(7,2)
(8,4)

Fine lavoro

```

6 Conclusioni.

Si è presentato in estrema sintesi uno dei problemi combinatori più longevi e famosi, con qualche fondamentale riferimento alle principali tappe nello sviluppo delle soluzioni fino a tutti gli anni Ottanta. Per un più completo orientamento nella vasta letteratura esistente si rimanda senz'altro il lettore interessato alle monografie già citate, alla loro bibliografia ed al più recente survey disponibile: [IM08].

Si è quindi fornita una completa implementazione in COMAL 80 2.01 per Commodore 64 che consente di esemplificare le caratteristiche e la grande eleganza di tale linguaggio evoluto, rendendone trasparente il funzionamento grazie alla semplicità dell'algoritmo. La duplice speranza dell'Autore è che da un lato i lettori siano incuriositi dal problema e dalle sue numerose varianti, dall'altro che apprezzino sempre meglio il linguaggio e la sua grande espressività.

Riferimenti bibliografici

- [GI89] Dan Gusfield and Robert W. Irving, *The stable marriage problem: Structure and algorithms*, MIT Press, Cambridge, MA, USA, 1989.
- [GILS87] Dan Gusfield, Robert Irving, Paul Leather, and Michael Saks, *Every finite distributive lattice is a set of stable matchings for a small stable marriage instance*, Journal of Combinatorial Theory, Series A 44 (1987), no. 2, 304–309.

- [GS62] D. Gale and L. S. Shapley, *College admissions and the stability of marriage*, The American Mathematical Monthly **69** (1962), no. 1, 9–15.
- [Gus87] Dan Gusfield, *Three fast algorithms for four problems in stable marriage*, SIAM Journal on Computing **16** (1987), no. 1, 111–128.
- [IL86] Robert W. Irving and Paul Leather, *The complexity of counting stable marriages*, SIAM Journal on Computing **15** (1986), no. 3, 655–667.
- [ILG87] Robert Irving, Paul Leather, and Dan Gusfield, *An efficient algorithm for the "optimal" stable marriage*, J.ACM **34** (1987), 532–543.
- [IM08] Kazuo Iwama and Shuichi Miyazaki, *A survey of the stable marriage problem and its variants*, Proceedings of the International Conference on Informatics Education and Research for Knowledge-Circulating Society (Icks 2008) (USA), ICKS08, IEEE Computer Society, 2008, pp. 131–136.
- [Knu97] Donald E. Knuth, *Stable marriage and its relation to other combinatorial problems*, American Mathematical Society, Providence, R.I, 1997.
- [Man13] David Manlove, *Algorithmics of matching under preferences*, World Scientific Publishing Company, March 2013.
- [MW70] D. G. McVitie and L. B. Wilson, *Stable marriage assignment for unequal sets*, BIT Numerical Mathematics **10** (1970), no. 3, 295–309.
- [MW71a] ———, *The stable marriage problem*, Communications of the ACM **14** (1971), no. 7, 486–490.
- [MW71b] ———, *Three procedures for the stable marriage problem*, Communications of the ACM **14** (1971), no. 7, 491–492.
- [RS90] Alvin E. Roth and Marilda A. Oliveira Sotomayor, *Two-sided matching: A study in game-theoretic modeling and analysis*, Econometric Society Monographs, Cambridge University Press, 1990.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

Il problema dei matrimoni stabili

Sommario

Dopo la breve digressione introduttiva sul problema proposto nel 1962 da David Gale e Lloyd Shapley [GS62], doverosamente incentrata nel più genuino ambito retroinformatico, dedichiamo un altro articolo alla parte più moderna della sua lunga storia. Questo problema e i suoi numerosi derivati sono stati affrontati nei decenni attingendo a piene mani a tutto l'arsenale di tecniche tipiche dell'ottimizzazione combinatoria: programmazione lineare e intera, *constraint programming*, algoritmi paralleli, metodi decentralizzati, perfino algoritmi genetici e ant-colony. Tuttavia, le più radicali ottimizzazioni sono arrivate solo tra fine anni Ottanta (grazie ad un cambio di paradigma che ha gettato una luce diversa sulla reale natura algebrica dello spazio delle soluzioni) e la prima metà degli anni Novanta, con una reinterpretazione «circuitale» del problema nell'ambito delle reti logiche: tutti algoritmi che sono poi giunti sostanzialmente invariati fino ai giorni nostri.

«*Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.*»
(Leopold Kronecker, 1823-1891)

1 Introduzione.

La maggior parte dei testi di algoritmica che fanno menzione del problema dei matrimoni stabili (SM) si fermano, senza apparente motivo, all'algoritmo man-optimal di Gale & Shapley [GS62], il quale tuttavia non è che il capostipite della famiglia di algoritmi risolutivi. Tra questi testi alcuni sono meritatamente famosissimi, come il Sedgewick [Sed88], il Wirth [Wir76] o il Lawler [Law11], ma ne esistono almeno altre due dozzine, tra cui ad esempio [PTW09, KG88, KT06], tanto che una elencazione esaustiva sarebbe del tutto inappropriata in questo contesto.

Pagato il doveroso tributo ai padri fondatori di questo settore di ricerca con un precedente articolo a pieno titolo intriso di tematiche di *retroprogramming* tra computer a transistor KDF-9, memorie a nuclei magnetici, ALGOL e COMAL, vogliamo ora andare oltre e dedicare il giusto spazio alla vera e propria pietra angolare nella storia del problema: l'algoritmo di generazione esaustiva delle soluzioni di Dan M. Gusfield (University of California, Davis) e Robert W. Irving (University of Glasgow, Scotland), i cui principi di funzionamento saranno esaminati nel modo più didattico possibile, al fine di raggiungere la più vasta platea di lettori, con particolare riguardo agli studenti più giovani.

Come a più riprese accennato, nei decenni il problema è stato affrontato da numerosi punti di vista, non solo da quello algoritmico: in particolare, economisti e teorici dei giochi¹ hanno approfondito ad esempio l'uso di metodi come inganno, manipolazione, corruzione, coalizioni segrete per coartare a vantaggio di singoli o piccoli gruppi l'output dell'algoritmo classico.

Dal punto di vista computazionale, è stato collaudato l'uso di tecniche asseverate come programmazione lineare e intera (PLI), *constraint programming*, algoritmi paralleli, metodi decentralizzati, perfino algoritmi genetici² e ant-colony per risolvere casi particolari del problema (es. SMI, Hospital/Residents) e sue immediate derivazioni: su tutte il problema detto Stable Roommates (SR). Tuttavia, ad oggi l'approccio maggiormente efficiente per la generazione esaustiva rimane quello del quale qui discutiamo.

Si esorta il lettore a fissare subito bene in mente i tre capisaldi fondamentali che hanno motivato la stesura del presente articolo:

1. Le istanze del problema Stable Marriage di cui parliamo sono dell'ordine di grandezza di $10^3 \div 10^5$ propensi (cliniche, college, posti pubblici in generale a livello di nazioni o grandi regioni): sono questi i numeri

¹Categoria alla quale appartengono anche i pionieri Gale e Shapley, nonché Alvin Roth [RS90]: per i loro contributi a questo problema, nel 2012 a Shapley e Roth è stato assegnato il Nobel per l'Economia.

²Si tratta di euristiche maturate nel campo della IA, ispirate ad un generico concetto di «evoluzione» darwiniana. L'idea che sta alla base degli algoritmi genetici è sostanzialmente quella di selezionare potenziali soluzioni ottimali e di ricombinarle più o meno sistematicamente fra loro, in maniera tale che esse evolvano «spontaneamente» verso un punto di ottimo. Questo genere di approccio, alla fine del secolo scorso, ha goduto anche di un effimero quarto d'ora di gloria nel mondo della programmazione mainstream, in una delle numerose ondate di entusiasmo e di neofilia che spesso attraversano il mondo della IT: in special modo quando collegate a tematiche di grande risonanza, come l'Intelligenza Artificiale. Oggi fa semplicemente parte dell'arsenale di euristiche classificate nei canonici manuali della IA (es. [RN09]) e utilizzate per problemi NP-hard, in mancanza di approcci migliori.

che hanno motivato le ricerche di Gusfield, Irving e loro collaboratori negli anni Ottanta. Per questo motivo le prestazioni dell'algoritmo in spazio e in tempo sono fondamentali, ed è essenziale comprendere almeno a grandi linee la struttura algebrica dello spazio delle soluzioni e le idee alla base dell'algoritmo;

2. Enumerare *tutte* le soluzioni stabili per istanze di tali dimensioni ha ampiamente senso, come in ogni altro caso in cui riusciamo ad ottenere una generazione combinatoria esaustiva con prestazioni accettabili. Nella pratica, molto lavoro è stato dedicato ad algoritmi maggiormente specifici in grado di individuare direttamente una soluzione intermedia con determinate caratteristiche (ad esempio le cosiddette soluzioni «minimum regret» nelle quali si cerca un compromesso tra le preferenze di ambedue i gruppi, approccio diverso rispetto all'ottimalità paretiana debole). Tuttavia, rimane importante dal punto di vista didattico comprendere il funzionamento dell'algoritmo esaustivo e gli strumenti teorici utilizzati da Gusfield e Irving, così come poter esaminare autonomamente dei set completi di soluzioni e i relativi grafi associati. Questi ultimi possono essere facilmente generati a partire dall'output sintetico dell'applicazione esemplificativa.
3. La monografia a cui facciamo riferimento è totalmente incentrata sugli aspetti algoritmici e sulla struttura algebrica dello spazio delle soluzioni: gli autori, come d'abitudine in simili testi teorici, non forniscono neppure un minimale esempio di codice in un qualsivoglia linguaggio, né nel testo né in un sito web di riferimento. Anche i riferimenti alle strutture dati da utilizzare concretamente in una implementazione rimangono solo cenni impliciti, affidati all'intuizione e all'esperienza del lettore. Il risultato di numerose ricerche compiute a più riprese negli ultimi anni conferma che non sembra esservi pubblicamente reperibile sul web alcuna implementazione di riferimento o di studio dell'algoritmo di Gusfield e Irving³, mentre ve ne sono in abbondanza per l'algoritmo classico di Gale-Shapley come per molti altri ben noti esempi presentati privi di codice in manuali e monografie famose (un esempio su tutti: i RBT di Sedgewick [Sed01]). Per contro, l'esperienza didattica conferma che un semplice esempio di codice aumenta notevolmente la comprensione della dinamica dell'intero algoritmo, dato anche il modo ovviamente molto graduale e diluito in cui viene spiegato e presentato nel testo [GI89].

Il presente articolo consta di due parti fondamentali.

Nella sezione 2 si affrontano in modo molto conciso e didattico i preliminari, le definizioni, i concetti fondamentali e lo pseudocodice così come presentati nella monografia di Gusfield e Irving, da cui è tratto praticamente tutto il materiale teorico.

La sezione 3 presenta invece una originale implementazione dell'Autore in linguaggio C89 (ANSI/INCITS X3.159-1989, ISO/IEC 9899:1990), che costituisce una essenziale ma completa *proof of concept* per l'algoritmo di generazione esaustiva ideato dai due studiosi negli anni Ottanta.

Abbiamo inoltre aggiunto nella sezione 5 una intervista cortesemente rilasciata all'Autore dal professor Robert W. Irving.

2 Preliminari e definizioni.

Per padroneggiare completamente le idee e i concetti alla base dell'algoritmo di Gusfield e Irving ([GI89, GIL87, Gus87, ILG87, IL86]) occorre presentare ordinatamente alcuni risultati così come esposti nella monografia, iniziando dai più intuitivi ed elementari e mantenendo il comodo riferimento matrimoniale degli autori originali. L'algoritmo in particolare è stato descritto in dettaglio dal professor Dan Gusfield in [Gus87], mentre l'uso delle rotazioni per una generazione efficiente appare già nell'articolo [IL86] del professor Robert Irving e del suo allievo Paul Leather.

L'algoritmo finale non è eccessivamente complesso, ma è comunque articolato e richiede la padronanza di una serie di concetti e definizioni non sempre intuitivi per gli studenti più giovani, anche se le idee computazionali su cui si basa sono da decenni patrimonio comune nel campo della generazione combinatoria e dell'ottimizzazione. L'intero articolo, come già accennato, è basato sullo pseudocodice, sui teoremi, sulle definizioni, sugli esempi e sulle idee esposte nella monografia [GI89] ed è impostato in modo da mantenere la spiegazione il più semplice possibile, rinunciando (sia pure malvolentieri) a tutte le dimostrazioni e cercando di non appesantire inutilmente le notazioni alla ricerca di un rigore estremo, che giustamente non può trovare spazio in un contesto divulgativo come il presente, anche se si è cercato di mantenere una minimale coerenza notazionale tra le varie sezioni del presente testo. Non è comunque compito né tantomeno intenzione dell'Autore riscrivere la monografia in oggetto condensandola in queste poche pagine, anche se per dovere di cronaca occorre rilevare che numerosi studenti

³In una recente comunicazione privata, il professor David Manlove (Università di Glasgow, peraltro autore di [Man13]) ha reso noto all'Autore che un suo ex studente di dottorato, Augustine Kwanashie, nell'ambito della sua tesi, ha effettivamente implementato anche l'algoritmo esaustivo della monografia, sebbene non ne abbia reso pubblici i sorgenti per vari ordini di motivi. Tale lavoro è poi confluito in un framework collettivo della medesima Università, un toolkit accademico che offre una interfaccia web alle implementazioni di circa 40 diversi algoritmi nell'area degli accoppiamenti con preferenze: MATWA (si veda la sezione «Hospitals Residents» per il contributo in questione).

dei giorni nostri non riescono a produrre autonomamente un programma completo partendo dal semplice studio del testo di Gusfield e Irving.

L'idea è quella di rendere il più comprensibili possibile le idee alla base dell'incremento prestazionale ottenuto dai due autori. Chiamiamo TSM l'insieme (finito) di tutte le soluzioni stabili, $|TSM|$ la cardinalità di tale insieme, ed $n \in \mathbb{N}$ (come di consueto) la dimensione del problema, con $n > 1$: i metodi enumerativi esaustivi elaborati nei primi 25 anni dopo la pubblicazione seminale di Gale e Shapley (su tutti gli algoritmi in ALGOL60 di McVitie e Wilson [MW71b, MW71a]) richiedevano non meno di $O(n^3 \cdot |TSM|)$, e si può mostrare che i migliori di essi avevano come limite inferiore $\Omega(n^3 \cdot |TSM| / \log^2 |TSM|)$: per contro l'algoritmo che qui analizziamo è in grado di generare *tutte le soluzioni stabili* dell'insieme TSM in $O(n^2 + n \cdot |TSM|)$.

Sarà quindi opportuno e utile tracciare subito una *roadmap* minimale dei concetti e dei risultati necessari per raggiungere un simile miglioramento, così come illustrati nella monografia alla quale facciamo costante riferimento.

- Innanzi tutto occorrono delle **strutture dati** astratte modificate rispetto al problema originale 2.1, che garantiscano in tempo costante sia la cancellazione (senza alterare l'ordine) che l'accesso casuale: le *GS-lists* 2.5.
- Occorre poi ricavare le **due soluzioni stabili estremali** man-optimal e woman-optimal 2.1.1.
- Di seguito è necessario mostrare la *struttura reticolare* 2.6 dell'insieme delle soluzioni, grazie ad una *relazione d'ordine* 2.6 appositamente definita. La struttura algebrica dello spazio delle soluzioni ci consente una rappresentazione compatta tramite differenze, la quale può essere trattata algoritmicamente in modo molto più efficiente definendo un apposito operatore: la *rotazione* 2.7.
- Infine, dopo avere mostrato come elencare esaustivamente le rotazioni 2.7 e generarne il grafo orientato ridotto $\bar{G}(TSM)$, si procede a **generare dinamicamente** il *reticolo delle soluzioni*.

L'algoritmo finale di generazione esaustiva della soluzioni al problema SM risulta quindi articolato in quattro fasi principali:

- **Fase 1:** generazione delle GS-lists e delle soluzioni stabili estremali;
- **Fase 2:** elencazione di tutte le rotazioni;
- **Fase 3:** generazione del grafo orientato compatto delle rotazioni $\bar{G}(TSM)$;
- **Fase 4:** generazione esaustiva delle soluzioni tramite una visita completa del grafo orientato $\bar{G}(TSM)$.

2.1 Il problema Stable Marriage (SM).

Iniziamo ricordando in termini intuitivi e informali il problema e i risultati presentati da David Gale (Brown University) e Loyd Shapley (RAND corporation) in [GS62]: per modellare un particolare problema di assegnazione con preferenze (l'ammissione degli studenti nei college), gli autori hanno fatto ricorso alla semplice astrazione dello stable marriage (SM). Abbiamo due gruppi di uomini (celibi) M e donne (nubili) W , di pari dimensione. A ciascun individuo viene chiesto di produrre una lista nella quale elenca, in ordine di preferenza decrescente, *tutti* gli individui dell'altro gruppo, dal più desiderabile al meno desiderabile. In questa sede trattiamo infatti unicamente il problema nella sua formulazione originale e non siamo interessati alle numerose varianti che prevedono, tra l'altro: gruppi di diverse dimensioni, un singolo gruppo (SR, Stable Roommates), elenchi di preferenze incompleti, presenza di partner inaccettabili, condizioni di indifferenza e quant'altro.

Ogni soluzione del problema è rappresentata da un elenco di coppie che forma una biezione tra i due insiemi dati, ponendo in corrispondenza biunivoca ciascun uomo con la rispettiva partner donna: tale soluzione, informalmente, si dice «stabile» se in essa non esistono due individui, uomo e donna, che si preferirebbero reciprocamente rispetto agli attuali partner, secondo quanto espresso nelle loro liste di preferenze.

Gale & Shapley hanno dimostrato che *qualsiasi* istanza del problema ammette almeno una soluzione stabile, e che tale soluzione espone la proprietà di assegnare a ciascun membro del gruppo proponente il **migliore partner possibile**, ed a ciascun membro dell'altro gruppo sistematicamente **il partner meno desiderabile**.

2.1.1 L'algoritmo originale di Gale e Shapley.

Per comprendere il concetto di «miglior partner possibile» è sufficiente considerare il funzionamento dell'algoritmo originale GS, scritto in modo da favorire il gruppo maschile (*man-optimal*):

```

procedure STABLE_MARRIAGE
    assign each person to be free;
    while some man  $m$  is free do
        begin
             $w :=$ the next woman on  $m$ 's list;
            if  $free(w)$  then
                assign  $m$  and  $w$  to be engaged {to each other};
            else
                if  $w$  prefers  $m$  to her fiancé  $m'$  then
                    assign  $m$  and  $w$  to be engaged;
                    free  $m'$ ;
                else
                     $w$  rejects  $m$  { $m$  remains free};
        end;
    output the  $n$  engaged pairs;

```

Come è immediato osservare, ogni volta che esiste un conflitto di preferenze maschili (un potenziale nuovo partner si propone ad una donna che ha già un partner) esso viene risolto valutando la lista di preferenze femminile. Intuitivamente, avviene così che l'uomo relativamente più desiderabile per la donna «contesa» ottiene il migliore accoppiamento, mentre l'altro è costretto a procedere nella sua lista di preferenze verso la prossima potenziale partner, per definizione meno desiderabile dell'attuale. Non occorre molto a convincersi che l'algoritmo opera in $O(n^2)$: nel caso peggiore, ciascuno degli n uomini dovrà proporsi a tutte le n donne della sua lista di preferenze, da cui la complessità computazionale indicata.

Definizione. La potenziale coppia con il partner meno desiderabile viene detta *instabile*, mentre l'altra coppia è detta *bloccante* in quanto impedisce la formazione della coppia col partner maschile meno desiderabile.

Al termine dell'esecuzione, ciascun membro dell'insieme che avanza le proposte (in questo caso, gli uomini) avrà così ottenuto la migliore partner, compatibilmente con i conflitti di preferenze inevitabilmente sorti durante le iterazioni. Gale e Shapley dimostrano facilmente che tale risultato è ottimale in assoluto e non migliorabile in nessun'altra eventuale soluzione stabile esistente. In ogni caso, anche se qualcuno rimarrà inevitabilmente scontento (perché non potrà avere la sua prima scelta...), è dimostrato il seguente

Teorema. *L'algoritmo GS termina sempre senza lasciare uomini o donne non accoppiati, e la soluzione stabile prodotta è sempre la migliore possibile per il gruppo proponente. D'altro canto, tale soluzione è anche la peggiore possibile dal punto di vista del gruppo che riceve le proposte.*

Tale proprietà, come già accennato anche in altre sedi, costituisce la cosiddetta ottimalità paretiana debole.

2.1.2 Numero massimo di soluzioni stabili.

Poiché il nostro scopo finale è quello di enumerare esaustivamente le soluzioni in modo efficiente, poniamo a margine i casi dotati solamente di una o due soluzioni e ci concentriamo senza ulteriore specificazione sui casi che ammettono numerose soluzioni stabili, che nel seguito indicheremo complessivamente come *TSM* e singolarmente come SM_i , con i opportuno indice. Questo ci conduce subito alla prima e più importante domanda in questo tipo di problemi: quante sono le soluzioni stabili per una data istanza? Gale & Shapley, come abbiamo appena ricordato, hanno dimostrato che esiste sempre almeno una soluzione stabile. Abbiamo anche già accennato al fatto che in taluni casi tale soluzione è unica. Ma purtroppo non esiste ancora una risposta semplice per il caso generale, ed è un problema ancora aperto determinare il numero massimo di soluzioni stabili per una generica istanza del problema. Non solo: non siamo neppure in grado di generare o selezionare deterministicamente la lista di preferenze che massimizza il numero di soluzioni stabili. D'altro canto, è stato dimostrato che il numero *medio* di matrimoni stabili $g(n)$ è asintotico ad $e^{-1}n \ln n$; tuttavia, è parimenti confermato (si veda in particolare [DBS13] e la relativa bibliografia) che il numero massimo di soluzioni cresce in modo esponenziale col crescere della dimensione del problema, rendendo del tutto implausibile qualsiasi approccio bruteforce di ricerca esaustiva. Esistono come sempre in questi casi delle tabelle empiriche (si veda ad esempio [Thu02]) per piccoli valori di n , ricavate per via computazionale.

2.2 Liste di preferenze.

Siano date le due liste M e W , rispettivamente per uomini e donne, ovviamente non vuote e tali che $|M| = |W| = n$, dove vale come già ricordato $n \in \mathbb{N}$, $n > 1$. I membri delle due liste $m_i \in M, w_i \in W$ sono

codificati per comodità con il segmento iniziale degli interi positivi $[1, n]$ ereditandone l'ordine naturale, salvo dove diversamente opportuno. Quindi, ad esempio, per una istanza di dimensione $n = 4$ avremo:

$$M = \{1, 2, 3, 4\}, \quad W = \{1, 2, 3, 4\}, \quad m_2 = 2, \quad w_1 = 1$$

e così via. Nel seguito faremo riferimento a uomini e donne usando le notazioni di volta in volta più adeguate:

- Con le lettere minuscole m e w , normalmente associate ad un pedice opportuno che assume valori nell'intervallo $[1, n]$;
- Usando direttamente l'intero $[1, n]$ corrispondente, laddove ciò non crei ambiguità e confusione: in particolare per semplificare la lettura delle liste di preferenze.

Definizione. Le *liste di preferenze* (per fissare le idee, maschili) associate a ciascun uomo $m_i \in M$ sono sequenze complete dei membri della lista per il sesso opposto, a ciascuno dei quali viene associato un ordinale naturale crescente $p_k = \{1, 2, \dots, n\}$. In altri termini, per ciascun uomo $m_i \in M$ la lista di preferenze non è che una *permutazione*⁴ dei membri dell'insieme W , e lo stesso vale per il sesso opposto.

Pertanto, supponendo di avere $n = 5$, $W = \{w_1, w_2, w_3, w_4, w_5\}$ e che l'uomo m_i abbia espresso la seguente lista di preferenze, in ordine: w_4, w_1, w_3, w_5, w_2 diremo ad esempio che m_i *preferisce* w_4 a w_3 perché la prima *precede* (ovvero è *a sinistra di*) w_3 . Questa terminologia risulta più intuitiva e relativamente meno ambigua di altre locuzioni equivalenti, pensando alla lista delle preferenze scritta su una riga come successione per ordinale (rank) *crescente*, e quindi per *preferenza decrescente*. In questo contesto, le posizioni $1 \dots n$ costituiscono infatti il *rank* delle varie donne nella lista di m_i , così che a rank *maggiore* corrisponde una *preferenza minore*.

Vale ovviamente la definizione speculare per le preferenze femminili. Nel seguito useremo liberamente le varie espressioni qui definite (precede/segue, destra/sinistra), per cercare di evitare pedantesche ripetizioni.

Ecco un semplice esempio numerico con $n = 4$, nello stile ubiquo in letteratura, dove la matrice a sinistra elenca le preferenze degli uomini, ordinate per righe:

$$\begin{array}{ccccccccc} 1 & : & 4 & 1 & 2 & 3 & 1 & : & 4 & 1 & 3 & 2 \\ 2 & : & 2 & 3 & 1 & 4 & 2 & : & 1 & 3 & 2 & 4 \\ 3 & : & 2 & 4 & 3 & 1 & 3 & : & 1 & 2 & 3 & 4 \\ 4 & : & 3 & 1 & 4 & 2 & 4 & : & 4 & 1 & 3 & 2 \end{array} \tag{1}$$

In questo esempio, il gentiluomo 1 *preferisce* la signorina 4 (in prima posizione) a tutte le altre, e così via in ordine decrescente di preferenza fino alla 3: la gentile signorina 4, a sua volta, *preferisce* il signor 4 a tutti gli altri, e in particolare a 1. Nella lista di preferenze dell'uomo 3: il rank di 4 è pari a 2 e il rank di 1 è pari a 4, quindi la signorina 1 si trova *a destra* di tutte le altre, in ultima posizione, mentre la signorina 4 è *a sinistra* di 3 e 1 e *a destra* di 2. E così via.

2.2.1 Numero di istanze per una dimensione fissata n .

Al paragrafo 2.1.2 abbiamo visto che per una data istanza il numero di soluzioni stabili può crescere esponenzialmente rispetto alla dimensione del problema, anche se non esiste una formula definita per il limite superiore. La domanda immediatamente successiva, ovvero «quante sono le possibili istanze per una dimensione n data?» ammette invece una risposta notevolmente più semplice, ma certo non molto incoraggiante. Sappiamo che qualsiasi istanza del problema ammette almeno una soluzione stabile, il che non pone alcun limite alle permutazioni che costituiscono le singole liste di preferenze, che possono essere ripetute e quindi nel caso limite possono anche essere tutte uguali. Pertanto, per ognuno dei $2n$ membri delle due liste sommate abbiamo *indipendentemente* $n!$ possibili liste di preferenze, ovvero permutazioni dei membri dell'altro insieme, per un totale di $\underbrace{n! \cdot n! \cdots n!}_{2n} = (n!)^{2n}$: si arriva al medesimo risultato anche considerando le due liste come disposizioni con

⁴Dato un insieme finito contenente n elementi distinti, si dicono *permutazioni semplici* le presentazioni ordinate che si possono formare in modo da soddisfare contemporaneamente i seguenti criteri:

1. Ogni presentazione deve contenere *tutti* gli n elementi, ciascuno considerato una e una sola volta;
2. Ogni presentazione deve differire dalle altre solo e unicamente per *l'ordine* degli elementi.

In altri termini, una permutazione è una *regola* che a ogni elemento dell'insieme dato associa un naturale (un ordinale finito) che ne descrive la posizione, in modo univoco - ovvero, in ultima analisi, essa costituisce un **criterio di ordinamento**. La formula che esprime il numero di permutazioni in funzione dell'ampiezza, data dall'intero non negativo n , è ben nota e fa riferimento al fattoriale:

$$P(n) = n! \stackrel{\text{def}}{=} \begin{cases} 1 & \text{per } n \in \{0, 1\} \\ \prod_{j=1}^n j & \text{per } n \in \mathbb{N} \setminus \{0, 1\} \end{cases}$$

ripetizioni⁵, i cui $2n$ elementi sono in realtà a loro volta permutazioni. In questo modo, per un semplice problema di ampiezza 4 abbiamo $(4!)^8 = 110.075.314.176$ di possibili diverse configurazioni per le liste di preferenze⁶. Numeri di quest'ordine di grandezza non fanno che confermare quanto già asserito riguardo a ogni possibile velleità di esplorazione esaustiva delle istanze, anche per dimensioni n molto ridotte.

2.3 Soluzioni stabili.

Definizione. Una soluzione stabile $SM_p \in TSM$ è una biezione tra M e W nella quale non esistono $m_i \in M$ e $w_j \in W$ tali che m_i preferisce w_j alla sua attuale partner e contemporaneamente anche w_j preferisce m_i al suo attuale partner. Come suggerisce il nome, in una soluzione stabile non vi sono appunto coppie instabili 2.1.1.

Tale soluzione può essere espressa come sottoinsieme del prodotto cartesiano $M \times W$ sotto forma di elenco esplicito di coppie (m_i, w_j) dove ordinatamente $m_i \in M$, $w_j \in W$, oppure in forma sintetica come permutazione $w_a w_b \dots w_k$ che implicitamente specifica la biezione associando ciascun elemento di W all'uomo in posizione $1, 2, 3, \dots, n$ della lista M , seguendo quindi la convenzione posizionale della notazione 2-line per le permutazioni. Un semplice esempio chiarirà definitivamente l'uso della notazione. La soluzione stabile seguente, nella quale ogni coppia indica prima l'uomo e poi la donna, è una delle due soluzioni stabili ammesse dall'istanza 1:

$$SM = \{(1, 4), (2, 1), (3, 2), (4, 3)\}$$

Essa rappresenta gli accoppiamenti illustrati in figura 1, e può essere espressa in notazione a due linee e in singola linea, rispettivamente, come segue:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix} \rightarrow 4 \ 1 \ 2 \ 3$$

Tutte le notazioni esemplificate indicano esattamente la medesima configurazione di coppie. Definiamo alcune ulteriori pseudofunzioni di comodo, volte ad abbreviare le spiegazioni ed evitare inutili perifrasi, tenendo ben presente che dal punto di vista implementativo esse saranno sistematicamente realizzate con semplici accessi a strutture indicizzate e LUT in $O(1)$.

$$wP(w_i, SM_p) = m_j$$

$$mP(m_i, SM_p) = w_j$$

$$wRank(w_j, m_i) = k$$

$$mRank(m_i, w_j) = k$$

Dove ovviamente gli indici i, j, k, p sono implicitamente tutti opportuni. Le prime due pseudofunzioni restituiscano rispettivamente il partner di w_i, m_i nella soluzione SM_p considerata. La funzione $wRank(w_j, m_i)$ ci restituisce la posizione dell'uomo m_i nella lista di preferenze della donna w_j , mentre $mRank(m_i, w_j)$ svolge il lavoro complementare restituendo la posizione di w_j nella lista di preferenze di m_i .

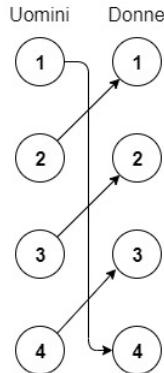


Figura 1: Esempio di soluzione SM per un'istanza di dimensione 4.

2.4 Un utile esempio di dimensione $n = 8$.

Restando ben consapevoli che l'algoritmo qui trattato è specificamente pensato per istanze del problema di grandi o grandissime dimensioni, dell'ordine di $10^3 \div 10^5$ proponenti, ci sarà utile fare costante riferimento

⁵Dati n simboli distinti e un intero $0 < k \leq n$, si dicono *disposizioni con ripetizioni* di questi n elementi in classe k tutte le possibili presentazioni tali che:

- Ciascuna presentazione contenga esattamente k simboli, non necessariamente distinti;
- Ciascun simbolo possa essere ripetuto fino a k volte;
- Due presentazioni differiscano per qualche simbolo, oppure per l'ordine in cui i simboli sono disposti.

Il totale delle possibili disposizioni con ripetizioni di n elementi in classe k è pari a $DR(n, k) = n^k$. Ad esempio, i $DR(2, 4) = 16$ numeri binari a 4 bit 0000, 0001, ..., 1111 sono uno dei più classici esempi di disposizione con ripetizione, e le disposizioni con ripetizione $DR(3, 2) = 9$ degli elementi dell'insieme $A = \{a, b, c\}$ sono tutte le coppie $aa, ab, ac, ba, bb, bc, ca, cb, cc$.

⁶Qui potremmo dilungarci per diverse pagine, seguendo in particolare il già citato [Thu02], per esaminare le peculiari proprietà delle soluzioni nelle istanze le cui liste di preferenze siano quadrati latini, un oggetto combinatorio di grande fascino e lungamente studiato. Sfortunatamente l'articolo è già molto lungo e gli argomenti strettamente inerenti l'algoritmo esaminato, sia pure ridotti all'osso, sono già fin troppo numerosi anche senza aggiungere ulteriori divagazioni.

alla istanza utilizzata come esempio di base nella monografia [GI89], caratterizzata da $|M| = |W| = 8$, e dalle seguenti liste di preferenze 8×8 , ordinate per righe, rispettivamente espresse dagli uomini $m_i \in M$ (a sinistra) e dalle donne $w_i \in W$:

1:	5	7	1	2	6	8	4	3
2:	2	3	7	5	4	1	8	6
3:	8	5	1	4	6	2	3	7
4:	3	2	7	4	1	6	8	5
5:	7	2	5	1	3	6	8	4
6:	1	6	7	5	8	4	2	3
7:	2	5	7	6	3	4	8	1
8:	3	8	4	5	7	2	6	1
1:	5	3	7	6	1	2	8	4
2:	8	6	3	5	7	2	1	4
3:	1	5	6	2	4	8	7	3
4:	8	7	3	2	4	1	5	6
5:	6	4	7	3	8	1	2	5
6:	2	8	5	3	4	6	7	1
7:	7	5	2	1	8	6	4	3
8:	7	4	1	5	2	3	6	8

(2)

Applicando l'algoritmo GS all'istanza di esempio 2 si ottiene la soluzione *man-optimal* seguente, usando sempre la convenzione di esprimere le coppie come (*uomo, donna*):

$$SM_0 = \{(1, 5), (2, 3), (3, 8), (4, 6), (5, 7), (6, 1), (7, 2), (8, 4)\}$$

Invertendo le liste di preferenze ed eseguendo GS, si ottiene la soluzione woman-optimal che segue:

$$SM_z = \{(1, 3), (2, 6), (3, 2), (4, 8), (5, 1), (6, 5), (7, 7), (8, 4)\}$$

Sarà utile evidenziare queste soluzioni estremali nelle liste di preferenze date, per rendere massimamente intuitive anche dal punto di vista visuale le proprietà fondamentali dell'insieme TSM delle soluzioni stabili.

1:	5	7	1	2	6	8	4	3
2:	2	3	7	5	4	1	8	6
3:	8	5	1	4	6	2	3	7
4:	3	2	7	4	1	6	8	5
5:	7	2	5	1	3	6	8	4
6:	1	6	7	5	8	4	2	3
7:	2	5	7	6	3	4	8	1
8:	3	8	4	5	7	2	6	1
1:	5	3	7	6	1	2	8	4
2:	8	6	3	5	7	2	1	4
3:	1	5	6	2	4	8	7	3
4:	8	7	3	2	4	1	5	6
5:	6	4	7	3	8	1	2	5
6:	2	8	5	3	4	6	7	1
7:	7	5	2	1	8	6	4	3
8:	7	4	1	5	2	3	6	8

Sono marcati in rosso gli accoppiamenti man-optimal e in ciano quelli woman-optimal. Questo semplice esempio evidenzia già alcune proprietà molto importanti, e in particolare gli intuitivi spostamenti lungo le liste di preferenze nel passare da una soluzione estrema all'altra. Si nota immediatamente che, a parte la coppia $(8, 4)$ che in questo caso costituisce un'invariante (definita come *coppia fissa*), tutte le altre partner della soluzione woman-optimal risultano meno desiderabili (ossia hanno un rank *maggior*) rispetto a quelle della soluzione man-optimal. Identica considerazione, a parti invertite, si applica all'insieme delle donne W .

Nel seguito stabiliremo le semplici regole di una vera e propria algebra delle soluzioni, che ci consentirà di passare dall'una all'altra soluzione stabile in modo deterministico. Andremo anche a caratterizzare in senso molto più preciso una relazione d'ordine tra le varie soluzioni, fermo restando fin da ora che man-optimal e woman-optimal sono rispettivamente il *minimo* e il *massimo* nell'insieme TSM delle soluzioni stabili, come abbiamo implicitamente anticipato parlando di soluzioni *estremali* ed usando la notazione SM_0, SM_z per denotare le soluzioni presentate. Per il momento, il lettore tenga presente che l'istanza proposta ammette un totale di otto soluzioni stabili.

2.5 Liste ridotte o *GS-lists*.

Per ottimizzare le prestazioni dell'algoritmo esaustivo di cui discutiamo, il primissimo passo consiste nell'eliminazione dei dati superflui dalle liste di preferenze. Per fare questo introduciamo un semplice operatore di cancellazione.

Definizione. L'operatore $\text{delete}(m_i, w_j)$ cancella l'uomo m_i dalla lista di preferenze della donna w_j , e viceversa.

Ad esempio, date le liste 2 e presi $m = 1, w = 7$, dopo una singola $\text{delete}(1, 7)$ le relative liste risultano così modificate: 1: 5 ~~X~~ 1 2 6 8 4 3, 7: 7 5 2 ~~X~~ 8 6 4 3.

Pensando all'algoritmo GS, consideriamo una iterazione durante la quale l'uomo m_i si propone alla donna w_j (che supponiamo libera, o impegnata con un partner m_k tale che $wRank(w_j, m_k) > wRank(w_j, m_i)$): indipendentemente dal fatto che la coppia sia o meno stabile, w_j non avrà in alcun caso partner il cui rank nella sua lista di preferenze risulti «peggiore» di quello di m_i , ovvero alla sua destra. Questo vale sia per la soluzione stabile corrente, sia per tutte le altre soluzioni stabili esistenti. Il motivo è estremamente semplice e intuitivo: osservando le due soluzioni estreme man-optimal e woman-optimal è immediato inferire che quando esistono

più soluzioni, esse portano successivamente gli uomini ad accoppiarsi con partner sempre meno desiderabili (spostando quindi il rank verso valori più elevati nella relativa lista di preferenze) e viceversa per le donne, procedendo dalla soluzione man-optimal verso quella woman-optimal. In qualsiasi caso, tutti i partner «a destra» di m_i possono essere cancellati dalla lista di w_j , e allo stesso modo w_j verrà eliminata dalle liste preferenziali di tutti i partner appena eliminati.

Tornando all'esempio appena fatto, se ipoteticamente $w = 7$ ricevesse la proposta di $m = 5$ sarebbe possibile eliminare dalla sua lista i candidati 2, 8, 6, 4, 3 in quanto non potranno mai comparire in alcuna successiva soluzione stabile, e viceversa sarebbe necessario anche eliminare $w = 7$ dalle liste di tutti i candidati sopra elencati, per il medesimo motivo.

2.5.1 Fase 1: l'algoritmo Gale-Shapley esteso.

A questo punto è possibile introdurre una versione modificata dell'algoritmo GS, denominata EGS, in grado di produrre una lista ridotta che chiameremo MGS-list (se sono gli uomini a proporsi) o WGS-list nel caso opposto. Eseguendo due volte l'algoritmo (inizialmente con gli uomini che si propongono, poi all'opposto) ed usando la MGS-list direttamente come input per il secondo passaggio, si otterrà la lista ridotta detta GS-list che gode di proprietà altamente desiderabili dal punto di vista del nostro algoritmo. Di seguito lo pseudocodice, così come riportato nella monografia [GI89].

```

procedure EXTENDED_GS
    assign each person to be free;
    while some man  $m$  is free do
        begin
             $w := mFirst(m)$ ;
            if  $w$  is engaged with some man  $m'$  then
                free  $m'$ ;
            assign  $m$  and  $w$  to be engaged;
            for each  $m' := successor(m, w)$ 
                delete( $m', w$ );
        end;
    
```

Risulta immediato notare che l'algoritmo mantiene la medesima complessità computazionale $O(n^2)$ dell'originale. La pseudofunzione $mFirst(m)$ restituisce la donna col rank minore attualmente sulla lista di preferenze di m , mentre $successor(m, w)$ restituisce iterativamente tutte le eventuali preferenze di w con rank maggiore di $wRank(w, m)$. Si nota subito come in questo algoritmo le proposte vengano immediatamente accettate: se infatti w preferisse m' ad m , quest'ultimo sarebbe già stato cancellato dalla lista di w , e lei dalla sua, pertanto $mFirst(m)$ non restituirebbe in alcun caso w .

Risulta immediato notare come il partner di w sarà così sempre collocato in una *posizione notevole*: la più a destra della sua lista di preferenze. Gusfield e Irving dimostrano infatti il seguente

Teorema. *Per ogni data istanza del problema SM, valgono le seguenti proprietà:*

- 1) *Tutti gli accoppiamenti stabili possibili sono presenti nella GS-list;*
- 2) *Nessun accoppiamento contenuto nella GS-list può essere bloccato da una coppia non presente in tale lista;*
- 3) *Nella soluzione man-optimal (risp. woman-optimal), ogni uomo è accoppiato con la prima (risp. ultima) donna sulla sua lista ridotta GS-list e ciascuna donna con l'ultimo (risp. il primo) uomo sulla sua GS-list.*

Queste proprietà sono cruciali per consentire un accesso in $O(1)$ a tali locazioni critiche: il che è il primo passo elementare che consente di ottimizzare le prestazioni dell'algoritmo. Nel seguito faremo riferimento a tali locazioni con le intuitive pseudofunzioni qui elencate:

$$\begin{aligned} w_j &= mFIRST(m_i) = mP(m_i, SM_p) \\ w_k &= SECOND(m_i) \\ m_j &= wLAST(w_i) = wP(w_i, SM_p) \end{aligned}$$

La funzione $w_k = SECOND(m_i)$ fornisce la donna immediatamente a destra di $mFIRST(m_i)$, se esiste. Si noti che le prime due pseudofunzioni prendono in input un uomo e restituiscono una donna, mentre la terza opera in modo speculare.

Lemma. *Se una donna w_i , dopo l'elaborazione di EGS, si trova ancora nella lista ridotta di un uomo m_j con un rank superiore o uguale a 2, ciò significa che non è stata cancellata e quindi specularmente anche m_j è*

ancora presente nella lista preferenziale di w_i , necessariamente con un rank inferiore rispetto al partner man-optimal di lei, che ha il rank più elevato per definizione: $m_k = wP(w_i, SM_0) = wLAST(w_i)$. Quindi, in ultima analisi, dato un $m_j \in M$, per ogni eventuale w_i con $mRank(m_j, w_i) \geq 2$ si ha che $wRank(w_i, m_j) < wRank(w_i, wLAST(w_i))$. Ciò vale, in particolare, per $w_i = SECOND(m_j)$ laddove esista.

Il lemma appena esposto ha importanti conseguenze nell'applicazione delle rotazioni, come vedremo a breve 2.7. Il risultato dell'esecuzione di EGS sulle liste 2 genera le MGS-lists, come già detto. Una nuova esecuzione fornendo in input tali liste, ma a tabelle invertite, ci fornisce infine le GS-lists desiderate, sempre seguendo la convenzione di anteporre le liste maschili:

1: 5 8 3	1: 5 3 6
2: 3 8 6	2: 3 5 7
3: 8 5 1 6 2	3: 1 2
4: 6 8	4: 8
5: 7 2 1	5: 6 7 3 1
6: 1 5	6: 2 3 4
7: 2 5 7	7: 7 5
8: 4	8: 4 1 2 3

Si noti anche come questo pur semplicissimo algoritmo, principalmente a causa delle cancellazioni, richiede una struttura dati *ordinata* che consenta sia l'accesso indicizzato in $O(1)$ a locazioni arbitrarie (sono ovviamente escluse a priori farraginose e costose ricerche lineari), sia la cancellazione in $O(1)$ (senza modificare l'ordine⁷) per poter garantire una performance in $O(n^2 + n \cdot |TSM|)$: il che esclude a priori alberi ed heap, ma anche il ricorso alle più elementari strutture come array e liste. Occorrerà quindi pensare ad una struttura dati ibrida, di tipo *augmented*, che al solo costo di qualche scrittura extra consenta di ottenere le caratteristiche richieste.

2.6 Il reticolo delle soluzioni.

In modo del tutto analogo alla definizione di rank e preferenza entro le liste di una persona data, è possibile definire la preferenza di una soluzione stabile rispetto all'altra da parte di ciascun uomo o donna presenti nelle liste M, W .

Definizione. Date due distinte soluzioni stabili $SM_i, SM_j \in TSM$, una generica persona p preferisce la soluzione SM_i alla SM_j se e solo se preferisce il/la sua partner in SM_i a quello/a in SM_j . Si presti attenzione al fatto che abbiamo tre casi possibili:

1. La persona p preferisce SM_i a SM_j ;
2. La persona p preferisce SM_j a SM_i ;
3. La persona p è *indifferente* tra i due, poiché il/la partner è identico/a nelle due soluzioni.

Confrontare due soluzioni stabili rispetto a questa relazione diventa quindi possibile, come da definizione seguente.

Definizione. Date due distinte soluzioni stabili $SM_i, SM_j \in TSM$ diremo che SM_i *domina* SM_j , ovvero la *precede* (e lo indichiamo con $SM_i \prec SM_j$) dal punto di vista maschile se e solo se ciascun uomo della lista M preferisce SM_i a SM_j , oppure è indifferente.

Consideriamo ad esempio le seguenti soluzioni stabili:

$$SM_3 = \{(1, 8), (2, 3), (3, 1), (4, 6), (5, 7), (6, 5), (7, 2), (8, 4)\}$$

$$SM_5 = \{(1, 8), (2, 3), (3, 1), (4, 6), (5, 2), (6, 5), (7, 7), (8, 4)\}$$

Si tratta, per inciso, di soluzioni effettivamente ammesse dall'istanza 2. Si nota immediatamente che ogni uomo ha mantenuto la propria partner (il che equivale ad una condizione di indifferenza), tranne 5 e 7. Riportiamo qui le rispettive GS-lists:

5: 7 2 1
7: 2 5 7

⁷Nel vastissimo novero di casi che consentono di ignorare l'ordine degli elementi (ad esempio negli algoritmi di shuffle pseudo-casuali, nella rappresentazione di insiemi, negli algoritmi generatori di oggetti combinatori e in numerose altre situazioni) è sempre ottima prassi di progettazione l'uso di un vettore adeguatamente dimensionato, che consente senza alcun aggravio anche le operazioni tipicamente critiche per tale struttura dati, considerata come successione di valori rigidamente ordinata posizionalmente. Le «cancellazioni» vengono in tale caso realizzate sovrascrivendo l'elemento da eliminare con il contenuto dell'ultima posizione del vettore, il cui limite superiore di indicizzazione viene poi diminuito di una unità; gli «inserimenti», per contro, consistono unicamente di aggiunte in coda, con conseguente aumento del limite superiore di indicizzazione.

Ambedue gli uomini considerati hanno una partner in SM_5 (marcata in rosso) con un rank *superiore*, dunque meno preferibile, rispetto a quella in SM_3 (marcata in ciano): abbiamo quindi due uomini che *preferiscono* SM_3 ad SM_5 e sei condizioni di indifferenza, pertanto SM_3 *domina* SM_5 dal punto di vista maschile e scriviamo $SM_3 \prec SM_5$. D'altro canto, se consideriamo anche la soluzione seguente:

$$SM_2 = \{(1, 3), (2, 6), (3, 5), (4, 8), (5, 7), (6, 1), (7, 2), (8, 4)\}$$

non è difficile rilevare che risulta *impossibile* stabilire una relazione d'ordine tra SM_2 ed SM_3 , poiché abbiamo tutti e tre i casi contemporaneamente: sia condizioni di indifferenza, sia uomini che preferiscono SM_2 ad SM_3 , sia uomini che all'opposto preferiscono SM_3 ad SM_2 . L'esistenza di elementi non comparabili caratterizza gli *ordini parziali*.

Definizione. Un *ordine parziale* su un insieme \mathcal{A} è una relazione \preceq riflessiva, antisimmetrica e transitiva.

- Riflessiva: se $x \in \mathcal{A}$, allora $x \preceq x$;
- Antisimmetrica: se $x, y \in \mathcal{A}$, $x \preceq y$ e $y \preceq x$, allora $x = y$;
- Transitiva: se $x, y, z \in \mathcal{A}$, $x \preceq y$ e $y \preceq z$, allora $x \preceq z$;

Risulta infatti banale per gli autori mostrare che l'intero insieme delle soluzioni TSM è un ordine parziale finito (TSM, \preceq) sotto la relazione d'ordine appena definita, e che risulta un ordine *limitato* perché il suo unico elemento minimo è SM_0 (la soluzione man-optimal), dal momento che vale $SM_0 \prec SM_p$ per ogni $p \neq 0$, mentre per il medesimo motivo il massimo è SM_z (woman-optimal). Il passo successivo nella monografia, altrettanto immediato, è quello di mostrare che tale ordine parziale è un reticolo distributivo, secondo la seguente

Definizione. Un *reticolo distributivo* è un ordine parziale finito, caratterizzato dalle seguenti proprietà:

1. Ogni coppia di elementi a, b ammette un *massimo comune minorante* (indicato con $a \wedge b$ e detto *meet*) tale che $a \wedge b \preceq a$, $a \wedge b \preceq b$ e non esiste un elemento c tale che $c \preceq a$, $c \preceq b$ e $a \wedge b \prec c$;
2. Ogni coppia di elementi a, b del reticolo ammette un *minimo comune maggiorante* (indicato con $a \vee b$ e detto *join*) tale che $a \preceq a \vee b$, $b \preceq a \vee b$ e non esiste un elemento c tale che $a \preceq c$, $b \preceq c$ e $c \prec a \vee b$;
3. Vale la proprietà *distributiva*: $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ e $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

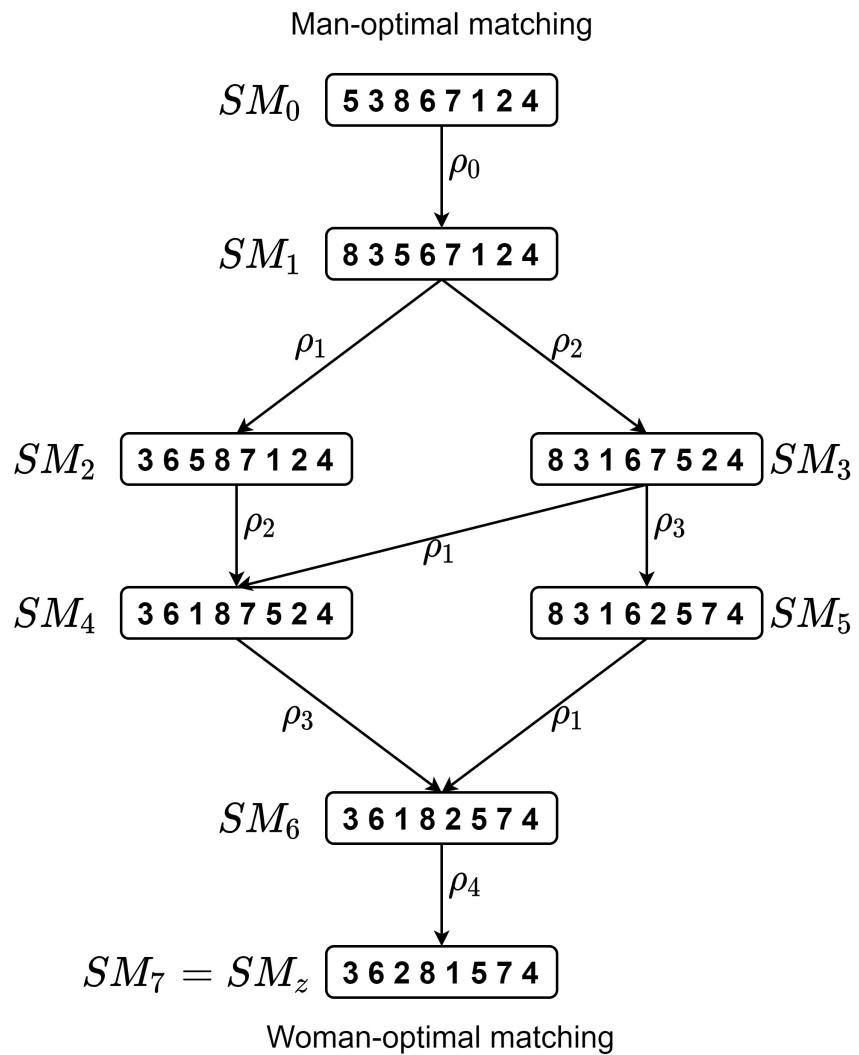


Figura 2: Il diagramma di Hasse delle soluzioni per l'istanza 2. Si prega il lettore di notare che la rappresentazione utilizzata nella monografia qui discussa risulta verticalmente invertita rispetto a quella impiegata nella vasta maggioranza della letteratura, probabilmente anche allo scopo di sottolineare visivamente l'effettivo cammino seguito dagli algoritmi di visita. Per coerenza si è qui voluta rispettare la medesima convenzione.

2.6.1 Anelli d'insiemi.

L'insieme delle soluzioni TSM , in quanto reticolo distributivo finito, è isomorfo ad una peculiare struttura algebrica denominata *ring of sets*, la cui rappresentazione ci consentirà implicitamente di ottimizzare le prestazioni dell'algoritmo di generazione esaustiva.

Definizione. Una famiglia non vuota di insiemi \mathfrak{R} è detta *ring of sets* (o *anello d'insiemi*) se e solo se è chiusa sotto le operazioni di *unione* e *intersezione* [Bir37, Bir93].

In altri termini, dati due insiemi \mathcal{A}, \mathcal{B} , si ha:

$$\mathcal{A}, \mathcal{B} \in \mathfrak{R} \text{ implica } \mathcal{A} \cup \mathcal{B} \in \mathfrak{R} \text{ e } \mathcal{A} \cap \mathcal{B} \in \mathfrak{R} \quad (3)$$

In estrema sintesi, la fondamentale importanza dei *ring of sets* in ambito discretistico, quindi nel caso di default degli anelli *finiti*, è sancita dal teorema di rappresentazione di Birkhoff (si veda anche [Sto36]), il quale ne stabilisce l'isomorfismo con un reticolo distributivo parimenti finito [Grä11] e quindi con un ben definito ordine parziale come quello che abbiamo appena rapidamente identificato al paragrafo precedente 2.6.

L'aspetto rilevante degli anelli d'insiemi è la possibilità di rappresentarli per differenze in maniera molto compatta: il nostro scopo finale è infatti quello di rappresentare indirettamente l'insieme delle soluzioni TSM come grafo orientato e attraversarlo efficientemente, enumerando così tutte le soluzioni esistenti per una data istanza. Si veda la figura 3 per uno dei più semplici esempi di ring of sets (in questo caso specifico, sia nel senso discretistico che in quello della teoria della misura!), l'insieme delle parti o insieme potenza, che contiene tutti i sottinsiemi propri di un generico insieme dato (ovviamente non vuoto), oltre all'insieme stesso. Sugli archi sono evidenziate le *differenze*

tra gli insiemi situati agli estremi dell'arco stesso, aspetto che viene affidato all'intuizione del lettore per non appesantire ulteriormente la trattazione.

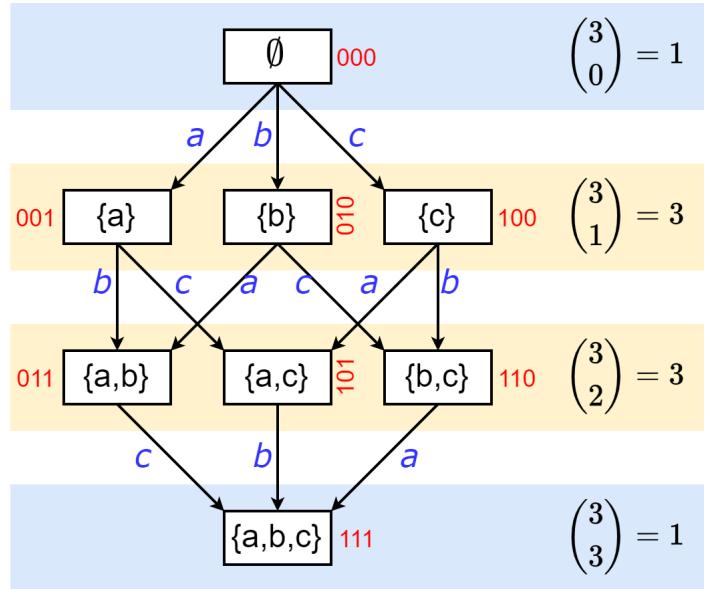


Figura 3: Esempio di insieme potenza $\mathcal{A} = \wp(3)$ come anello d'insiemi. L'immagine fornisce numerose informazioni combinatorie, tra cui: conteggio dei sottoinsiemi per dimensione, evidenziazione delle anticateze massimali, indicazione delle differenze (come elementi singoli) sulle frecce, codifica binaria nel vettore $\mathcal{V}(i)$ con uso della funzione caratteristica per ogni sottoinsieme $\mathcal{B}_k \subseteq \mathcal{A}$. Tale codifica a sua volta può essere utilizzata direttamente nella costruzione per differenze, ad esempio $\{a\} \cup \{c\} = \{a, c\} \equiv 001 + 100 = 101$. Oltre all'algebra degli anelli di insiemi, anche la combinatoria degli insiemi finiti è un campo estremamente interessante: si veda, ad esempio, il classico [And02] e in particolare [Spe28, Lub66] sul teorema di Sperner.

Osservazione. L'idea delle differenze, strettamente analoga al concetto di *distanza di Levenshtein* (o distanza di edit, [Lev66]), è estremamente consolidata e potente nell'ambito della combinatorica generativa. Essa è alla base dei numerosi *metodi di successione*, che sono in grado di generare il *successivo* oggetto combinatorio (secondo un ordinamento prestabilito) applicando una trasformazione a quello correntemente dato (di norma operano tale trasformazione in tempo $O(n \cdot \log_2 n)$ o superiore, con ben poche eccezioni). Allo stesso modo, gli autori Gusfield e Irving hanno dato corpo all'intuizione matematica che una soluzione stabile in TSM può essere ricavata da un'altra che la *precede*, nel senso rigorosamente definito entro il nostro reticolo, definendo e applicando una semplice trasformazione algebrica.

Il lettore tenga ben presente che questa, in estrema sintesi, è l'idea fondante alla base dell'algoritmo di generazione esaustiva e delle sue sorprendenti prestazioni. Dal punto di vista algoritmico, la rappresentazione tramite differenze può essere equivalentemente, ma più efficientemente, resa tramite un apposito operatore che andremo a definire nella prossima sottosezione.

Definiamo invece brevemente, per i nostri scopi, il P-set di una soluzione SM_p , e lo indichiamo con $P(SM_p)$, come segue:

Definizione. $P(SM_p)$ è l'elenco delle coppie (m, w) nelle quali per ogni uomo m_i compaiono nell'elenco sia la sua partner $w_j = mP(m_i, SM_p)$ sia le eventuali donne w_k sulla lista di preferenze (*completa*⁸) di m_i tali che $mRank(m_i, w_k) < mRank(m_i, w_j)$.

Tale P-set non è che una descrizione alternativa della soluzione SM_p , in ultima analisi. Quindi, ad esempio:

$$P(SM_0) = \{(1, 5), (2, 2), (2, 3), (3, 8), (4, 3), (4, 2), (4, 7), (4, 4), (4, 1), (4, 6), (5, 7), (6, 1), (7, 2), (8, 3), (8, 8), (8, 4)\}$$

come si può facilmente verificare dalle liste di preferenze complete dell'esempio 2, ed è immediato il seguente

Lemma. Date due soluzioni stabili distinte SM_i , SM_j si ha: $P(SM_i \vee SM_j) = P(SM_i) \cup P(SM_j)$ e $P(SM_i \wedge SM_j) = P(SM_i) \cap P(SM_j)$. Inoltre, SM_i domina SM_j se e solo se $P(SM_i) \subseteq P(SM_j)$.

Abbiamo quindi le caratteristiche sancite dalla definizione 3: per una istanza di dimensione n , l'insieme di base è il prodotto cartesiano $M \times W$ che contiene tutte le n^2 coppie (m, w) e la famiglia dei P-sets $P(TSM)$ è chiusa sotto l'unione e l'intersezione in quanto il reticolo distributivo finito delle soluzioni è chiuso sotto le operazioni di *meet* e *join* come da definizione 2.6.

Anelli di insiemi: un breve excursus. L'esperienza didattica degli scorsi decenni ci suggerisce che numerosi studenti, in particolare italiani e francesi, trovano difficoltà nel comprendere la nozione di *ring of sets* nei corsi avanzati di algoritmica, combinatorica, teoria dei reticolati e in altri ambiti afferenti alla matematica discreta. Pare causare problemi e confusione la pregressa (sovra)esposizione scolastica a concetti probabilistici e di teoria della misura, nei quali ambiti si fa uso di una peculiare definizione *settoriale* di anello d'insiemi, finalizzata alle necessità e agli scopi di tali discipline e diversa da quella usata in matematica discreta. Il mero fatto che per tale concetto esistono due distinte definizioni deve semplicemente suggerire, anche a chi non è esplicitamente specializzato in teoria dei modelli, che ciascuna di esse ha parimenti senso ed è strettamente funzionale ad un preciso scopo nel contesto al quale si riferisce.

Nelle **teorie dell'ordine** e in **matematica discreta**, e quindi anche nel presente contesto, la definizione universalmente in uso è la 3, senza necessità di ulteriore specificazione.

In teoria della misura, l'anello di insiemi è invece definito in maniera più restrittiva (che non è assolutamente sinonimo di «più rigorosa»!). In questo caso, una famiglia non vuota di insiemi \mathfrak{R} forma un anello di insiemi se è chiusa sotto l'intersezione e la differenza simmetrica: $A, B \in \mathfrak{R}$ implica $A \cap B \in \mathfrak{R}$ e $A \Delta B \in \mathfrak{R}$. Dal momento che $A \cup B = (A \Delta B) \Delta (A \cap B)$ si ha anche che $A, B \in \mathfrak{R}$ implica $A \cup B \in \mathfrak{R}$; allo stesso modo, da $A \setminus B = A \Delta (A \cap B)$ deriva che $A, B \in \mathfrak{R}$ implica $A \setminus B \in \mathfrak{R}$. Quest'ultima implicazione ha come conseguenza che un anello di insiemi, nel contesto della teoria della misura, include sempre l'insieme vuoto $A \setminus A = \emptyset$ e un anello $\mathfrak{R} = \{\emptyset\}$ che consista unicamente di quest'ultimo è il più piccolo anello d'insiemi possibile. Di fatto, le ultime due implicazioni ci dicono che nell'ambito della teoria della misura una famiglia non vuota di insiemi \mathfrak{R} forma un anello di insiemi se e solo se è chiusa sotto l'intersezione, l'unione, la differenza e la differenza simmetrica per ogni coppia di insiemi $A, B \in \mathfrak{R}$.

In ultima analisi, qualunque anello di insiemi nel senso usato in teoria della misura ha la struttura algebrica di un anello booleano ed è *anche* un anello di insiemi in senso discretistico, ma non vale necessariamente il contrario. Per le finalità e necessità proprie della teoria della misura e nel correlato campo probabilistico, nel cui merito sarebbe del tutto fuori luogo addentrarsi in questa sede, risulta evidentemente necessario *aggiungere struttura* rispetto alla definizione discretistica nel finito e alle strutture ad essa isomorfe secondo il classico teorema di rappresentazione di Birkhoff.

In realtà non è poi difficile mostrare che, dal punto di vista algebrico-topologico, si può trattare in modo uniforme la duplice incarnazione (anzi, triplice se consideriamo anche la teoria dei domini, di più stretta pertinenza dell'informatica teorica) di queste utili strutture e delle loro estensioni. Ma ovviamente non è questa la sede per simili considerazioni, per cui si rimanda il lettore interessato, in primissima istanza, al classico [DP02] e relativa bibliografia.

2.7 Rotazioni.

L'operazione elementare alla base della trasformazione di una soluzione stabile in una successiva è la *rotazione*.

Definizione. Una rotazione

$$\rho = (m_0, w_0), (m_1, w_1), \dots, (m_r, w_r)$$

è una lista ordinata di coppie, con $r \geq 1$, appartenenti ad una soluzione stabile SM_p e tali che per ogni i ($0 \leq i \leq r$), $w_{i+1} = SECOND(m_i)$ dove l'incremento di i è preso in modulo $(r + 1)$. In questo caso ρ è una

⁸Il lettore presti attenzione a questo dettaglio: nelle GS-lists (a rigore, già nelle MGS-lists), infatti, tutte le donne con rank minore dell'attuale partner vengono eliminate dall'algoritmo EGS, in modo tale che la partner risulti sempre in prima posizione. Tuttavia, il concetto di *P-set* $P(SM_p)$ prescinde dagli aspetti implementativi e caratterizza l'insieme delle soluzioni *TSM* da un punto di vista puramente algebrico, come anello d'insiemi.

rotazione esposta in SM_p e alla rotazione *appartengono* gli uomini $m_i \in M$ e le donne $w_i \in W$ che appaiono in ciascuna delle coppie elencate.

Molto importante notare che la medesima rotazione può essere esposta in varie soluzioni stabili, ed in generale si dimostra che il numero di rotazioni è molto *inferiore* rispetto al numero complessivo di soluzioni stabili presenti nell'insieme TSM : anche questa proprietà è cruciale nella progettazione dell'algoritmo.

Le rotazioni sono caratterizzate da numerose fondamentali proprietà, che tuttavia qui dobbiamo limitarci ad enunciare senza dimostrazione.

1. C'è una corrispondenza biunivoca tra le rotazioni in TSM e le minime differenze $P(SM_p)$, e tutte le rotazioni compaiono esattamente una volta in ogni catena massimale⁹ del reticolo TSM ;
2. Le rotazioni formano a loro volta un ordine parziale $\Pi(TSM)$, nel quale la relazione d'ordine è definita come segue: la rotazione ρ_i precede ρ_j in $\Pi(TSM)$ se e solo se ρ_i appare prima di ρ_j in *ogni* catena massimale di TSM ;
3. Le rotazioni di $\Pi(TSM)$, adeguatamente applicate, consentono di generare *tutte* le soluzioni stabili in TSM ;
4. Ciascuna coppia (m_i, w_j) con $m_i \in M, w_j \in W$ appare al più in una rotazione. Pertanto, vi sono al più $n(n - 1)/2$ rotazioni in una qualsiasi istanza di SM di dimensione n . D'altro canto, è evidente che ogni singola rotazione ρ_i può contenere al più n coppie.

Consideriamo come esempio la soluzione man-optimal SM_0 e riportiamo per comodità le GS-list:

1: 5 8 3	1: 5 3 6
2: 3 8 6	2: 3 5 7
3: 8 5 1 6 2	3: 1 2
4: 6 8	4: 8
5: 7 2 1	5: 6 7 3 1
6: 1 5	6: 2 3 4
7: 2 5 7	7: 7 5
8: 4	8: 4 1 2 3

Alle righe 1 e 3 si nota immediatamente che $mFIRST(1) = SECOND(3) = 5$ e contemporaneamente $mFIRST(3) = SECOND(1) = 8$. Pertanto la soluzione SM_0 espone la rotazione

$$\rho_0 = (1, 5), (3, 8)$$

riportata anche in figura 4. Non vi sono altre rotazioni esposte in SM_0 .

Applicare una rotazione ρ_j ad una soluzione SM_p , operazione che gli autori denotano come *eliminazione* e indicano con SM_p/ρ_j , significa semplicemente creare nuove coppie tra m_i e $w_{i+1(\text{mod } r+1)} = SECOND(m_i)$ per ogni uomo $m_i \in \rho_j$. Si ottiene così il seguente elenco di nuove coppie, dette *coppie implicite*, che saranno presenti nella soluzione $SM_p/\rho_i = SM_{p+1}$: $(m_0, w_1), (m_1, w_2), \dots, (m_r, w_0)$. Questa semplice rotazione ciclica¹⁰

⁹Ricordiamo che si definisce *catena* un sottoinsieme (proprio) totalmente ordinato di un insieme parzialmente ordinato. La sua rappresentazione entro un diagramma di Hasse è una sequenza di elementi verticali, detta anche diagramma «banale» in quanto tipico degli ordini totali (ad esempio, i numeri naturali sono una catena infinita con diagramma di Hasse banale). Tale catena si dice *massimale* se non esiste alcuna altra catena che la contiene come sottoinsieme proprio. Ciò è strettissimamente correlato con il fondamentale **assioma della scelta**, che possiamo formulare come segue:

in un insieme parzialmente ordinato ogni catena è contenuta in una catena massimale (principio di Hausdorff), o equivalentemente:

se ogni catena di un insieme parzialmente ordinato ha un maggiorante, ogni elemento dell'insieme precede qualche elemento massimale (lemma di Zorn).

¹⁰Si ponga particolare attenzione al fatto che in questo contesto i pedici degli individui vengono utilizzati per contrassegnare la specifica coppia alla quale appartengono entro la rotazione e *non* la loro posizione nelle liste di appartenenza M o W . Gli autori passano in modo piuttosto libero e informale da una notazione all'altra, senza particolari indicazioni, il che talora è potenzialmente fonte di confusione anche se semplifica in modo notevole la descrizione del «funzionamento» di una rotazione a livello di coppie esplicite (m_i, w_i) ed implicite $(m_i, w_{i+1(\text{mod } r+1)})$.

$$\rho_0 = (1, 5), (3, 8)$$

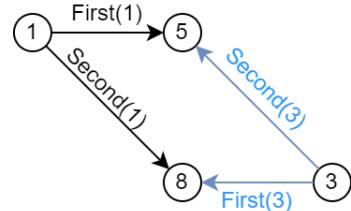


Figura 4: Esempio di rotazione di due coppie.

dei partner, lasciando invariate le coppie che non appartengono a ρ_j , fornisce tutta l'informazione necessaria per generare la soluzione stabile SM_{p+1} da SM_p . Applicando la rotazione ρ_0 appena esemplificata a SM_0 , si ottiene la soluzione $SM_0/\rho_0 = SM_1$:

$$SM_1 = \{(1, 8), (2, 3), (3, 5), (4, 6), (5, 7), (6, 1), (7, 2), (8, 4)\}$$

Come si nota, gli uomini 1 e 3 si sono letteralmente scambiati le partner, scegliendo ciascuno la seconda preferenza nella propria GS-list, che coincide con la precedente partner dell'altro. A questo punto, dopo le necessarie cancellazioni, le nuove GS-lists si presentano come segue:

1:	8	3		1:	5	3	6
2:	3	6		2:	3	5	7
3:	5	1	6	2	1	2	
4:	6	8		4:	8		
5:	7	2	1	5:	6	7	3
6:	1	5		6:	2	3	4
7:	2	5	7	7:	7	5	
8:	4			8:	4	1	

Osservando le righe evidenziate 1, 2, 4 e 3, 6, risulta immediato rilevare le seguenti rotazioni esposte:

$$\rho_1 = (1, 8), (2, 3), (4, 6)$$

$$\rho_2 = (3, 5), (6, 1)$$

Si ha che $SM_2 = SM_1/\rho_1$ e $SM_3 = SM_1/\rho_2$. Si noti come le rotazioni vengono applicate *una alla volta* per generare diverse soluzioni: si faccia riferimento al diagramma di Hasse completo¹¹ per il reticolo delle soluzioni stabili in figura 2. Le uniche altre rotazioni presenti nell'esempio 2, per un totale di 5, sono le seguenti:

$$\rho_3 = (7, 2), (5, 7)$$

$$\rho_4 = (3, 1), (5, 2)$$

Si può già intuitivamente apprezzare l'enorme vantaggio della connotazione dello spazio delle soluzioni come anello d'insiemi caratterizzato da queste operazioni di rotazione, le quali (come si dimostra facilmente) pur essendo in numero ridotto rispetto alle soluzioni (che, lo ricordiamo, possono crescere esponenzialmente con n , nell'ordine di $O(\alpha^n)$ con $\alpha \gg 2$) generano *tutte* le soluzioni stabili.

2.7.1 Coppie stabili.

Dopo avere definito le rotazioni al paragrafo precedente, è possibile concedersi una brevissima digressione (in pratica l'unica dell'intero articolo, per il resto totalmente focalizzato sui concetti utilizzati direttamente nell'algoritmo di Gusfield e Irving) e fornire la definizione di *coppie stabili*.

Definizione. Una coppia (m_i, w_j) con $m_i \in M, w_j \in W$ si dice *stabile* se e solo se compare nella soluzione woman-optimal SM_z oppure compare in qualche rotazione ρ_k .

Alternativamente, si applica la seguente definizione equivalente:

Definizione. Una coppia (m_i, w_j) con $m_i \in M, w_j \in W$ si dice *stabile* se e solo se compare nella soluzione man-optimal SM_0 oppure se è *implicita* in una rotazione $\rho_s = (m_0, w_0), (m_1, w_1), \dots, (m_r, w_r)$, ovvero se è una delle coppie (m_k, w_{k+1}) dove l'incremento dell'indice k è considerato in modulo $(r + 1)$.

Nel nostro esempio 2 le 19 coppie stabili sono il seguente sottoinsieme di $M \times W$:

$$\begin{aligned} &(1, 3) \quad (1, 5) \quad (1, 8) \\ &(2, 3) \quad (2, 6) \\ &(3, 1) \quad (3, 2) \quad (3, 5) \quad (3, 8) \\ &(4, 6) \quad (4, 8) \\ &(5, 1) \quad (5, 2) \quad (5, 7) \\ &(6, 1) \quad (6, 5) \\ &(7, 2) \quad (7, 7) \\ &(8, 4) \end{aligned}$$

Sebbene nell'algoritmo non venga utilizzata direttamente questa definizione, in altri frangenti (es. verifica di stabilità di una o più soluzioni) può comunque essere utile costruire (insieme all'elenco delle rotazioni) una tabella di lookup che ci consenta di sapere in $O(1)$ se una qualsiasi coppia è stabile o meno: tale tabella sarà

¹¹Dal diagramma risulta anche evidente la proprietà 1. secondo la quale tutte le rotazioni compaiono in ciascuna catena massimale, ovvero in tutti i possibili cammini che uniscono l'elemento minimo a quello massimo.

conforme alla seguente matrice binaria SL , con la convenzione che le righe corrispondono agli uomini e che $SL_{m,w} = 1$ se e solo se la coppia (m_r, w_c) appartiene all'elenco delle coppie stabili. Si noti che ciò, dopo una pre-elaborazione a monte in $O(n^2)$ praticamente gratuita se effettuata durante l'elencazione delle rotazioni, rende possibile il controllo di stabilità di una soluzione arbitraria in $O(n)$.

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
m_1	0	0	1	0	1	0	0	1
m_2	0	0	1	0	0	1	0	0
m_3	1	1	0	0	1	0	0	1
m_4	0	0	0	0	0	1	0	1
m_5	1	1	0	0	0	0	1	0
m_6	1	0	0	0	1	0	0	0
m_7	0	1	0	0	0	0	1	0
m_8	0	0	0	1	0	0	0	0

Definizione. Una coppia (m_i, w_j) con $m_i \in M, w_j \in W$ si dice *fissa* se e solo se compare invariata unicamente nelle soluzioni estremali SM_0, SM_z o, equivalentemente, se non compare in alcuna rotazione ρ_k . Nell'esempio 2, la coppia $(8, 4)$ è fissa, come peraltro già notato al paragrafo 2.4.

2.7.2 Fase 2: l'algoritmo MINIMAL-DIFFERENCES per l'enumerazione delle rotazioni.

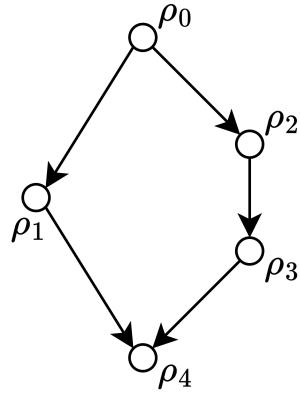
Le rotazioni possono essere elencate esaustivamente con complessità quadratica usando il seguente algoritmo, disponendo unicamente delle soluzioni stabili estremali SM_0, SM_z e delle GS-lists, senza conoscere il grafo orientato completo delle soluzioni o equivalentemente gli accoppiamenti irriducibili derivati dai P-sets 2.6.1. Si noti che questa caratteristica non vale in generale per tutti gli anelli d'insiemi e le relative rappresentazioni tramite differenze, ma è una peculiare proprietà dello spazio delle soluzioni TSM .

La pseudofunzione `NEXT(m)`, usata dagli autori del testo, equivale a `wLAST(SECOND(m))`, ossia il partner attuale della seconda entry nella lista di preferenza di m . L'algoritmo, riportato così come esposto nel testo originale [GI89], percorre implicitamente una delle catene massimali del reticolo delle soluzioni TSM , garantendo così di elencare tutte le rotazioni. Con riferimento all'esempio 2 e al relativo diagramma 2, l'algoritmo percorre la catena massimale di soluzioni $\{SM_0, SM_1, SM_2, SM_4, SM_6, SM_7\}$, come peraltro è immediato evincere dalla nomenclatura rigidamente sequenziale delle rotazioni ρ_i associate agli archi orientati lungo tale percorso.

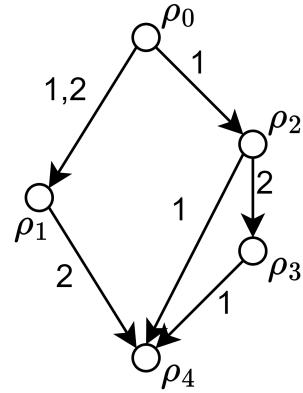
La sequenza di `push()` e `pop()` nel blocco 2 può a prima vista disorientare il lettore e apparire superflua: perché salvare sullo stack l'intera rotazione (a rigore, solo gli uomini che ne fanno parte) per poi estrarla, invertita? In realtà, è molto semplice convincersi con un controsenso che esistono casi nei quali il criterio di variazione utilizzato per la `push()` nel blocco 1 a monte non individua necessariamente un membro della rotazione correntemente elaborata. Nell'esempio 2, quando si giunge ad elaborare la soluzione SM_4 e quindi la rotazione $\rho_3 = (5, 7), (7, 2)$, il valore di x al momento della `push()` è pari a 3 poiché $mP(3, SM_4) = 1$ che differisce da $mP(3, SM_z) = 2$, tuttavia il 3 non compare nella rotazione.

Vale la pena di notare che l'algoritmo modifica le GS-lists ad ogni passaggio, riducendole ad una singola colonna quando giunge al termine della catena massimale visitando la soluzione woman-optimal SM_z . Gli autori sorvolano tranquillamente su questo aspetto, probabilmente dando per scontato che le varie fasi dell'elaborazione vengano implementate tramite eseguibili distinti, il che presumibilmente avveniva all'epoca della stesura dei loro articoli. Tuttavia, tali liste servono poi (nella loro versione integrale, come prodotta nella prima fase di elaborazione dall'algoritmo EGS 2.5.1) per la costruzione del grafo orientato \bar{G} , dunque occorre creare una copia al momento opportuno, con un sostanziale raddoppio dell'occupazione di memoria: il che significa, per le istanze di grandi dimensioni che costituiscono l'oggetto del nostro interesse, che con ogni probabilità in una implementazione *real-world* vi sarà un passaggio per la memoria secondaria.

Di seguito lo pseudocodice fornito dagli autori del testo.



$\Pi(TSM)$



$G(TSM)$

Figura 5: I diagrammi di Hasse relativi a $\Pi(TSM)$ e $G(TSM)$ per l'esempio 2.

```

procedure MINIMAL-DIFFERENCES
{trova  $SM_0$ ,  $SM_z$  e le GS-lists}
call EXTENDED_GS (2.5.1)
initialize stack;
i := 0;
x := 1;
while  $x \leq n$  do
begin
{Blocco 1}
if stack is empty then
begin
while  $(mP(x, SM_i) = mP(x, SM_z)) \text{ and } (x \leq n)$  do
 $x := x + 1$ ;
if  $x \leq n$  then
push(x);
end; {x è l'uomo con indice più basso che ha cambiato partner tra  $SM_i$  e  $SM_z$ }
{Blocco 2}
if stack is not empty then
begin
m:=top();
m:=NEXT(m);
while m not in stack do
begin;
push(m);
m:=NEXT(m);
end;
m':=pop();
set up list  $\rho_i := (m', m\text{FIRST}(m'))$ ;
while  $m \neq m'$  do
begin
m':=pop();
 $\rho_i := (m', m\text{FIRST}(m')), \rho_i$ 
end;
output  $\rho_i$ ;
 $SM_{i+1} := SM_i / \rho_i$ ;
 $i := i + 1$ ;
update GS-lists;
end;
end;

```

2.8 Fase 3: la costruzione del grafo orientato \bar{G} .

La figura 5 mostra il diagramma $\Pi(TSM)$ per il reticolo delle rotazioni dell'esempio 2 e quello equivalente $G(TSM)$, la cui chiusura transitiva è sempre $\Pi(TSM)$ ma risulta computazionalmente e algoritmicamente più vantaggioso da gestire in modo efficiente. Si noti la peculiare marcatura degli archi in $G(TSM)$, che mostra gli archi di «tipo 1» e «tipo 2». Si noti in particolare come un arco può lecitamente avere *ambedue* le etichettature contemporaneamente, fatto che influenza direttamente la costruzione del grafo.

Si dimostra facilmente che, mentre l'unico limite per il numero di coppie nella relazione d'ordine di $\Pi(TSM)$ è pari a $O(n^4)$, $G(TSM)$ può avere al massimo $O(n^2)$ archi (il numero di nodi resta invariato rispetto a $\Pi(TSM)$, essendo quindi al più pari come già accennato a $n(n - 1)/2$). Tuttavia, anche queste limitazioni non sono sufficienti a garantire l'efficienza desiderata per la generazione esaustiva nei casi di grandi dimensioni che qui ci interessano e che gli autori avevano in mente nella seconda metà degli anni Ottanta.

Occorre quindi definire un ulteriore grafo orientato ridotto $\bar{G}(TSM)$, sempre con la medesima chiusura transitiva, che però presenta la richiesta limitazione sul numero massimo di archi uscenti da ciascun singolo nodo, in questo caso pari ad n . Quest'ultimo grafo molto sparso è quello che sarà effettivamente utilizzato per la generazione efficiente delle soluzioni.

In questo caso specifico le regole per la costruzione del grafo orientato sparso $G(TSM)$ delle rotazioni vengono brevemente fornite dagli autori della monografia solo in forma testuale, nell'ambito di una dimostrazione costruttiva della complessità computazionale dell'algoritmo. La costruzione finale di $\bar{G}(TSM)$ è poi descritta come parziale modifica ad una delle regole costruttive fornite per $G(TSM)$.

Si fornisce qui invece una versione in pseudocodice, coerente con le altre presenti nel resto della monografia. L'algoritmo di costruzione del grafo sparso fa uso delle GS-lists, come già accennato, così come prodotte nella prima fase di elaborazione dall'algoritmo EGS.

```

procedure BuildGraph
    {Marcatura dei nodi di tipo 1}
    for each rotation  $\rho_i$  do
        begin
            for each  $m_j, w_j \in \rho_i$  do
                begin
                    label woman  $w_j$  in  $m_j$ 's list with  $i$ , type 1;
                end;
        end;
    {Marcatura dei nodi di tipo 2}
    for each rotation  $\rho_i$  do
        begin
            for each  $m_j, w_{j+1} \in \rho_i$  do
                begin
                    for each  $m$  on  $w_{j+1}$ 's list with  $wRank(w_{j+1}, m_{j+1}) < wRank(w_{j+1}, m) < wRank(w_{j+1}, m_j)$  do
                        begin
                            label woman  $w_{j+1}$  in  $m$ 's list with  $i$ , type 2;
                        end;
                end;
        end;
    {Costruzione del grafo  $\bar{G}(TSM)$ }
    for each man  $m \in M$  do
        begin
             $\rho_0 :=$ label of the first woman in  $m$ 's list;
            for each other woman  $w$  in  $m$ 's list do
                begin
                     $\rho_1 :=$ current woman's label;
                    if type=1 then
                        begin
                            add edge  $\rho_0 \rightarrow \rho_1$ ;
                             $\rho_0 := \rho_1$ ;
                        end;
                    if type=2 and  $\rho_1$  has not appeared before in  $m$ 's list
                        begin
                            add edge  $\rho_1 \rightarrow \rho_0$ ;
                        end;
                end;
        end;
    end;

```

L'ultimo passaggio隐含的 di questo algoritmo consiste nel banale calcolo degli *in-degree* per ciascun nodo del grafo, vale a dire il numero complessivo di archi entranti in ciascun nodo, come richiesto in input dalla fase successiva.

2.9 Fase 4: la generazione delle soluzioni stabili.

Dopo numerose pagine dense di preliminari, definizioni e algoritmi preparatori, sia pure ridotti al minimo indispensabile, siamo infine giunti all'ultima fase: la vera e propria enumerazione delle soluzioni. L'algoritmo proposto da Gusfield e Irving in [GI89] è basato su una duplice ricorsione con *backtracking* e fa uso delle seguenti strutture dati principali:

- $N[][]$ è l'array bidimensionale degli *out-neighbors* per ciascuna rotazione ρ : una rotazione ρ_i appare in $N[\rho][k]$ se e solo se esiste in $\bar{G}(TSM)$ un arco da ρ a ρ_i ;
- $L[]$ è un array che contiene le rotazioni esposte nella soluzione stabile corrente. Inizialmente (L_0 secondo la notazione degli autori) contiene le rotazioni in SM_0 , la soluzione man-optimal (caratterizzate nel grafo $\bar{G}(TSM)$ dall'avere *in-degree* pari a zero);

- $D[]$ è l'array degli *in-degree*, ovvero del numero totale di archi entranti nel nodo, per ciascuna rotazione ρ in $\bar{G}(TSM)$. Anche qui gli autori usano la notazione D_0 per lo stato iniziale, perché tali valori vengono modificati e poi ripristinati per backtracking tra le due chiamate ricorsive;
- SM_n è la soluzione stabile corrente, inizializzata a SM_0 (che viene stampata a monte, direttamente al termine dell'elaborazione in EGS 2.5.1).

```

begin
  find and output  $SM_0$ ;
  construct  $\bar{G}(TSM)$ ;
   $D_0[ ] :=$  in-degree of each rotation in  $\bar{G}(TSM)$ ;
   $L_0[ ] :=$  list of rotations exposed in  $SM_0$ ;
  GetStableMatchings( $SM_0$ ,  $L_0$ ,  $D_0$ );
end.
procedure GetStableMatchings( $SM$ ,  $L$ ,  $D$ );
begin {This is the start of block 1}
  if  $L[ ]$  is nonempty then
    begin
      remove rotation  $\rho$  from the head of  $L[ ]$ ;
       $SM := SM/\rho$ ;
      output  $SM$ ;
      for each rotation  $\pi$  in  $N[\rho]$  do
        begin
           $D[\pi] := D[\pi] - 1$ ;
          if  $D[\pi] = 0$  then append  $\pi$  to the end of  $L[ ]$ ;
        end;
      GetStableMatchings( $SM$ ,  $L$ ,  $D$ ); {End of block 1}
      for each rotation  $\pi$  in  $N[\rho]$  do
        begin
           $D[\pi] := D[\pi] + 1$ ;
          if  $D[\pi] = 1$  then remove the last rotation from  $L[ ]$ ;
        end;
       $SM := SM * \rho$ ;
      GetStableMatchings( $SM$ ,  $L$ ,  $D$ ); {End of block 2}

      restore  $\rho$  to the head of  $L[ ]$ ;
    end;
  end;
end;

```

Gli autori dimostrano facilmente che l'algoritmo sopra riportato, così come presentato in [GI89] a pag. 125, visita esattamente una volta tutte le soluzioni nel reticolo TSM tramite una visita completa del grafo orientato ridotto delle rotazioni $\bar{G}(TSM)$. Il metodo seguito per la generazione è un metodo diretto, perfettamente consequenziale a quanto discusso in merito alle rotazioni e alla loro presenza sulle catene massimali di TSM : poiché è ormai intuitivamente chiaro (e parimenti semplice da dimostrare) che ad ogni rotazione esposta in ciascuna soluzione stabile corrisponde una catena contenente una o più soluzioni stabili subordinate, il metodo enumerativo consiste nel percorrere una alla volta queste catene, partendo dalle rotazioni esposte nella soluzione man-optimal. Ovviamente tale metodo, per non incorrere in duplicazioni, deve tenere dinamicamente conto delle sole rotazioni consentite, ed è esattamente questo il ruolo degli array ausiliari $L[]$ (gestito come una vera e propria FIFO) e $D[]$.

L'operazione $SM * \rho$ che compare nella fase di backtracking, quando le modifiche vengono annullate e si prepara la seconda chiamata ricorsiva, rappresenta il *ripristino* della rotazione ρ , ossia l'operazione complementare alla cancellazione, ed è l'operazione tramite la quale vengono eliminate dalla soluzione SM_{i+1} le coppie implicite in ρ e aggiunte le coppie che vi compaiono esplicitamente, in modo da ottenere la soluzione precedente SM_i .

2.9.1 La struttura dati per il grafo $\bar{G}(TSM)$.

Come ormai i lettori hanno ben compreso, la monografia [GI89] non è certo un testo applicativo. Gli autori si concentrano sugli aspetti algebrici ed algoritmici, forniscono dimostrazioni (prevalentemente costruttive) contenenti indicazioni sulla complessità computazionale delle varie fasi di elaborazione, ma non si sbilanciano in

indicazioni implementative di qualsiasi sorta. Anche in questa sede non è certo possibile affrontare nel dettaglio le tematiche di ottimizzazione in spazio e in tempo che condizionano una eventuale scelta di progetto legata ad una ben definita piattaforma e un dato linguaggio. Si possono solamente fornire alcune indicazioni di massima come prima valutazione, da approfondire in sede progettuale sulla base dei comuni parametri ingegneristici e dei vincoli implementativi (es. occupazione massima di memoria, come era il caso all'epoca in cui gli autori hanno concepito l'algoritmo).

Nonostante quanto sopra asserito, spesso nel testo le principali strutture dati ipotizzate traspaiono tra le righe, anche se non vengono referenziate in modo esplicito. Questo è il caso anche dell'algoritmo `GetStableMatchings()`, che appare sostanzialmente concepito pensando alle liste di adiacenza LdA (vedi fig. 6): l'array $N[][]$ degli *out-neighbors* utilizzato dagli autori sostanzialmente non è altro che una lista di adiacenze per ogni nodo ρ_i , nella quale una rotazione ρ_j compare se e solo se esiste un arco $\rho_i \rightarrow \rho_j$. I due loop di scansione del «vicinato» `for each rotation π in $N[\rho]$` do raggiungono la loro massima efficienza con l'uso di una lista concatenata o di un array contenente tutti e soli i riferimenti ai nodi connessi direttamente al nodo corrente da un singolo arco, poiché tale lista - per costruzione - non può avere più di n nodi. Per contro, il codice si arricchisce di numerosi dettagli e la complessità ciclomatica cresce a causa della natura dinamica di tali liste e dei controlli necessari ad evitare le collisioni (si ricordi che un arco può avere ambedue le marcature 1 e 2, ma il relativo nodo di destinazione deve comunque figurare in lista solamente una volta), anche se è possibile mitigare in parte tali effetti negativi usando una singola preallocazione e emulando le LdA tramite un array bidimensionale con un massimo di $n(n - 1)/2$ righe¹² e un numero costante di colonne, pari ad n , con uno *slack* generalmente accettabile in prima approssimazione (soprattutto in un ambito puramente illustrativo come il presente).

L'immediata alternativa all'uso di LdA è la matrice booleana di adiacenza MdA: una matrice quadrata, triangolare superiore (ricordiamo che il grafo $\bar{G}(TSM)$ è orientato), nella quale ciascuna cella $a_{r,c}$ contiene un valore *TRUE* se e solo se esiste un arco diretto $r \rightarrow c$. Il codice per la costruzione si semplifica in modo drastico, come sarà possibile vedere a breve (nell'esempio dimostrativo di codice si è scelto di fare uso di tale matrice, sia pure implementata in modo *naive*, proprio in nome della semplificazione estrema consentita). Le collisioni, anche quando non gestite per evitare statement condizionali aggiuntivi (che intralcianno il lavoro delle pur sofisticate BPU), divengono irrilevanti in quanto si riducono ad innocue sovrascritture di valori *TRUE* con *TRUE*. Il limite dimensionale superiore per tale matrice quadrata è chiaramente $(n^2 - n)^2/4$, tuttavia una tale matrice può essere validamente compressa con un rapporto 1 : 64 o superiore (in funzione della dimensione della word sulla piattaforma target) gestendola come *bitarray* per il quale le funzioni booleane di impostazione, azzeramento, inversione, controllo di un bit sono sostanzialmente atomiche sulla maggioranza delle piattaforme mainstream grazie al ricorso ad *intrinsic* e *builtins* dei compilatori, e/o ad apposite *performance libraries* accuratamente ottimizzate. Purtroppo il contrappasso per tali vantaggi, l'unico reale *overhead* introdotto dall'uso di una tale struttura dati, si verifica nei due loop di scansione del vicinato, dove può giungere ad ipotecare molto pesantemente le prestazioni dell'intera funzione ricorsiva. Occorre pertanto un'analisi approfondita e puntuale prima di decidere in favore di questa implementazione per una applicazione realmente produttiva, eventualmente supportata da una valutazione dinamica a runtime del rapporto tra la dimensione del problema n e il numero effettivo di nodi presenti: quando tale numero è dell'ordine di $O(n)$ l'uso di una MdA è plausibilmente quasi sempre la scelta migliore in termini di performance e occupazione di memoria, almeno a livello di principio.

Con riferimento all'esempio 2 e al grafo di $G(TSM)$ di figura 5, e seguendo la convenzione già richiamata di porre sulle *righe* i nodi di *partenza* degli archi orientati, la relativa matrice booleana di adiacenza assume il seguente aspetto:

	ρ_0	ρ_1	ρ_2	ρ_3	ρ_4
ρ_0	0	1	1	0	0
ρ_1	0	0	1	0	1
ρ_2	0	0	0	1	1
ρ_3	0	0	0	0	1
ρ_4	0	0	0	0	0

Come è evidente anche per il lettore meno familiare con le strutture dati utilizzate per la rappresentazione di grafi, il numero di archi uscenti da ogni singolo nodo (*out-degree*) è dato dalla somma dei valori sulla relativa

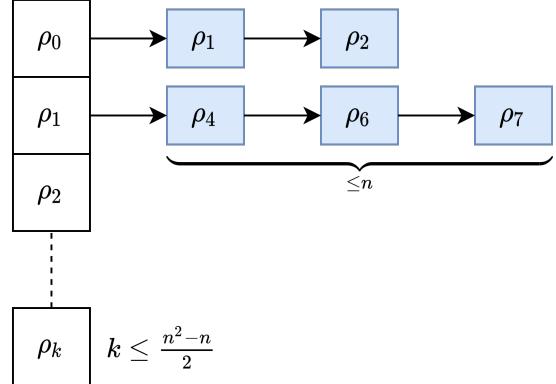


Figura 6: Esempio di lista di adiacenze per $\bar{G}(TSM)$.

¹²Tale matrice sarà ovviamente allocata dinamicamente, una volta noto il numero effettivo delle rotazioni.

riga, mentre quello di archi entranti (in-degree) è dato dalla somma per colonna. Si ricorda ancora una volta (a costo di apparire pedanti) che sebbene nell'esempio illustrato non vi sia differenza nel numero di archi tra i grafi G e \bar{G} , solo quest'ultimo garantisce per costruzione un massimo di n archi uscenti da ciascun nodo, mentre il numero di nodi al caso peggiore è dell'ordine di $O(n^2)$, pertanto nel caso medio la matrice di adiacenza sarà molto sparsa e prestazionalmente svantaggiosa da trattare rispetto alle liste di adiacenza.

3 Implementazione di esempio in C89.

Come accennato in apertura, non risulta esservi pubblicamente disponibile sul web alcuna implementazione dell'algoritmo esaustivo di Gusfield e Irving. Senza alcuna particolare pretesa, se non quella di illustrare meglio le varie fasi dell'algoritmo qui sommariamente descritto, mostrandolo all'opera su strutture dati definite in modo concreto, si è deciso di proporre una **implementazione esemplificativa** in linguaggio C89 basata sulle seguenti ovvie decisioni di progetto:

1. Gestione al più basso livello possibile delle strutture dati astratte richieste dall'algoritmo, in modo da sottolinearne le numerose peculiarità e nel pieno rispetto delle possibilità concesse dai sistemi e linguaggi esistenti all'epoca della stesura della monografia. Certo oggi, in uno scenario operativo realistico, sarebbe assai più sbrigativo utilizzare linguaggi come Python, lasciando al compilatore tutta la gestione del lavoro «sotto il cofano» con le liste di preferenze e i numerosi grafi gestiti: certamente però non avrebbe il medesimo valore didattico e di approfondimento del punto centrale dell'algoritmo proposto, ossia la gestione ottimizzata di strutture dati dinamiche non standard.
2. Massima semplificazione del codice e rinuncia a priori a qualsiasi ottimizzazione, tranne le più banali e intuitive.
3. Uso abbondante di variabili ausiliarie ridondanti, per aumentare la leggibilità del codice. Ciò vale in particolare per puntatori e indicizzazioni: l'accesso in loop annidati alle strutture dati costituisce infatti la maggior parte delle operazioni compiute dal codice stesso.
4. Uso di macro autoesplicative, allineate con la nomenclatura utilizzata negli algoritmi e per le pseudofuzioni (es. SECOND(), LAST(), NEXT(), ...).
5. Uso sistematico di variabili condivise (raggruppate in una singola struttura con scope globale) e di **allocazione statica**: praticamente nessuna attenzione è stata posta all'effettivo ciclo di vita delle strutture dati più ingombranti e all'avvicendamento delle liste di preferenze tra le varie fasi elaborative, per evitare di distrarre il lettore con eccessivi dettagli di bassa cucina. L'allocazione dinamica è limitata al minimo indispensabile, si veda in particolare la sottosezione 3.1. Si è del tutto evitato il passaggio di parametri a tutte le funzioni principali, non solo (come concettualmente d'obbligo) per la routine doppiamente ricorsiva. Si è inoltre optato per una sostanziale (quanto innocua: parliamo di circa una dozzina di LOC) duplicazione del codice nell'algoritmo EGS 2.5.1, sempre in nome della semplificazione e al fine di evitare dispersioni nella gestione a monte dello scambio delle liste.
6. Drastica limitazione a priori delle dimensioni dei problemi trattabili. Non è infatti intenzione dell'Autore svolgere i «compiti per casa» di qualcuno o proporre una implementazione *turnkey*, ma solo illustrare in modo comprensibile un algoritmo - certo non complesso ma comunque basato su idee non banali né intuitive per chi non frequenta spesso il mondo dell'ottimizzazione e della generazione combinatoria, come confermato da una lunga esperienza didattica. Peraltra si tratta di un algoritmo oggettivamente articolato e decisamente poco o nulla implementato, almeno a livello di sorgenti liberamente consultabili sul web. Pertanto, ad esempio:
 - (a) La dimensione effettiva del problema è definita come costante di preprocessore.
 - (b) I dati delle preferenze sono contenuti in un file di include (facilmente sostituibile, anzi la sperimentazione con altri dataset è *fortemente incoraggiata*, ma richiede comunque una ricompilazione).
 - (c) Il tipo di dati utilizzati quasi ovunque (per le preferenze, i totalizzatori, le strutture correlate al grafo $\bar{G}(TSM)\dots$) nonché lo stesso valore sentinella BLANK consentono al più 254 membri per ogni insieme.
 - (d) Stampe e visualizzazioni sono volutamente limitate ai casi con un decina di partecipanti, sempre in nome della semplificazione estrema del codice d'esempio.
 - (e) Le considerazioni esposte in forma condensata al paragrafo 2.9.1 hanno guidato la scelta verso una semplice matrice di adiacenza booleana, in nome della semplificazione estrema e leggibilità del codice, pur nella piena consapevolezza del notevole *overhead* da essa introdotto al caso peggiore nell'ultima fase dell'algoritmo. Il medesimo criterio di semplificazione del codice ha indotto ad una implementazione *naive* di codesta matrice, rinunciando al ricorso ai bitarray.

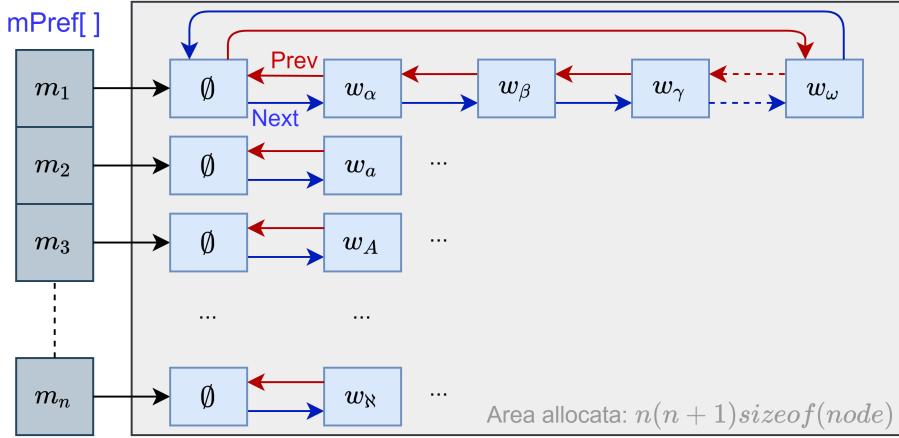


Figura 7: Schema logico dei puntatori per la lista di preferenze maschili.

(f) Osservando la figura 7, risulta immediata la ridondanza dell'array `mPref[]` di nodi di tipo `head`. Normalmente, infatti, sarebbe preferibile inglobare direttamente nell'array i nodi dummy di testa, per risparmiare così sull'allocazione di memoria. Si è invece scelto diversamente per maggiore chiarezza didattica e per non dover essere vincolati a realizzare un array di struct di tipo `node_t` (il puntatore `next` dell'ultimo nodo, come tutti gli altri, deve obbligatoriamente puntare ad una struttura di tale tipo) o a realizzare poco simpatici camuffamenti sintattici per ingannare il compilatore. In questo modo risulta possibile aggiungere campi e, in generale, si è pienamente liberi di utilizzare una struct differente per fronteggiare ulteriori specifiche di una applicazione reale. Lo slack di memoria, anche per dimensioni del problema di ordine 10^4 , è comunque assai poco significativo.

Una volta doverosamente esposte queste banali premesse, comunque implicite nella definizione stessa di «implementazione di esempio», vediamo rapidamente le principali strutture dati e alcune significative porzioni del codice, ovviamente limitate alle funzioni principali già illustrate a livello algoritmico.

3.1 La struttura dati principale: lista di preferenze e array di lookup.

Abbiamo già menzionato a più riprese il fatto che gli algoritmi EGS 2.5.1 e MINIMAL-DIFFERENCES 2.7.2 hanno necessità di effettuare delle cancellazioni (potenzialmente molto numerose) in posizioni arbitrarie dalle liste di preferenza, ovviamente preservandone l'ordine: stando ai manuali elementari, ciò implicherebbe il ricorso ad una lista concatenata, semplice o doppia. Tuttavia, allo stesso tempo, le già illustrate pseudofunzioni come `mRank()` o `SECOND()` appaiono fin dall'origine ideate per operare su array o comunque strutture ad accesso diretto: sarebbe assolutamente inconcepibile che dovessero ogni volta scorrere linearmente una linked list per raggiungere il dato interessato, a maggior ragione dopo tanto sforzo intellettuale degli autori per trovare una caratterizzazione algebrica dello spazio delle soluzioni e una rappresentazione sotto forma di grafo orientato sparso visitabile in modo efficiente. Inoltre la dualità dei ruoli di proponente richiesta da EGS impone di usare il medesimo tipo di struttura per ambedue le liste, senza poter sfruttare il fatto che ogni cancellazione, dal lato del ricevente, avviene unicamente a partire da «destra». Infine, nessuna fase dell'algoritmo richiede inserimenti o aggiunte arbitrarie alle liste di preferenze.

In considerazione di quanto sopra accennato e di altre motivazioni secondarie, le strutture dati principali sono esemplificativamente realizzate come segue:

```
/* Nodo header di lista , per ogni persona P */
typedef struct {
    node_t *first;
    uint8_t total;
} head_t;

/* Nodo della lista dinamica delle preferenze (DLL) */
typedef struct node {
    struct node *next;
    struct node *prev;
    marriage_t val;
} node_t;
```

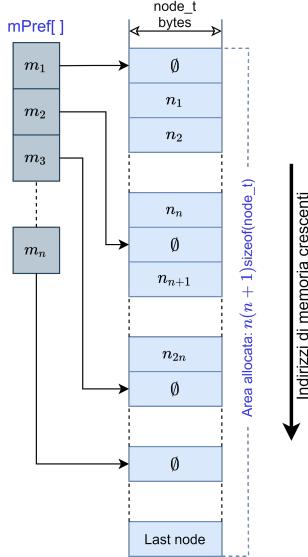


Figura 8: Allocazione in memoria dei nodi per le preferenze in un'unica area contigua.

```
/* Le vere e proprie liste di preferenze maschili e femminili */
head_t mPref[PROBLEM_SIZE];
head_t wPref[PROBLEM_SIZE];

/* Array di lookup inverso */
node_t *mRank[PROBLEM_SIZE][PROBLEM_SIZE];
node_t *wRank[PROBLEM_SIZE][PROBLEM_SIZE];
```

Si faccia riferimento alla figura 7, la quale riporta a titolo illustrativo la struttura *logica* delle liste di preferenze maschili (quelle femminili sono ovviamente identiche), tenendo sempre presente che tutti i nodi delle liste concatenate sulla destra sono in realtà *strettamente contigui* in memoria: le strutture *node_t* vengono allocate dinamicamente con una singola operazione per ambedue le liste maschili e femminili, quindi *in un unico blocco contiguo*¹³ che contiene gli n^2 nodi per ciascuna lista, più n nodi dummy per un totale di $2n(n + 1)$ nodi (ossia, nel codice, $2 * \text{PROBLEM_SIZE} * (\text{PROBLEM_SIZE} + 1) * \text{sizeof}(\text{node_t})$). Ciò è illustrato con chiarezza in figura 8. Il nodo *dummy* in testa a ciascuna lista circolare è utilizzato per armonizzare e semplificare le operazioni di cancellazione quando eseguite sull'effettivo primo nodo della lista.

Dato che i nodi sono strettamente contigui in memoria, in fase di inizializzazione per ogni locazione di *mPref[]* e *wPref[]* si procede banalmente a scorrere l'intera sequenza di strutture usando direttamente l'aritmetica dei puntatori e collegando ogni nodo ai suoi vicini, richiudendo poi la lista in modo che l'ultimo nodo punti al nodo *dummy* in testa, il cui campo *val* contiene un valore sentinella, definito come *BLANK*. In questo modo, al termine dell'inizializzazione, si avrà che ciascuna locazione *mPref[m]* referencia la relativa lista doppiamente concatenata di preferenze dell'uomo *m*, come rappresentato in figura 7, e lo stesso ovviamente vale per *wPref[w]*. Tutto ciò per consentire la cancellazione *order-preserving* di nodi in posizioni arbitrarie in $O(1)$, grazie anche agli array di reverse lookup *mRank[][]* e *wRank[][]* che - al costo di una inevitabile occupazione aggiuntiva di memoria pari a $O(n^2)$ - garantiscono l'implementazione in tempo costante delle pseudofunzioni *mRank()* e *wRank()* utilizzate nella descrizione degli algoritmi, evitando così ogni ricerca lineare tipicamente associata all'uso di semplici liste concatenate.

Il campo *total*, inizialmente valorizzato con la costante di preprocessore *PROBLEM_SIZE* (pari alla dimensione del problema) e poi aggiornato ad ogni cancellazione, è utilizzato per semplificare la gestione e rendere maggiormente leggibili alcuni loop di scansione delle preferenze.

Si omette per brevità l'intera definizione della struttura denominata *SM*, la quale in breve raccoglie tutte le variabili con visibilità globale, rendendole immediatamente riconoscibili in tutto il sorgente come membri di tale struttura, al prezzo di un'appena percettibile complicazione sintattica.

¹³Vale la pena di sottolineare, ad uso degli studenti più giovani, che questo tipo di allocazione in pool è il più usato nel real world, specialmente nelle applicazioni che a runtime creano e distruggono dinamicamente decine di migliaia di oggetti (ad esempio CAD/CAM, EDA, GIS, FEM e molti altri), le quali spesso e volentieri sostituiscono completamente le primitive di allocazione offerte dal SO con proprie versioni dedicate.

3.2 La funzione di cancellazione.

Grazie a tutte queste scelte progettuali, in realtà piuttosto ovvie, la funzione di cancellazione si riduce a quanto segue:

```

1  /*************************************************************************/
2  /*
3  ** Delete(m, w): cancella m dalla lista di w, e viceversa, in tempo costante.
4  */
5  void Delete(const marriage_t m, const marriage_t w)
6  {
7      node_t *p;
8
9      /* Cancella la donna w dalla lista di m */
10     p = SM.mRank[m][w];
11     SM.mPref[m].total -= 1;
12     /* Si usa la tecnica dei dancing links di Knuth */
13     p->prev->next = p->next;
14     p->next->prev = p->prev;
15
16     /* Cancella l'uomo m dalla lista di w */
17     p = SM.wRank[w][m];
18     SM.wPref[w].total -= 1;
19     p->prev->next = p->next;
20     p->next->prev = p->prev;
21 }
```

Si notino in particolare i seguenti aspetti:

- L'accesso al nodo desiderato è immediato, grazie all'array di lookup.
- Il totalizzatore è del tutto opzionale, ma ha un valore didattico che può essere sfruttato nelle stampe di controllo per seguire meglio la dinamica del codice. In ogni caso, il suo aggiornamento ha un costo decisamente trascurabile.
- La «cancellazione» consiste semplicemente nell'aggiornamento di due puntatori. Questa tecnica porta il nome di «dancing links» [Knu00]: lo spazio in memoria non viene deallocated, e la catena delle strutture (allocate in modo contiguo) può comunque essere percorsa linearmente con l'aritmetica dei puntatori perché i dati originali sono intatti, mentre la catena dei link riflette la situazione corrente e fa apparire cancellato il nodo, escludendolo logicamente dalla concatenazione. Con l'uso di questa tecnica sarebbe anche possibile, al medesimo costo, ripristinare il nodo all'interno della lista concatenata, nelle situazioni in cui ciò risulti necessario (non è comunque il nostro caso).
- La cancellazione di un nodo in posizione notevole (prima, ultima) non comporta alcuna eccezione né un indebito aumento della complessità ciclomatica del codice tramite aggiunta di statement condizionali. Grazie alla presenza del nodo *dummy* in testa alla lista, la cancellazione del primo nodo utile in testa non differisce in alcun modo dalle altre; grazie alla effettiva circolarità della lista stessa, i puntatori *prev* e *next* di ciascun nodo sono sempre congruenti ed entro i limiti.

3.3 Implementazione dell'algoritmo EGS.

Una volta implementata in modo massimamente efficiente la funzione di cancellazione, siamo pronti per dare uno sguardo all'implementazione dell'algoritmo EGS 2.5.1. Come accennato, si è scelto di accoppare esplicitamente ambedue i passaggi in un'unica funzione, evitando del tutto il passaggio di una serie di parametri e relativa gestione a monte al costo di una risibile duplicazione del codice, che però rende maggiormente comprensibile l'intera funzione.

```

1  /*************************************************************************/
2  /*
3  ** Algoritmo di Gale & Shapley esteso EGS.
```

```

4  */
5 void FindGSLists( void )
6 {
7     node_t *p;
8     size_t i;
9     marriage_t w0, w2, m0, m2;
10
11    for ( i = 0; i < PROBLEM_SIZE; ++i ) {
12        Push( i );
13        SM. Solution[ i ] = BLANK;
14        SM. F_Optimal[ i ] = BLANK;
15    }
16
17 /* Produce le MGS-List e la soluzione man-optimal */
18 while (SM. stk_ptr > 0) {
19     m0 = Pop();
20     w0 = wFIRST(m0);
21     m2 = SM. Solution[w0];
22
23     if (BLANK != m2) {
24         Push(m2);
25     }
26
27     /* Aggiorna gli array: lookup e soluzione man-optimal. */
28     SM. Solution[w0] = m0;
29     SM. M_Optimal[m0] = w0;
30
31     /*
32      ** Cancella tutte le coppie instabili formate con gli uomini
33      ** "a destra" di m0 nella lista di w0.
34      */
35     p = SM. wPref[w0]. first->prev;
36     while (p->val != m0) {
37         Delete(p->val, w0);
38         p = p->prev;
39     }
40 }
41
42 for ( i = 0; i < PROBLEM_SIZE; ++i ) {
43     Push( i );
44 }
45
46 /*
47 ** Il secondo passaggio di EGS produce SMz e le GS-List invertendo i
48 ** proponenti e usando le MGS-Lists come input.
49 */
50 while (SM. stk_ptr > 0) {
51     w0 = Pop();
52     m0 = wFIRST(w0);
53     w2 = SM. F_Optimal[m0];
54
55     if (BLANK != w2) {
56         Push(w2);
57     }
58
59     SM. F_Optimal[m0] = w0;
60
61     /*
62      ** Cancella tutte le coppie instabili formate con le donne
63      ** "a destra" di w0 nella lista di m0.
64      */

```

```

65     p = SM.mPref[m0].first->prev;
66     while (p->val != w0) {
67         Delete(m0, p->val);
68         p = p->prev;
69     }
70 }
71
72 /*
73 ** Prepara l'array di lavoro SM.Solution per la discesa alla
74 ** ricerca delle rotazioni, che inizia da SM0 (man-optimal).
75 */
76 memcpy(SM.Solution, SM.M_Optimal, PROBLEM_SIZE);
77
78 /*
79 ** Facoltativo: aggiorna la matrice delle coppie stabili con le
80 ** coppie in SMz.
81 */
82 for (i = 0; i < PROBLEM_SIZE; ++i) {
83     SM.StableCouples[i][SM.F_Optimal[i]] = TRUE;
84 }
85
86 /*
87 ** Aggiorna le liste secondarie destinate alla creazione del grafo orientato
88
89 ** L'algoritmo MINIMAL_DIFFERENCES modifica le liste per poter percorrere
90 ** una catena massimale dalla soluzione man-optimal a quella woman-optimal
91 ** alla ricerca delle rotazioni, tuttavia la creazione del grafo richiede le
92 ** GS-list così come prodotte in questa fase di elaborazione.
93 ** In un contesto applicativo, con decine di migliaia di elementi nei due
94 ** insiemi, sarà ovviamente opportuno e necessario un salvataggio su
95 ** memoria secondaria.
96 */
97 DuplicateData();
}

```

Il codice è sostanzialmente privo di sorprese e conforme al semplice algoritmo EGS. Risultano presenti alcuni banali passaggi aggiuntivi di *housekeeping*, tutti abbondantemente commentati. Nel primo passaggio, l'array globale `SM.Solution[]` funziona da reverse lookup per velocizzare le operazioni, rispondendo alla domanda «chi è il partner attuale della donna w_0 ?». Nel secondo passaggio, a parti invertite, tale ruolo è naturalmente svolto dal medesimo array che contiene la soluzione, `SM.F_Optimal[]`.

Si noti il meccanismo della cancellazione dalle liste di preferenze tramite puntatori: si parte dall'ultima locazione, $wLAST(w_0)$ (si ricordi che per ogni uomo e per ogni donna la lista delle preferenze è circolare, l'ultimo nodo è quindi il predecessore del primo) e si procede a ritroso fino a trovare per valore il nuovo partner, m_0 .

La seconda sezione è speculare alla prima, ovviamente a parti invertite (questa volta l'insieme proponente è W), e ciò porta in definitiva, come già ben spiegato, alla creazione delle GS-lists per ambedue i gruppi, nonché alle due soluzioni estremali SM_0, SM_z che serviranno per le successive fasi di elaborazione.

3.4 L'implementazione di MINIMAL_DIFFERENCES.

Una tipica scelta scolastica, influenzata anche dalle descrizioni nella monografia che qui trattiamo, avrebbe facilmente condotto ad implementare l'elenco delle rotazioni come una lista di liste, oppure un array di liste. Si è ritenuto che tale scelta non dovesse essere presa in considerazione ai fini della semplificazione del codice e della drastica minimizzazione delle allocazioni dinamiche, per non distrarre il lettore con codice di fatto scorrelato da quello essenziale degli algoritmi esposti. L'array `Rho[][]` è quindi un semplice array bidimensionale statico, dimensionato a priori secondo le ormai note formule worst case per il numero di rotazioni, ossia dei nodi nel grafo $\Pi(TSM)$:

```
rnode_t Rho[MAX_ROTATIONS][PROBLEM_SIZE+1];
```

La locazione aggiuntiva al termine di ogni riga è utilizzata per memorizzare il numero di coppie di ciascuna rotazione, che è come noto variabile. Inutile aggiungere che in una applicazione real world, con decine di migliaia di proponenti, questi dati sarebbero allocati dinamicamente e senza sprechi, dopo una pre-elaborazione per il conteggio delle rotazioni: altrettanto inutile aggiungere che in questa sede una tale complessificazione del codice

sarebbe inutile quanto deleteria. Coerentemente con le decisioni di progetto fin qui assunte, anche la soluzione adottata per l'aggiunta delle rotazioni in testa alla lista è grezzamente sbrigativa quanto efficace, basandosi su un semplice array ausiliario locale e su un banalissimo accorgimento di indicizzazione a ritroso. Al di là di questi aspetti, invero poco significativi, il codice è perfettamente conforme allo pseudocodice dell'algoritmo MINIMAL_DIFFERENCES 2.7.2 e, sperabilmente, del tutto comprensibile anche per gli studenti più giovani.

```

1  /************************************************************************/
2  /*
3   ** Ricerca le rotazioni percorrendo una catena massimale di TSM.
4   */
5 void FindRotations( void )
6 {
7     /* Buffer temporaneo per il deque della rotazione corrente. */
8     rnode_t Rho[PROBLEM_SIZE];
9
10    marriage_t m0 = 0;
11    size_t rotation = 0;
12
13    while (m0 < PROBLEM_SIZE) {
14        DisplaySolution(SM.Solution, "## ");
15        /*
16         ** Ricerca l'uomo con indice minore che cambia partner tra
17         ** la soluzione attuale e quella woman-optimal.
18         */
19        if (SM.stk_ptr == 0) {
20            while ((SM.Solution[m0] == SM.F_Optimal[m0]) &&
21                   (m0 < PROBLEM_SIZE)) {
22                ++m0;
23            }
24            if (m0 < PROBLEM_SIZE) Push(m0);
25        }
26
27        if (SM.stk_ptr > 0) {
28            size_t i, j;
29            size_t idx = PROBLEM_SIZE-1;
30            marriage_t m1, m2;
31
32            m1 = NEXT(Top());
33
34            while (!IsInStack(m1)) {
35                Push(m1);
36                m1 = NEXT(m1);
37            }
38
39            /* Estrae la rotazione dallo stack LIFO. */
40            do {
41                m2 = Pop();
42                Rho[idx].m = m2;
43                Rho[idx--].w = mFIRST(m2);
44            } while (m1 != m2);
45
46            /* Salva la rotazione corrente rho_n. */
47            ++idx;
48            for (j = 0, i = idx; i < PROBLEM_SIZE; ++i, ++j) {
49                SM.Rho[rotation][j].m = Rho[i].m;
50                SM.Rho[rotation][j].w = Rho[i].w;
51                SM.StableCouples[Rho[i].m][Rho[i].w] = TRUE;
52            }
53
54            /* Memorizza il numero di coppie di rho(n) in una locazione
      riservata. */

```

```

55     SM.Rho[ rotation ][ PROBLEM_SIZE ].m = j ;
56
57     /*
58      ** Applica la rotazione appena trovata per produrre la soluzione
59      ** successiva sulla catena massimale che sta percorrendo
60      ** implicitamente.
61      ** Contestualmente aggiorna anche le liste di preferenza.
62      */
63     for ( i = 0; i < j; ++i ) {
64         node_t *p;
65         marriage_t w0 = SM.Rho[ rotation ][ ( i+1)%j ].w;
66
67         /* Aggiorna la soluzione corrente con la nuova coppia. */
68         SM.Solution[SM.Rho[ rotation ][ i ].m] = w0;
69
70         /* Aggiorna le liste di preferenza ridotte. */
71         p = SM.wPref[ w0 ].first->prev;
72         while ( p->val != SM.Rho[ rotation ][ i ].m ) {
73             Delete( p->val, w0 );
74             p = p->prev;
75         }
76         ++rotation;
77     }
78 }
79 SM.TotalRho = rotation;
80 }
```

Si noti, in coda al codice, in cosa consiste effettivamente l'applicazione della rotazione ρ_k (la cosiddetta «cancellazione» SM_i/ρ_k) per trasformare la soluzione SM_i in quella successiva SM_{i+1} , seguita dal medesimo meccanismo di cancellazione multipla per l'aggiornamento delle liste di preferenza già visto nella funzione `FindGSLists()`. Rimane ovviamente inteso che, in una applicazione reale, al termine dell'elaborazione si provvederà opportunamente anche al riciclo dello spazio di memoria occupato dalle liste di preferenze primarie e relative tabelle di lookup, giunte qui al termine del loro ciclo di vita nell'applicazione.

3.5 La costruzione del grafo sparso \bar{G} : `BuildGraph()`.

La funzione fa un uso delle liste di preferenze sostanzialmente diverso da quanto vista finora: totalmente statico, senza cancellazioni o altre modifiche. Si limita ad accedere direttamente alle locazioni interessate e ad aggiornare dei campi supplementari per l'etichettatura, utilizzati nel terzo loop per la creazione degli archi. Questa caratteristica ci consente di utilizzare una struttura dati più semplice e convenzionale per le liste «secondarie», ossia una semplice matrice bidimensionale di strutture.

Si è già discusso della scelta di utilizzare una matrice di adiacenza per la rappresentazione interna del grafo orientato ridotto delle rotazioni: si apprezzi qui l'assoluta semplificazione e linearità del codice che ne deriva, nel terzo loop della funzione riportata, pur nella consapevolezza dell'overhead introdotto e quindi del valore precipuamente illustrativo e didattico di una tale decisione progettuale.

L'array booleano temporaneo `nodes[]` viene utilizzato esclusivamente per un controllo in $O(1)$ durante la marcatura dei nodi di tipo 2: per ogni uomo m , se esiste già un nodo con la medesima label (indifferentemente di tipo 1 o 2) del nodo corrente di tipo 2, quest'ultimo viene ignorato.

```

1 /*****
2 */
3 ** Costruzione del grafo sparso G_ tramite matrice d'adiacenza .
4 ** Fa uso di liste di preferenze ridotte modificate, salvate
5 ** prima dell'esecuzione di FindRotations().
6 */
7 boole_t BuildGraph( void )
8 {
9     size_t i , j , p;
10    size_t n_rho;
```

```

11 marriage_t m0, m1, w0;
12 boole_t *nodes;
13
14 /* Marca i nodi di tipo 1 con le coppie presenti in ciascuna rotazione. */
15 for (i = 0; i < SM.TotalRho; ++i) {
16     n_rho = SM.Rho[i][PROBLEM_SIZE].m;
17     for (j = 0; j < n_rho; ++j) {
18         m0 = SM.Rho[i][j].m;
19         w0 = SM.Rho[i][j].w;
20         p = SM.smRank[m0][w0];
21         SM.smPref[m0][p].label = (label_t)i;
22         SM.smPref[m0][p].type = 1;
23     }
24 }
25
26 /* Marca i nodi di tipo 2 con le coppie implicite in ciascuna rotazione. */
27 for (i = 0; i < SM.TotalRho; ++i) {
28     n_rho = SM.Rho[i][PROBLEM_SIZE].m;
29     for (j = 0; j < n_rho; ++j) {
30         size_t k;
31         w0 = SM.Rho[i][(j+1) % n_rho].w;
32         /* Il nuovo partner di w0 */
33         m0 = SM.Rho[i][j].m;
34         /* Il partner attuale di w0 */
35         m1 = SM.Rho[i][(j+1) % n_rho].m;
36         k = SM.swRank[w0][m0] + 1;
37         while (SM.swPref[w0][k].val != m1) {
38             m0 = SM.swPref[w0][k].val;
39             p = SM.smRank[m0][w0];
40             SM.smPref[m0][p].label = (label_t)i;
41             SM.smPref[m0][p].type = 2;
42             ++k;
43         }
44     }
45 }
46
47 nodes = (boole_t *)malloc(SM.TotalRho * sizeof(boole_t));
48 if (NULL == nodes) {
49     fputs("Errore di allocazione array nodes[]!\n", stderr);
50     return FALSE;
51 }
52
53 /*
54 ** Costruisce il grafo orientato sparso G_ usando una matrice di adiacenza.
55 */
56 for (i = 0; i < PROBLEM_SIZE; ++i) {
57     label_t rho0 = SM.smPref[i][0].label;
58
59     /* Inizializza l'array delle occorrenze. */
60     memset(nodes, FALSE, SM.TotalRho * sizeof(boole_t));
61     nodes[rho0] = TRUE;
62     j = 1;
63
64     while (SM.smPref[i][j].val != BLANK) {
65         label_t lbl = SM.smPref[i][j].label;
66
67         switch (SM.smPref[i][j].type) {
68             case 1:
69                 SM.AdjMatrix[rho0][lbl] = TRUE;
70                 nodes[lbl] = TRUE;
71                 rho0 = lbl;

```

```

72         break ;
73
74     case 2:
75     /*
76      ** Controlla se esiste già sulla riga un nodo con la
77      ** medesima label:
78      ** in caso affermativo, ignora questo nodo di tipo 2 (pag.
79      ** 113).
80      ** Questa regola differenzia il grafo G_ dal grafo G e
81      ** garantisce
82      ** che l'outdegree massimo di ciascuno nodo sia minore
83      ** di PROBLEM_SIZE.
84     */
85     if (nodes[lbl] == FALSE) {
86         SM.AdjMatrix[lbl][rho0] = TRUE;
87         nodes[lbl] = TRUE;
88     }
89     break ;
90 }
91
92 free(nodes);
93
94 /*
95 ** Popola l'array degli Indegree e delle rotazioni esposte in SM0
96 ** (hanno tutte Indegree zero).
97 */
98 SM.Q_top = 0;
99 SM.Q_bottom = 0;
100 for (i = 0; i < SM.TotalRho; ++i) {
101     size_t j;
102     for (j = 0; j < SM.TotalRho; ++j) {
103         SM.Indegree[i] += SM.AdjMatrix[j][i];
104     }
105
106     if (0 == SM.Indegree[i]) {
107         SM.Exp_rho[SM.Q_bottom++] = i;
108     }
109 }
110
111 return TRUE;
112 }
```

3.6 la generazione delle soluzioni: GetStableMatchings().

Il codice della funzione `GetStableMatchings()` è stato mantenuto il più vicino possibile allo pseudocodice presentato alla sottosezione 2.9. L'array `SM.Exp_rho[]`, come già accennato, viene gestito come FIFO con aggiunte in coda ed estrazioni in testa grazie ai due indici ausiliari `SM.Q_top` e `SM.Q_bottom`: quando coincidono, la FIFO è vuota.

L'unico aspetto notevole rispetto a quanto già visto con lo pseudocodice, in pratica, è l'immediata conseguenza dell'uso della matrice di adiacenza `SM.AdjMatrix[][]` che impone ai due loop `for ()` alle linee 32 ÷ 39 e 45 ÷ 52 una lunghezza fissa costantemente pari al numero totale di rotazioni `SM.TotalRho` e una `if()` aggiuntiva all'interno del loop, il che nel caso peggiore comporta una sicura penalità prestazionale rispetto all'uso di liste di adiacenza, come già accennato in particolare al paragrafo 3.1.

Le uniche variabili locali significative, al di là delle necessarie variabili di induzione, sono in pratica la ρ prevista dall'algoritmo (contenente la rotazione estratta dalla FIFO, sotto forma di piccolo intero) e la relativa lunghezza n_r in termini di numero di coppie.

```

1  /*************************************************************************/
2  /*
3  ** Fase finale dell' algoritmo che enumera tutte le soluzioni stabili
4  ** visitando il grafo sparso delle rotazioni G_.
5  */
6 void GetStableMatchings(void)
7 {
8     size_t i , j ;
9     uint8_t rho;
10    uint8_t n_r;
11    char buff[8];
12
13    if (SM.Q_bottom == SM.Q_top)
14    {
15        return ;
16    }
17
18    rho = SM.Exp_rho[SM.Q_top++];
19    n_r = SM.Rho[rho][PROBLEM_SIZE].m;
20
21    /*
22    ** Applica la rotazione per produrre la soluzione SM(i+1)
23    */
24    for (i = 0; i < n_r; ++i) {
25        /* Aggiorna la soluzione corrente con le nuove coppie. */
26        SM.Solution[SM.Rho[rho][i].m] = SM.Rho[rho][(i+1) % n_r].w;
27    }
28    sprintf(buff, "SM%02u: ", SM.soln_cnt++);
29    DisplaySolution(SM.Solution, buff);
30
31    /* Scansione del "vicinato" della rotazione corrente. */
32    for (i = 0; i < SM.TotalRho; ++i) {
33        if (SM.AdjMatrix[rho][i] == TRUE) {
34            SM.Indegree[i] -= 1;
35            if (0 == SM.Indegree[i]) {
36                SM.Exp_rho[SM.Q_bottom++] = i;
37            }
38        }
39    }
40
41    /* Prima chiamata ricorsiva. */
42    GetStableMatchings();
43
44    /* Backtracking */
45    for (i = 0; i < SM.TotalRho; ++i) {
46        if (SM.AdjMatrix[rho][i] == TRUE) {
47            SM.Indegree[i] += 1;
48            if (1 == SM.Indegree[i]) {
49                SM.Q_bottom--;
50            }
51        }
52    }
53
54    /*
55    ** Ripristina la rotazione per tornare a SM(i)
56    */
57    for (i = 0; i < n_r; ++i) {
58        SM.Solution[SM.Rho[rho][i].m] = SM.Rho[rho][i].w;
59    }
60
61    /* Seconda e ultima chiamata ricorsiva. */

```

```

62     GetStableMatchings();
63     SM.Q_top--;
64 }
```

3.7 L'output del programma con i dati dell'esempio (2.4).

Fornendo in input al programma le liste di preferenze dell'esempio 2 si ottengono i seguenti risultati, perfettamente conformi a quanto riportato nella monografia [GI89] ed a quanto calcolabile manualmente con gli algoritmi illustrati. Si noti come la numerazione delle soluzioni attribuita dal software durante la discesa ricorsiva è necessariamente diversa da quella utilizzata nel grafo delle soluzioni 2, essendo strettamente legata all'ordine di visita indiretto delle catene massimali di *TSM* imposto dalla valutazione ricorsiva delle rotazioni esposte in ciascuna soluzione stabile via via calcolata.

```

>> Algoritmo di Gusfield e Irving per la generazione
>> esaustiva delle soluzioni stabili di SM.

***** Liste di preferenze date:
***** Uomini           Donne
** 1: 5 7 1 2 6 8 4 3   5 3 7 6 1 2 8 4
** 2: 2 3 7 5 4 1 8 6   8 6 3 5 7 2 1 4
** 3: 8 5 1 4 6 2 3 7   1 5 6 2 4 8 7 3
** 4: 3 2 7 4 1 6 8 5   8 7 3 2 4 1 5 6
** 5: 7 2 5 1 3 6 8 4   6 4 7 3 8 1 2 5
** 6: 1 6 7 5 8 4 2 3   2 8 5 3 4 6 7 1
** 7: 2 5 7 6 3 4 8 1   7 5 2 1 8 6 4 3
** 8: 3 8 4 5 7 2 6 1   7 4 1 5 2 3 6 8

>> STEP 1: Elaborazione GS-lists e soluzioni estremali SM0, SMz.

***** Soluzioni estremali:
SM0: 5 3 8 6 7 1 2 4
SMz: 3 6 2 8 1 5 7 4

***** Liste di preferenze ridotte:
***** Uomini           Donne
** 1: 5 8 3             5 3 6
** 2: 3 8 6             3 5 7
** 3: 8 5 1 6 2         1 2
** 4: 6 8               8
** 5: 7 2 1             6 7 3 1
** 6: 1 5               2 3 4
** 7: 2 5 7             7 5
** 8: 4                 4 1 2 3

>> STEP 2: Elenco delle rotazioni rho.

***** Rotazioni totali.....: 5
***** Elenco delle rotazioni:
rho0: (1,5),(3,8)
rho1: (1,8),(2,3),(4,6)
rho2: (3,5),(6,1)
rho3: (7,2),(5,7)
rho4: (3,1),(5,2)

>> STEP 3: Creazione del grafo sparso G_.

Matrice di adiacenza 5x5 del grafo G_:
 0 1 1 0 0
 0 0 0 0 1
 0 0 0 1 1
```

0	0	0	0	1
0	0	0	0	0

>> STEP 4: Elenco delle soluzioni stabili.

SM00:	5	3	8	6	7	1	2	4
SM01:	8	3	5	6	7	1	2	4
SM02:	3	6	5	8	7	1	2	4
SM03:	3	6	1	8	7	5	2	4
SM04:	3	6	1	8	2	5	7	4
SM05:	3	6	2	8	1	5	7	4
SM06:	8	3	1	6	7	5	2	4
SM07:	8	3	1	6	2	5	7	4

4 Conclusioni e ringraziamenti.

In queste pagine si è cercato di illustrare in modo succinto ed essenziale, ma sperabilmente corretto, l'algoritmo di generazione esaustiva delle soluzioni stabili al problema SM elaborato da Dan M. Gusfield e Robert W. Irving. Abbiamo condensato in appena trenta pagine (di cui almeno dieci dedicate ad una implementazione di esempio) l'esposizione di concetti, definizioni, idee, principi di funzionamento che ha richiesto in origine oltre metà di una monografia che consta di 240 pagine - peraltro di sola teoria, essendo la monografia stessa per sua natura del tutto priva di esempi di codice, come già più volte richiamato. Questo ovviamente implica che qui si sono potuti solo accennare gli elementi fondamentali, peraltro rinunciando a tutte le dimostrazioni: si esorta caldamente il lettore interessato allo studio integrale della fondamentale monografia.

Si sono schematizzate le quattro fasi fondamentali dell'elaborazione, esponendo in modo estremamente sintetico le idee e i concetti più rilevanti, limitando i tecnicismi ed espungendo ogni dimostrazione col fine di raggiungere la più vasta platea di lettori, con particolare riguardo agli studenti più giovani.

Pensando a istanze del problema dell'ordine di grandezza di $10^3 \div 10^5$ proponenti si è costantemente sottolineato quali scelte di design algoritmico conducono alle prestazioni in $O(n^2 + n \cdot |TSM|)$ che caratterizzano i vari algoritmi preparatori e la fase finale di enumerazione. Tali algoritmi sono stati accompagnati da brevissime indicazioni sulle principali caratteristiche di ciascuno e sulle strutture dati astratte che gli autori avevano plausibilmente in mente al momento della stesura.

Nella seconda parte si è proposta una implementazione esemplificativa, una vera e propria *proof of concept* in forma volutamente e adeguatamente limitata alle sole funzioni principali, con sorgenti in linguaggio C89 (ANSI/INCITS X3.159-1989, ISO/IEC 9899:1990) che dovrebbero risultare compilabili senza modifiche rilevanti su un vasto numero di piattaforme: anche retroinformatiche, per chi desiderasse sperimentare nelle condizioni originali in cui sono stati elaborati e testati gli algoritmi, attorno alla metà degli anni Ottanta e quindi sui primi XT, AT, Amiga, Atari, Unix dipartimentali e vari altri sistemi midi o mainframe di DEC, IBM, Bull, Honeywell e altri.

Si è fatto cenno, con la presentazione delle funzioni principali dell'implementazione, ad alcune idee fondanti, evidenziando a scopo didattico la relativa sofisticazione delle strutture dati impiegate e delle corrispondenti tecniche allocative e implementative per l'efficiente gestione (in particolare i "dancing links") rispetto ai dispersivi e inefficienti approcci classici dei manuali elementari e, ahinoi, di numerosi corsi introduttivi.

Nella speranza che gli studenti più giovani e i practitioners interessati abbiano tratto spunti costruttivi dalla lettura e stimolo per ulteriori approfondimenti, l'Autore ringrazia i lettori per il tempo dedicato al presente articolo.

Un sentito ringraziamento speciale va ai professori Dan M. Gusfield e Robert W. Irving: innanzi tutto per il loro lavoro, che costituisce una vera e propria pietra miliare nella storia del problema SM. Una traduzione del presente articolo è stata sottoposta alla loro attenzione: generosamente e con grande signorilità hanno dedicato del tempo alla sua accurata lettura ed hanno fornito preziosi consigli, indicazioni, suggerimenti assieme ad alcune fondamentali precisazioni.

5 Appendice: intervista al professor Robert W. Irving.

Contattato per informarlo del presente articolo, il professore ha anche risposto ad alcune domande, che qui riportiamo in lingua originale e, in calce, in una traduzione a cura dell'Autore che lo ringrazia per la sua straordinaria disponibilità.

1) Speaking of the stable marriage problem, you and Dan Gusfield have rightly deserved your place among the protagonists of its history together with Gale, Shapley, McVitie, Wilson, Roth and Knuth, a name which certainly needs no introduction. How did your interest in this problem arise? What motivated your research?

Robert Irving: In 1981 I attended a conference on Combinatorial Optimization held at the University of Stirling (in Scotland). One of the conference organisers was Les Wilson (of McVitie and Wilson), who gave a survey talk on “The stable marriage assignment problem”. This was my first encounter with the concept, and I found it very intriguing. Sometime later I came across Knuth’s book (in French), and was especially interested in his set of open problems. The one that particularly inspired me was the Stable Roommates problem, and in due course I developed and published my polynomial-time algorithm for this problem (Journal of Algorithms 1985), which was my first publication in this area.

I subsequently worked with my student Paul Leather to exploit the concept of rotation (first introduced under a different name in the Roommates paper) in the Stable Marriage context, resulting in two papers (SIAM Journal on Computing 1986 and Journal of the ACM 1987). Dan Gusfield was asked to referee this latter paper, and because he could see scope for significant improvements, he asked the journal editor for permission to approach Paul and myself directly, and as a consequence he became a joint author on the paper. So began our fruitful collaboration.

2) As an eminent scholar, what is your feeling in having provided such a fundamental contribution to one of the most well-known and important problems in the history of combinatorics?

R.I.: Well, it’s a nice feeling! It’s very rewarding to see the way the subject area has developed and expanded over the years.

3) Professor Irving, your first paper on the Stable Marriage problem dates back to 1986 (SIAM Journal on Computing 15, no. 3), while your joint, rightly famous monograph with Professor Gusfield was printed in 1989. During these years of work, how did you develop the fundamental intuition about the algebraic nature of the problem as a distributive lattice? More generally, how would you describe in a few words the fundamental ideas on which your innovative algorithm is based?

R.I.: Well, we weren’t the first to note the lattice structure. Knuth attributes this observation to John Conway. However, we were the first to exploit the compact representation of the lattice as a rotation poset. My focus was on the practical use of the rotations to facilitate algorithmic developments, while Dan formulated the more abstract view as a ring of sets.

4) The majority of our readers, for age or cultural reasons, are very interested in retrocomputing. As it often happens in research work since the Sixties (or even before), the mathematical and formal part is normally aided by the use of computers: to develop examples, test implementations of the algorithms, process the data and so on. So, was this your case as well? Do you remember any episodes of those days related to hardware, languages and systems from the 1980s that possibly helped you fine-tune your work?

R.I.: The Roommates paper includes a full implementation (in Pascal) of the Roommates algorithm. I was always interested in implementations of the algorithms, since I enjoyed programming so much. The referees of that paper actually suggested that the implementation be removed, but I resisted. However, at that time it was certainly unusual for research papers in Algorithmics to include fully coded implementations, and we did not try to do this in subsequent papers. The implementation of a whole range of stable matching algorithms became a valuable source of undergraduate, and even graduate, student projects over many years.

5) The exhaustive generation algorithm that you developed, as we know, is the most efficient to date for the original instance of the Gale & Shapley problem. Over the last three decades, it has certainly been implemented

and used in numerous circumstances, both in private industry and in the public sector. Over the years, how often did you happen to receive some direct feedback from developers, engineers and computer scientists who were actually implementing the algorithm? Maybe requests for explanations or clarifications on application details?

R.I.: I was involved for over ten years in the Scottish version of the NRMP (for much of that time with David Manlove, who had been my PhD student and postdoc.). I did virtually all of the coding for the programs that were used in this scheme, and it was hugely rewarding to see the results of our fundamental research being used in an important practical context. However, this implementation work was almost entirely focused on finding optimal stable matchings in various contexts, rather than on generating all stable matchings.

6) In your opinion, what are the most important open questions regarding the stable marriage problem? What advice would you like to give to a young researcher or scholar who wants to take an interest in the problem today?

R.I.: In my view, one of the most challenging open questions is to find some way of improving on exhaustive search for the generation of all stable matchings when ties are allowed in the preference lists. There is no lattice structure to exploit in this context. It may be, in some sense, provably hard. For example, it is not known whether there is a polynomial-time algorithm to determine whether an instance of Stable Marriage with Ties has more than two stable matchings. (There is a polynomial-time algorithm for the question of whether more than one stable matching exists – but even this is non-trivial). If the former problem turned out to be NP-complete then the prospects for any kind of efficient generation algorithm would be bleak. But even so, an algorithm that works reasonably well in practice, in most cases, would be useful and may well be feasible. David Manlove's 2013 monograph contains an 'open problems' section at the end of each chapter, and many of these problems remain unsolved to this day. This would provide a good starting point for anyone interested in research in this area.

Traduzione italiana dell'intervista.

1) Parlando del problema dei matrimoni stabili, il suo nome e quello di Gusfield hanno giustamente meritato il loro posto tra i protagonisti della sua storia assieme a Gale, Shapley, McVitie, Wilson, Roth e Knuth, un nome che certo non ha bisogno di presentazioni. Come nasce il vostro interesse in questo problema? Cosa ha motivato la vostra ricerca?

Robert Irving: Nel 1981 ho partecipato ad una conferenza sull'ottimizzazione combinatoria tenutasi presso l'Università di Stirling (in Scozia). Uno degli organizzatori della conferenza era Les Wilson (della coppia di autori McVitie e Wilson), che ha tenuto un discorso su "Il problema dei matrimoni stabili". Questo è stato il mio primo incontro con il concetto e l'ho trovato molto intrigante. Qualche tempo dopo mi sono imbattuto nel libro di Knuth (in francese) e sono rimasto particolarmente interessato alla sua serie di problemi aperti. Quello che mi ha particolarmente ispirato è stato il problema Stable Roommates, e a tempo debito ho sviluppato e pubblicato il mio algoritmo in tempo polinomiale per questo problema (Journal of Algorithms 1985), che è stata la mia prima pubblicazione in quest'area.

Successivamente ho lavorato con il mio studente Paul Leather per sfruttare il concetto di rotazione (introdotto per la prima volta con un nome diverso nell'articolo Roommates) nel contesto del matrimonio stabile, dando vita a due articoli (SIAM Journal on Computing 1986 e Journal of the ACM 1987). A Dan Gusfield fu chiesto di fare da revisore per quest'ultimo articolo e, poiché vedeva la possibilità di miglioramenti significativi, chiese all'editore della rivista il permesso di avvicinarsi direttamente a me e a Paul, e di conseguenza divenne coautore dell'articolo. Così è iniziata la nostra fruttuosa collaborazione.

2) Come eminente studioso, qual è la sensazione nell'aver fornito un contributo così fondamentale ad uno dei problemi più noti ed importanti nella storia della combinatoria?

Robert Irving: Beh, è una bella sensazione! È molto gratificante vedere come l'area tematica si è sviluppata e ampliata nel corso degli anni.

3) La sua prima pubblicazione sul problema risale al 1986 (SIAM Journal on Computing 15, no. 3), mentre la famosa monografia in coppia col Professor Gusfield è stata stampata nel 1989. In questi anni di lavoro,

come avete sviluppato la vostra fondamentale intuizione sulla natura algebrica del problema come reticolo distributivo? Più in generale, come descriverebbe in poche parole le idee fondamentali su cui si basa il vostro innovativo algoritmo?

Robert Irving: Ebbene, non siamo stati i primi a notare la struttura reticolare. Knuth attribuisce questa osservazione a John Conway. Tuttavia, siamo stati i primi a sfruttare la rappresentazione compatta del reticolo come insieme parzialmente ordinato di rotazioni. Io ero focalizzato sull'uso pratico delle rotazioni per facilitare gli sviluppi algoritmici, mentre Dan [Gusfield] ha formulato la visione più astratta come anello di insiemi.

4) La maggioranza dei nostri lettori, per motivi anagrafici o culturali, si interessa di retrocomputing. Come spesso avviene nel lavoro di ricerca, la parte matematica e formale è supportata dall'uso del calcolatore per sviluppare esempi, provare delle implementazioni degli algoritmi, elaborare i dati, etc. Anche per voi è stato così? Ricordate qualche episodio di quegli anni legato ad hardware e sistemi tipici degli Ottanta che vi ha aiutato a mettere a punto il vostro lavoro?

Robert Irving: L'articolo sul problema dei Roommates include un'implementazione completa (in Pascal) dell'algoritmo. Sono sempre stato interessato alle implementazioni degli algoritmi, dato che mi piaceva molto programmare. I revisori di quel documento suggerirono effettivamente di rimuovere l'implementazione, ma io mi opposi. Tuttavia, a quel tempo era certamente insolito che gli articoli di ricerca in algoritmica includessero implementazioni completamente codificate, e non abbiamo provato a farlo nei documenti successivi. L'implementazione di un'intera gamma di algoritmi di matching stabili è diventata una preziosa fonte di progetti di studenti universitari e di dottorato nel corso di molti anni.

5) L'algoritmo di generazione esaustiva che avete sviluppato è ad oggi il più efficiente per l'istanza originale del problema di Gale & Shapley. Negli scorsi tre decenni, sicuramente è stato implementato e utilizzato in numerose circostanze, sia nell'industria privata che nel settore pubblico. Nel corso degli anni, quanto spesso avete ricevuto dei feedback da sviluppatori, ingegneri e informatici che stavano effettivamente implementando l'algoritmo? Magari richieste di spiegazioni o chiarimenti su dettagli applicativi?

Robert Irving: Sono stato coinvolto per oltre dieci anni nella versione scozzese dell'NRMP (per gran parte di quel tempo con David Manlove, che era stato il mio studente di dottorato e postdoc). Ho fatto praticamente tutta la codifica per i programmi che sono stati utilizzati in questo schema, ed è stato estremamente gratificante vedere i risultati della nostra ricerca fondamentale utilizzati in un importante contesto pratico. Tuttavia, questo lavoro di implementazione era quasi interamente incentrato sulla ricerca di abbinamenti stabili ottimali in vari contesti, piuttosto che sulla generazione di tutti gli abbinamenti stabili.

6) Secondo lei, quali sono le questioni più importanti ancora aperte riguardo al problema dei matrimoni stabili? Quali consigli potrebbe dare ad un giovane ricercatore che voglia interessarsi oggi del problema?

Robert Irving: A mio avviso, una delle questioni aperte più difficili è trovare un modo per migliorare la ricerca esaustiva per la generazione di tutti i matching stabili quando legami preesistenti sono ammessi negli elenchi di preferenze. Non esiste una struttura reticolare da sfruttare in questo contesto. Potrebbe essere, in un senso preciso, dimostrabilmente difficile. Ad esempio, non si sa se esista un algoritmo in tempo polinomiale per determinare se un'istanza di «Stable Marriage with Ties» abbia più di due matching stabili (esiste un algoritmo in tempo polinomiale per verificare se esista più di un matching stabile, ma anche questo non è banale). Se il problema precedente si rivelasse NP-completo, le prospettive per qualsiasi tipo di algoritmo di generazione efficiente sarebbero scarse. Ma anche così, un algoritmo che funzioni abbastanza bene nella pratica, nella maggior parte dei casi, sarebbe utile e potrebbe essere fattibile. La monografia del 2013 di David Manlove [Man13] contiene una sezione "problemi aperti" alla fine di ogni capitolo e molti di questi problemi rimangono ad oggi irrisolti. Ciò potrebbe costituire un ottimo punto di partenza per chiunque fosse interessato alla ricerca in questo ambito.

Riferimenti bibliografici

- [And02] Ian Anderson, *Combinatorics of finite sets*, Mathematics, Dover Publications, Inc., 2002, "Slightly corrected Dover (2002) republication of the edition published by Oxford University Press, Oxford, UK, and New York, 1989".
- [Bir37] Garrett Birkhoff, *Rings of sets*, Duke Math. J. **3** (1937), no. 3, 443–454.

- [Bir93] ———, *Lattice theory*, American Mathematical Society, Providence, R.I, 1993.
- [DBS13] Ewa Drgas-Burchardt and Zbigniew Switalski, *A number of stable matchings in models of the gale-shapley type*, Discrete Applied Mathematics **161** (2013), no. 18, 2932–2936.
- [DP02] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*, 2 ed., Cambridge University Press, 2002.
- [GI89] Dan Gusfield and Robert W. Irving, *The stable marriage problem: Structure and algorithms*, MIT Press, Cambridge, MA, USA, 1989.
- [GILS87] Dan Gusfield, Robert Irving, Paul Leather, and Michael Saks, *Every finite distributive lattice is a set of stable matchings for a small stable marriage instance*, Journal of Combinatorial Theory, Series A **44** (1987), no. 2, 304–309.
- [Grä11] George Grätzer, *Lattice theory: Foundation*, Dover Publications Inc., 01 2011.
- [GS62] D. Gale and L. S. Shapley, *College admissions and the stability of marriage*, The American Mathematical Monthly **69** (1962), no. 1, 9–15.
- [Gus87] Dan Gusfield, *Three fast algorithms for four problems in stable marriage*, SIAM Journal on Computing **16** (1987), no. 1, 111–128.
- [IL86] Robert W. Irving and Paul Leather, *The complexity of counting stable marriages*, SIAM Journal on Computing **15** (1986), no. 3, 655–667.
- [ILG87] Robert Irving, Paul Leather, and Dan Gusfield, *An efficient algorithm for the "optimal" stable marriage*, J.ACM **34** (1987), 532–543.
- [KG88] J. F. Korsh and L. J. Garrett, *Data structures, algorithms, and program style using c*, PWS-KENT Pub. Co, 1988.
- [Knu00] Donald E. Knuth, *Dancing links*, 11 2000.
- [KT06] J. Kleinberg and É. Tardos, *Algorithm design*, Alternative Etext Formats, Pearson/Addison-Wesley, 2006.
- [Law11] Eugene Lawler, *Combinatorial optimization: networks and matroids*, Dover Books on Mathematics, Dover, Mineola, NY, 2011.
- [Lev66] V. I. Levenshtein, *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*, Soviet Physics Doklady **10** (1966), 707.
- [Lub66] D. Lubell, *A short proof of sperner's lemma*, Journal of Combinatorial Theory **1** (1966), no. 2, 299.
- [Man13] David Manlove, *Algorithmics of matching under preferences*, World Scientific Publishing Company, 2013.
- [MW71a] D. G. McVitie and L. B. Wilson, *The stable marriage problem*, Communications of the ACM **14** (1971), no. 7, 486–490.
- [MW71b] ———, *Three procedures for the stable marriage problem*, Communications of the ACM **14** (1971), no. 7, 491–492.
- [PTW09] George Polya, Robert E. Tarjan, and Donald R. Woods, *Notes on introductory combinatorics*, Birkhäuser Basel, 2009.
- [RN09] Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach*, 3rd ed., Prentice Hall Press, USA, 2009.
- [RS90] Alvin E. Roth and Marilda A. Oliveira Sotomayor, *Two-sided matching: A study in game-theoretic modeling and analysis*, Econometric Society Monographs, Cambridge University Press, 1990.
- [Sed88] R. Sedgewick, *Algorithms (2nd ed.)*, Addison-Wesley Longman Publishing Co., Inc., USA, 1988.
- [Sed01] Robert Sedgewick, *Algorithms in c, parts 1-5 (bundle)*, Pearson Education (US), 2001.
- [Spe28] E. Sperner, *Ein satz über untermengen einer endlichen menge*, Mathematische Zeitschrift **27** (1928), 544–548.

- [Sto36] M. H. Stone, *The theory of representations for boolean algebras*, Journal of Symbolic Logic **1** (1936), no. 3, 118–119.
- [Thu02] Edward G. Thurber, *Concerning the maximum number of stable matchings in the stable marriage problem*, Discrete Mathematics **248** (2002), no. 1, 195–219.
- [Wir76] Niklaus Wirth, *Algorithms + data structures = programs*, Prentice-Hall, 1976.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

Le problème des ménages

Sommario

Una breve digressione informatica sul problema proposto nel 1891 da Édouard Lucas nel suo classico “*Théorie des nombres*” [Luc91]. In questo contesto vengono introdotti alcuni algoritmi combinatori, dovuti a: S. G. Akl [Akl80], S. Effler & F. Ruskey [ER03], B. R. Heap [Hea63], D. E. Knuth [Knu05], K. Mikawa & K. Tanaka [MT14], nonché J. S. Rohl [Roh78] con successiva rielaborazione a cura di P. Ganapathi & B. Rama [GR10]; tutti accompagnati dalle relative implementazioni illustrate in linguaggio C.

«È più facile quadrare un circolo che arrotondare un Matematico»
(Augustus de Morgan, 1806-1871)

1 Introduzione.

Scopo del presente articolo è l’illustrazione di alcuni algoritmi combinatori comparsi in letteratura in varie epoche, scelti arbitrariamente nel contesto ludico di un gradevole e stimolante problema ottocentesco. Il linguaggio è intenzionalmente informale, divulgativo e didascalico, scevro da particolari pretese di rigore, al fine di raggiungere il più vasto uditorio informatico possibile e in particolare gli studenti più giovani.

Il materiale qui presentato costituisce una riorganizzazione ed un ampliamento di quanto già discusso sul *blog* dell’autore «Titolo provvisorio...» nella serie di articoli aperta da «Il valzer delle coppie». Tutti i sorgenti in linguaggio C ai quali qui si accenna sono presentati sul *blog* in forma integrale.

L’articolo è organizzato in tre parti principali. Nella prima parte si ricordano molto succintamente alcune definizioni e formule essenziali in combinatorica, evitando accuratamente ogni dimostrazione e senza insistere in modo troppo pedissequo sui vari passaggi algebrici.

Nella seconda parte, con inizio a pag. 8, si illustra il problema dei *ménages* nella sua forma originale, proponendo anche una brevissima cronistoria delle soluzioni e relativi affinamenti apparsi in letteratura.

Nella terza e ultima parte, a partire da pag. 14, si presentano alcuni utili algoritmi combinatori e relativi codici sorgenti in linguaggio C, partendo dal contesto del problema dato, come riassunto nel seguente specchietto.

Anno	Algoritmo	Paragrafo	Pag.
1963	Heap	4.2.1	16
1978	Rohl	4.2.3	19
1980	Akl	4.2.2	17
2003	Effler-Ruskey	5.1	24
2005	X di Knuth	4.2.4	20
2010	Rohl-Ganapathi-Rama	5.3	38
2014	Mikawa-Tanaka	5.2	27

2 Preliminari e definizioni.

L’idea alla base delle presenti note introduttive è che la materia sia in gran parte già nota al lettore. Le permutazioni e loro derivazioni sono senza dubbio uno degli oggetti matematici in assoluto più conosciuti e studiati: in parte a causa del loro isomorfismo con i gruppi simmetrici di ordine n , in parte per le innumerevoli applicazioni in informatica, dai banali anagrammi a problemi avanzati di ottimizzazione, *routing*, *packaging* e affini. Il relativo materiale di riferimento, ad ogni livello qualitativo e formale, è disponibile in quantità decisamente esorbitanti e non avrebbe alcun senso duplicarlo in questa sede.

Per evitare tediose e pedantesche ripetizioni, si danno quasi ovunque per scontati almeno due tra i presupposti più banali in qualsiasi testo e manuale di matematica discreta e combinatorica: gli unici «numeri» che qui rilevano sono sempre e soltanto naturali ossia *numeri interi non negativi*, salvo eventuali eccezioni che saranno ben specificate contestualmente. Allo stesso modo, insiemi e intervalli sono sempre e unicamente *finiti* e di conseguenza i relativi indici sono sempre implicitamente *opportuni*. Come in altre occasioni, rinunciamo anche

a dare una definizione formale di insieme, affidandoci invece all'idea vaga ma intuitiva che si tratti di una collezione non ordinata di oggetti matematici, nel senso più ampio dell'espressione. Diamo inoltre per scontati i simbolismi di base di qualsiasi manuale di matematica discreta per insiemi, intervalli, liste ordinate e affini; come pure la corrispondenza biunivoca di qualsiasi insieme (finito!) con l'intervallo dei primi n numeri naturali, che ci consente di usare come insieme dei simboli $\{1, 2, 3, \dots, n\}$ (rispettivamente $\{0, 1, 2, \dots, n-1\}$) laddove opportuno) senza alcuna perdita di generalità. Per il resto, si fa senz'altro appello all'intuito del lettore ed alle convenzioni normalmente diffuse in letteratura.

In ogni caso, l'autore rimane perfettamente consapevole che - nonostante la cura impiegata nella sua realizzazione - si può assicurare con assoluta sicurezza che il presente articolo contiene almeno un errore, come recita il notissimo *paradosso della prefazione*¹ del logico David C. Makinson [Mak65].

2.1 Permutazioni semplici.

Dato un insieme finito contenente n elementi distinti, si dicono *permutazioni semplici*² le presentazioni ordinate che si possono formare in modo da soddisfare contemporaneamente i seguenti criteri:

1. Ogni presentazione deve contenere tutti gli n elementi, ciascuno considerato una e una sola volta;
2. Ogni presentazione deve differire dalle altre solo e unicamente per l'ordine degli elementi.

In altri termini, una permutazione è una regola che a ogni elemento dell'insieme dato associa un naturale (un ordinale finito) che ne descrive la posizione, in modo univoco - ovvero, in ultima analisi, essa costituisce un **criterio di ordinamento**.

Nel seguito, data un'ampiezza indicata dal numero naturale $n \geq 0$, identificheremo gli elementi di una permutazione generica con a_1, a_2, \dots, a_n usando pedici i normalmente inizianti da uno $1 \leq i \leq n$: salvo esplicita indicazione contraria (es. $i \in [0, n-1]$), che sarà invece prevalente nella terza sezione, laddove nel descrivere algoritmi e implementazioni preferiremo seguire la convenzione zero-based in uso in informatica.

2.2 Numero totale di permutazioni di n elementi.

La formula che esprime il numero di permutazioni in funzione dell'ampiezza, data dall'intero non negativo n , è ben nota e fa riferimento al fattoriale:

$$P(n) = n! \stackrel{\text{def}}{=} \begin{cases} 1 & \text{per } n \in \{0, 1\} \\ \prod_{j=1}^n j & \text{per } n \in \mathbb{N} \setminus \{0, 1\} \end{cases} \quad (1)$$

2.3 Permutazioni: notazione a due linee.

Uno dei metodi standard per la descrizione di una permutazione richiama strettamente la sua natura - appena ricordata - di criterio di ordinamento: si tratta della cosiddetta notazione a due linee.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 1 & 7 & 6 & 4 & 5 & 8 & 3 \end{pmatrix} \quad (2)$$

Tale notazione si legge intuitivamente per colonne, ad esempio da sinistra a destra, e dal basso verso l'alto: $2 \rightarrow 1, 1 \rightarrow 2, 7 \rightarrow 3, 6 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 6, 8 \rightarrow 7, 3 \rightarrow 8$ e l'ovvio significato intuitivo è che, rispetto all'ordine «naturale» della *permutazione identità* riportata alla prima riga, il 2 «prende il posto» del numero 1, il 7 «prende il posto» del 3, e così via. Si sottolinea come tale notazione risulti estremamente espressiva e flessibile.

2.4 Permutazioni: notazione ciclica.

Dalla notazione a due linee appena esposta risulta immediato definire anche il concetto di cicli disgiunti. Rior diniamo le assegnazioni dell'esempio (2) come segue: 1 → 2, 2 → 1; 3 → 8, 8 → 7, 7 → 3; 4 → 5, 5 → 6, 6 → 4. Si nota facilmente come le assegnazioni tornano ciclicamente ad un valore di partenza, qui sottolineato per evidenziarlo. Tali sequenze di scambi rappresentano i **cicli**, di varia lunghezza, che possono evidentemente essere formalizzati in vari modi del tutto equivalenti. Stanley [Sta86] ha per primo proposto una *notazione standard* nella quale ogni ciclo è scritto in modo tale da avere in prima posizione l'elemento massimo, e i cicli sono

¹Se effettivamente, com'è molto probabile, vi sono nel presente testo uno o più errori, allora l'asserzione è banalmente vera. Ma rimane vera anche nel caso opposto: in tale circostanza, infatti, l'errore è introdotto proprio dalla presenza di tale proposizione inherente la presenza di errori.

²Anche semplicemente "permutazioni", senza ulteriore specificazione, quando non sussistono ambiguità contestuali rispetto ad altre varianti come "permutazioni con elementi ripetuti" o "permutazioni di multinsiemi".

ordinati in modo crescente in base a tale valore. Applicando tale convenzione, la permutazione (2) si esprime come segue: (21)(654)(837).

La notazione standard di Stanley ha anche il vantaggio di poter omettere le parentesi, senza creare ambiguità: consideriamo l'esempio appena visto 21654837 e leggiamolo da sinistra a destra alla ricerca dei massimi relativi, cioè dei valori a_i tali che $a_i > a_j$ per ogni $j < i$, che per comodità abbiamo sottolineato. Aggiungiamo subito la coppia di parentesi che delimita ai suoi estremi la notazione ciclica: (21654837). Inseriamo ora una *coppia invertita* di parentesi tonde)(prima di ciascuno di codesti massimi relativi, tranne davanti al primo di essi, ove è chiaramente già presente la parentesi iniziale: (21)(654)(837). Ecco così facilmente ricostruita la scrittura originale³.

Esistono poi ulteriori convenzioni in letteratura, data la chiara non univocità della notazione: ad esempio, è molto diffusa l'abitudine di omettere eventuali cicli di lunghezza unitaria, ossia i punti fissi, che non cambiano posizione rispetto alla permutazione identità.

Infine, può essere utile almeno accennare al fatto che le permutazioni possono facilmente essere categorizzate in ordine di numero di cicli, il quale intuitivamente può variare tra n (permutazione identità, composta unicamente di punti fissi per i quali $a_i = i$, per ogni $i \in [1, n]$) e 1 per le cosiddette permutazioni *cicliche*, ossia interamente composte da un singolo ciclo.

Il numeri di Stirling di prima specie senza segno $\begin{bmatrix} n \\ k \end{bmatrix} = s(n, k)$ esprimono il numero di permutazioni con esattamente k cicli, dove appunto $1 \leq k \leq n$ (si veda in particolare il §6.1 in «Concrete Mathematics» [GKP94]).

2.5 Permutazioni: notazione matriciale.

Un ulteriore metodo utilizzato per la rappresentazione di permutazioni consiste nell'usare una matrice quadrata $n \times n$ contenente unicamente valori dall'insieme $\{0, 1\}$, in modo tale che per ciascuna riga e per ciascuna colonna vi sia un unico valore unitario. Consideriamo ancora la permutazione (2). La sua rappresentazione matriciale, convenendo di assegnare le *righe* alla permutazione identità, si scriverà come segue:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (3)$$

Si presti attenzione al fatto che (naturalmente!) esistono almeno due convenzioni discordanti sulla lettura di tale matrice sparsa, secondo che si interpretino le righe o le colonne come permutazione identità: e (altrettanto naturalmente!) esse sono egualmente diffuse in letteratura.

La convenzione qui usata è tale che, leggendo per righe, si ha $2 \rightarrow 1, 1 \rightarrow 2, 7 \rightarrow 3, 6 \rightarrow 4, \dots$ poiché si ha, rispettivamente, $M_{1,2} = 1, M_{2,1} = 1, M_{3,7} = 1, M_{4,6} = 1, \dots$ e più in generale, $C \rightarrow R$ se e solo se $M_{R,C} = 1$.

Alternativamente, in analogia con i diagrammi di partizione di Ferrers, si trovano in letteratura anche rappresentazioni come la seguente, che evidenziano unicamente le posizioni delle unità nella matrice sparsa, sostituendole con vari simboli.

	■						
■							
						■	
				■			
					■		
							■
		■					

Tale rappresentazione simbolico-grafica rende ancora più evidente la similitudine con una scacchiera popolata da sole torri, in posizione mutuamente neutrale: un problema tipico in combinatorica, strettamente correlato

³Allo stesso modo, altre notazioni analoghe offrono la medesima proprietà, eventualmente invertendo il verso della scansione e/o quello della relazione d'ordine tra valori (ricerca di massimo o minimo relativo). Ad esempio, Knuth ed altri autori seguono una convenzione diversa, scrivendo i cicli in modo da anteporre l'elemento minimo, e poi ordinando i cicli in modo crescente in base a tale valore. Nel caso esemplificato, avremo quindi: (12)(378)(465).

anche con il tipo di applicazione che andiamo a discutere nel seguito. Vale la pena di rimarcare come questo genere di rappresentazione rivesta maggiore interesse generale dal punto di vista matematico che da quello computazionale, se si eccettuano ovviamente il contesto didattico e una nicchia applicativa ben specifica.

2.6 Composizione di permutazioni.

Se si applica una permutazione τ ad un insieme già ordinato secondo una permutazione π , si ottiene semplicemente una nuova permutazione dell'insieme di partenza, normalmente distinta sia da π che da τ . Un semplice esempio chiarirà informalmente il concetto, ricorrendo di nuovo alla potente intuitività della notazione a due linee. Consideriamo le permutazioni:

$$\tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 5 & 3 & 1 & 9 & 2 & 8 & 7 & 4 \end{pmatrix}, \quad \pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \end{pmatrix}$$

Riscriviamo τ riordinando le colonne in modo tale che la prima riga diventi identica alla seconda riga di π , per facilitare la lettura:

$$\begin{aligned} \pi &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \end{pmatrix} \\ \tau &= \begin{pmatrix} 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \\ 8 & 1 & 6 & 7 & 9 & 5 & 4 & 2 & 3 \end{pmatrix} \\ \tau \circ \pi &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 1 & 6 & 7 & 9 & 5 & 4 & 2 & 3 \end{pmatrix} \end{aligned}$$

Applicando l'operazione di composizione si ottiene, in sostanza: $(8 \rightarrow 7) \circ (7 \rightarrow 1) = 8 \rightarrow 1$ e così via per ogni altro simbolo della permutazione composita $\tau \circ \pi$.

2.7 Permutazioni: notazione a linea singola (vettore di permutazione).

Un ulteriore modo abbreviato per rappresentare le permutazioni è la notazione lineare o *1-line*, nella quale si omette la prima riga (sottintendendo così la permutazione identità). Nel caso dell'esempio (2), si può quindi scrivere in vari modi la medesima permutazione:

$$\begin{aligned} &\langle 2, 1, 7, 6, 4, 5, 8, 3 \rangle \\ &2, 1, 7, 6, 4, 5, 8, 3 \\ &\quad 2 \ 1 \ 7 \ 6 \ 4 \ 5 \ 8 \ 3 \\ &\quad 21764583 \end{aligned}$$

Ricordiamo che la presenza di delimitatori ai margini della permutazione, come pure dei separatori tra gli elementi (es. virgole, spazi...) è del tutto opzionale, purché le omissioni consentano ovviamente di distinguere senza ambiguità i singoli elementi. Vale la pena di notare che, eliminando le parentesi ed eventuali separatori tra le cifre, anche la notazione ciclica (v. paragrafo 2.4) scritta in modo completo e senza omettere i punti fissi coincide con una permutazione in notazione lineare, della medesima ampiezza n ma quasi sempre distinta da quella di partenza (trovare i casi notevoli è un simpatico ed istruttivo esercizio). Ad esempio, sono permutazioni valide quelle ottenute scrivendo come specificato le due notazioni cicliche ricavate dalla (2) al §2.4: 21654837 e 12378465.

2.8 Numero di inversioni in una permutazione.

Una statistica di fondamentale importanza per le permutazioni è il numero di inversioni. Sappiamo che con n valori distinti è possibile formare un numero di coppie pari a:

$$C(n, 2) = \binom{n}{2} = \frac{n^2 - n}{2} \tag{4}$$

Con l'ovvia eccezione della permutazione identità, è evidente che in una qualsiasi altra permutazione alcune di queste coppie saranno sempre "fuori ordine", nel senso che esisterà almeno una coppia di valori (a_i, a_j) tali che $a_i < a_j$ ma $i > j$, con i e j opportuni indici.

Un semplice esempio sarà senza dubbio chiarificatore: consideriamo le $3! = 6$ permutazioni di ampiezza $n = 3$. Le coppie che è possibile formare con l'insieme dei valori $\{1, 2, 3\}$ sono solamente tre: $(1, 2)$, $(1, 3)$ e $(2, 3)$. Ordiniamo le nostre permutazioni lessicograficamente e per numero di inversioni crescente, elencando a lato di ciascuna di esse le coppie che risultano invertite:

Inversioni	Permutazioni e coppie invertite	Totale	
0	$\langle 1, 2, 3 \rangle$	1	
1	$\langle 1, 3, 2 \rangle$ $(2, 3)$	$\langle 2, 1, 3 \rangle$ $(1, 2)$	2
2	$\langle 2, 3, 1 \rangle$ $(1, 2); (1, 3)$	$\langle 3, 1, 2 \rangle$ $(1, 3); (2, 3)$	2
3	$\langle 3, 2, 1 \rangle$ $(1, 2); (1, 3); (2, 3)$	1	

2.9 Indici di inversione di una permutazione.

I valori riportati nella colonna “Totale” della tabella al paragrafo precedente esprimono il numero totale di permutazioni di ampiezza n nelle quali esattamente k coppie sono invertite. Essi prendono il nome di indici di inversione $I_n(k)$ e seguono una ricorrenza ben nota:

$$\begin{cases} I_0(0) = 1 \\ I_n(k) = \sum_{j=0}^n [I_{n-1}(k-j)] \quad \text{per } n \geq 1 \end{cases} \quad (5)$$

Come già implicitamente richiamato dalla (4), k assume i seguenti valori in funzione di n :

$$0 \leq k \leq \binom{n}{2} \quad (6)$$

Per i soli valori di k minori di n , è applicabile una ricorrenza semplificata:

$$I_n(k) = I_n(k-1) + I_{n-1}(k) \quad \text{per } k < n \quad (7)$$

Tale ricorrenza si esprime facilmente ([Bon12], pagg. 53-62) in forma chiusa, utilizzando i numeri pentagonali:

$$I_n(k) = \sum_{j=0}^M \left[(-1)^j \binom{n+k-d_j-1}{k-d_j} \right] \quad \text{per } n \geq k \quad (8)$$

Dove $d_j = \frac{3j^2-j}{2}$ come da definizione di numero pentagonale, mentre il limite superiore M della sommatoria è dato da:

$$M = \left\lceil \frac{-1 + \sqrt{24k+1}}{6} \right\rceil \quad (9)$$

Riportiamo di seguito le prime righe della tabella delle inversioni. Si presti attenzione al fatto che il valor massimo per k è definitivamente maggiore di n per ogni $n > 3$, il che di fatto limita l'utilità computazionale della forma chiusa (8).

n	k_{MAX}	$I_n(0)$	$I_n(1)$	$I_n(2)$	$I_n(3)$	$I_n(4)$	$I_n(5)$	$I_n(6)$	$I_n(7)$	$I_n(8)$	$I_n(9)$	$I_n(10)$	$I_n(11)$
1	0	1											
2	1	1	1										
3	3	1	2	2	1								
4	6	1	3	5	6	5	3	1					
5	10	1	4	9	15	20	22	20	15	9	4	1	
6	14	1	5	14	29	49	71	90	101	101	90	71	49

2.10 Vettori di inversione: le varianti più diffuse.

Associati strettamente agli indici di inversione appena visti, troviamo i cosiddetti vettori d'inversione. Esistono in letteratura numerose tipologie di tali vettori, ciascuna caratterizzata da metodi e algoritmi costruttivi diversi: appare doveroso rimarcare che solo alcuni di tali metodi risultano efficaci ed efficienti dal punto di vista applicativo, mentre altri sono semplicemente inaccettabili in un normale contesto computazionale, il che a sua volta vincola la scelta delle tipologie di vettore d'inversione più idonee per talune applicazioni.

Knuth ([Knu97], in particolare all'es. 5.1.1-7, pag. 592) classifica quattro tipologie fondamentali di vettori d'inversione, basate su diverse convezioni: ne esistono tuttavia anche altre, di minore diffusione e di più limitata utilità. Consideriamo il generico elemento a_i e due opportuni indici i e j (tali che $i \neq j$, $i \in [1, n]$, $j \in [1, n]$): fondamentalmente, per quantificare gli elementi «fuori posto» in una permutazione e sue derivate, si possono contare i valori inferiori ad a_i situati alla sua «destra», tali che $a_i > a_j$ ma $i < j$; oppure, per converso, è possibile contare i valori maggiori di a_i situati alla sua «sinistra», avendosi quindi $a_i < a_j$ e $i > j$.

Consideriamo ad esempio la permutazione seguente: 4 6 5 3 1 2. Rispetto al valore $a_4 = 3$ evidenziato, sono 3 i valori ad esso superiori posti alla sua «sinistra» ovvero con indici inferiori a 4; d'altro canto, vi sono anche 2 valori ad esso inferiori posti a «destra» ossia posti in locazioni con indici maggiori di 4.

Una volta scelto uno dei due metodi appena esposti, gli elementi nel vettore di inversione possono poi essere ordinati, altrettanto banalmente, in numerosi modi. I più diffusi (non certo unici) sono almeno due: *posizionale*, quindi con una corrispondenza diretta di indici, tale che alla locazione i -esima del vettore d'inversione troviamo il numero di inversioni (vedi sopra) corrispondente all'elemento i -esimo della permutazione in esame; o per contro, con una indicizzazione fissa, basata sul *valore* di a_i nella permutazione identità, in modo tale che la locazione uno (risp. zero) del vettore d'inversione contiene sempre e solo il numero di inversioni relative al valore uno (risp. zero), e così via per i successori.

Vediamo un facile esempio che illustra tali alternative. Sia data la permutazione:

$$5 \ 9 \ 1 \ 8 \ 2 \ 6 \ 4 \ 7 \ 3 \quad (10)$$

Si avranno quindi i seguenti possibili vettori di inversione (seguendo pedissequamente la nomenclatura dello Knuth):

$$\begin{aligned} b &= 2 \ 3 \ 6 \ 4 \ 0 \ 2 \ 2 \ 1 \ 0 \\ B &= 0 \ 0 \ 2 \ 1 \ 3 \ 2 \ 4 \ 2 \ 6 \\ c &= 0 \ 0 \ 0 \ 1 \ 4 \ 2 \ 1 \ 5 \ 7 \\ C &= 4 \ 7 \ 0 \ 5 \ 0 \ 2 \ 1 \ 1 \ 0 \end{aligned} \quad (11)$$

Consideriamo la locazione $a_1 = 5$ della permutazione (10). Ovviamente nessun valore superiore è presente a «sinistra», dunque si avranno 0 inversioni secondo tale convenzione. Si avrà dunque $b_5 = 0$ e corrispondentemente $B_1 = 0$, poiché i due vettori sono rispettivamente indicizzati secondo il *valore* ($a_1 = 5$) e la *posizione* (indice 1, corrispondente alla posizione dell'elemento considerato nella permutazione). Per contro, tutti e quattro i valori minori di 5 si trovano alla sua «destra»: dunque, nei vettori c e C ove si considerano appunto i valori «minori a destra», si avrà $c_5 = 4$ e $C_1 = 4$, in base ai medesimi criteri di indirizzamento. Lo stesso si applica ordinatamente ad ogni altra cifra della permutazione (10).

Si noti come la convenzione seguita per la costruzione del vettore ha conseguenze immediate sulla struttura del vettore stesso, e in particolare sulla collocazione degli zeri fissi: è infatti intuitivo come non possono in alcun caso esservi a «destra» del valor minimo valori inferiori ad esso, né tantomeno a «sinistra» del valor massimo, il che si applica rispettivamente anche al valore in ultima e prima posizione.

La tipologia b è spesso indicata in letteratura come codifica Lehmer⁴ [Leh60] o *factoradic*⁵, essendo intrinsecamente legata anche ad una peculiare rappresentazione multibase delle permutazioni⁶. La tipologia C è invece normalmente associata al nome di Marshall Hall [Hal56], al quale si devono l'osservazione che ogni permutazione è unicamente ricostruibile dal vettore d'inversione associato, ed il relativo algoritmo di conversione⁷.

2.11 Proprietà fondamentale dei vettori d'inversione.

Il numero totale dei vettori di inversione, per ogni data tipologia, è sempre pari a $n!$ e in corrispondenza biunivoca con le rispettive permutazioni, secondo l'osservazione di Hall [Hal56]. Vale certamente la pena di continuare con l'esempio (11) per evidenziare la seguente proprietà notevole, invariante per le quattro tipologie di vettore:

$$\begin{aligned} 2 + 3 + 6 + 4 + 0 + 2 + 2 + 1 + 0 &= 20 \\ 0 + 0 + 2 + 1 + 3 + 2 + 4 + 2 + 6 &= 20 \\ 0 + 0 + 0 + 1 + 4 + 2 + 1 + 5 + 7 &= 20 \\ 4 + 7 + 0 + 5 + 0 + 2 + 1 + 1 + 0 &= 20 \end{aligned} \quad (12)$$

⁴Il professor Derrick N. Lehmer è molto noto nella comunità informatica grazie anche al test di Lucas-Lehmer per i primi di Mersenne, alla base del progetto di ricerca distribuita GIMPS. In maniera non casuale, il test porta anche il nome di Édouard Lucas (che effettivamente lo ideò nel 1870), l'autore del problema dei *ménages* al centro del presente articolo.

⁵In realtà, lo stesso Knuth nel già citato 5.1.1-7 rileva come il tipo di vettore denominato b fosse già citato nel «Lehrbuch der Combinatorik» (1901), mentre il tipo C «di Marshall Hall» era già noto a Rodriguez (*J. de Math.*, 4) nel 1839.

⁶Rappresentazione di interesse matematico, ma non particolarmente utile dal punto di vista computazionale, poiché i relativi algoritmi di conversione hanno complessità *worst case* quadratica praticamente in ogni variante di normale diffusione, con rarissime eccezioni. Buona parte dei metodi si riconduce infatti alla scansione dei valori unitari in matrici come la (3) illustrata al §2.5, una operazione ovviamente quadratica se non si fa uso esplicito di istruzioni native dedicate eventualmente presenti sulla piattaforma, come *find first one/set* (ffo, ffs), *count leading zeroes* (clz) o *number of leading zeroes* (nlz), il che a sua volta implica l'uso di una rappresentazione binaria di ciascuna riga matriciale, con tutti i relativi costi di codifica e decodifica; altri metodi, purtroppo di gran lunga più comuni, fanno ricorso ad ancor più farraginose convoluzioni con moltiplicazioni per valori fattoriali al fine di valorizzare un numero multibase *factoradic*.

⁷Anche in questo caso, si tratta purtroppo di un algoritmo di scarsa utilità computazionale: una implementazione realmente efficiente in tempo lineare richiederebbe una struttura dati **ordinata** capace di supportare inserimenti in posizioni arbitrarie in $O(1)$.

Ovvero: la somma delle cifre di un vettore di inversione, definito secondo le regole sopra esposte, è uguale al numero totale di inversioni presenti nella permutazione alla quale il vettore stesso si riferisce.

Risulta facile osservare che, in ultima analisi, i vettori di inversione fin qui visti sono a tutti gli effetti *composizioni deboli*, ossia somme a valor costante di interi, ordinate, che ammettono anche un numero limitato di fattori nulli. Tale fatto viene sovente sfruttato nelle dimostrazioni, come per l'esempio in [Bon12] visto al paragrafo 2.9.

2.12 Derangement o “dismutazioni”.

Chiamiamo *derangement* una permutazione nella quale nessun elemento rimane al proprio posto di partenza, ovvero una permutazione priva di punti fissi. Ad esempio, con esplicito riferimento alla permutazione base (permutazione identità) $a_1a_2a_3a_4$, si ha che $a_4a_1a_2a_3$ è un derangement, mentre $a_3a_2a_1a_4$ non lo è, perché (almeno) un elemento a_i occupa la i -esima posizione (in questo banale esempio, $i = 4$, come evidenziato), ove opportunamente $i \in [1, n]$.

2.13 Numero totale dei derangements di n elementi.

Il numero totale dei derangements per una data ampiezza n si indica generalmente con $D(n)$, ed è definito un apposito operatore «cofattoriale» per indicare tale quantità:

$$D(n) = !n \stackrel{\text{def}}{=} n! \sum_{k=0}^n [(k!)^{-1}(-1)^k] \quad (13)$$

Risulta inoltre nota un'altra fondamentale relazione inerente $D(n)$:

$$D(n) = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor \quad \text{per } n \geq 1 \quad (14)$$

Come immediata conseguenza, abbiamo che il rapporto tra $D(n)$ e il numero totale di permutazioni $P(n)$ della (1) può scriversi:

$$\frac{D(n)}{P(n)} \approx e^{-1} \quad (15)$$

Con buona approssimazione, per valori di n sufficientemente elevati, una permutazione ogni tre è quindi un derangement.

2.14 Proprietà fondamentali dei derangements: cicli e inversioni.

Al paragrafo 2.4 abbiamo ricordato che i punti fissi di una permutazione corrispondono a cicli di lunghezza unitaria nella relativa notazione ciclica. Dalla definizione di derangement come permutazione priva di punti fissi data al §2.12 discende immediatamente la conseguenza: *i derangement espressi in notazione ciclica non presentano cicli di lunghezza unitaria*.

La restrizione della definizione ha conseguenze anche sul numero totale di inversioni, e quindi sui limiti per k indicati dalla (6). Per avere un derangement, intuitivamente, occorre come minimo scambiare un numero di coppie pari alla metà dell'ampiezza, approssimata per eccesso. Si ha quindi:

$$k \geq \left\lceil \frac{n}{2} \right\rceil \quad (16)$$

D'altro canto, la singola permutazione col massimo numero di scambi è la «più disordinata», quella nella quale gli elementi sono in ordine inverso rispetto alla permutazione identità: $a_n a_{n-1} \dots a_2 a_1$ e quindi tutte le $\binom{n}{2}$ coppie risultano scambiate. Tuttavia, tale permutazione non è un derangement se n è dispari, perché l'elemento mediano non cambia posizione nell'inversione. Per i derangements vale quindi, in definitiva:

$$\left\lceil \frac{n}{2} \right\rceil \leq k \leq \begin{cases} \binom{n}{2} & \text{per } n \text{ pari} \\ \binom{n-1}{2} & \text{per } n \text{ dispari} \end{cases} \quad (17)$$

3 Le problème des ménages.

3.1 Brevi cenni storici sulla figura di Édouard Lucas.

Il matematico francese François Édouard Anatole Lucas (Amiens, 1842 - Parigi, 1891) studiò presso l'*École Normale* di Amiens e in seguito lavorò presso l'Osservatorio di Parigi. Dopo aver prestato servizio come ufficiale di artiglieria durante la guerra franco-prussiana del 1870-71, ebbe incarichi come docente presso il Lycée Saint Louis e successivamente il Lycée Charlemagne, ambedue a Parigi. Fu attivo soprattutto in teoria dei numeri, come mostrano i suoi ampi studi sulla sequenza di Fibonacci e su quella strettamente correlata che porta il suo nome:

$$L_n = \begin{cases} 2 & \text{per } n = 0 \\ 1 & \text{per } n = 1 \\ L_{n-1} + L_{n-2} & \text{per } n > 1 \end{cases} \quad (18)$$

Tale sequenza è catalogata nell'enciclopedia online OEIS con il codice A000032. Eccone i primi valori:

A000032: 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1.364, 2.207, 3.571, 5.778, 9.349, 15.127, 24.476, 39.603, 64.079, 103.682, 167.761, 271.443, 439.204, 710.647, 1.149.851, 1.860.498, 3.010.349, 4.870.847, 7.881.196, 12.752.043, 20.633.239, 33.385.282, ...



Non sorprende che per la serie numerica di Lucas esista un elegantissimo analogo in forma chiusa della nota formula di Binet per i numeri di Fibonacci:

$$L_n = \left(\frac{1 + \sqrt{5}}{2} \right)^n + \left(\frac{1 - \sqrt{5}}{2} \right)^n \quad (19)$$

Altro punto focale degli interessi di Lucas furono i primi di Mersenne, come già ricordato in nota 4 a pag. 6: il che lo portò ad ideare il test di primalità attualmente noto come Lucas-Lehmer ed a confermare, nel 1876, quello che ad oggi rimane il più grande numero primo di Mersenne mai calcolato senza l'ausilio di un computer: $2^{127} - 1$.

Édouard Lucas è generalmente citato anche per le curiose circostanze della sua prematura scomparsa, che in qualche misura lo accomunano nell'aneddotica allo sfortunato destino dell'astronomo Tycho Brahe. Laddove infatti Brahe morì alcuni giorni dopo lo scoppio della vescica avvenuto durante un sontuoso banchetto, fu proprio durante il banchetto annuale di un congresso scientifico che Lucas venne accidentalmente ferito al volto dalla scheggia vagante prodotta dalla rottura di un piatto sporco, collocato assieme ad altre stoviglie su un vassoio inopinatamente caduto ad un cameriere. Il povero Lucas morì pochi giorni dopo, a soli 49 anni, per l'infezione acuta causata dalla ferita.

Lucas è certamente ben noto nella comunità informatica anche per i suoi interessi ludomateematici, che includono l'ideazione del problema delle Torri di Hanoi⁸, punto di passaggio pressoché obbligatorio da decenni nella didattica informatica: il suo ponderoso lavoro in quattro volumi «*Récréations mathématiques*» (1882-94) rimane senza dubbio un classico, con ristampe ed ampie citazioni riportate in numerosissimi lavori e raccolte contemporanee.

3.2 Le problème des ménages: storia e soluzioni.

Nel suo ulteriore lavoro del 1891 «*Théorie des nombres*» [Luc91], un tassello fondamentale nello sviluppo moderno della teoria del numero, Édouard Lucas formulò anche il problema dei *ménages*, il quale in definitiva consiste nel contare i modi complessivi per disporre a sedere attorno ad un tavolo circolare n coppie di invitati, con n numero naturale maggiore di due, rispettando due semplici regole:

1. Uomini e donne occupano posti rigorosamente alternati attorno alla tavola;
2. Nessun coniuge può occupare alcuna delle due sedie immediatamente adiacenti a quella proprio partner: così, se la signora J siede al posto $i - esimo$, il signor J non dovrà occupare i posti $(i - 1) \bmod n$ e $(i + 1) \bmod n$, e viceversa.

Lucas non fornì una soluzione completa, ma solo una serie di tabelle precompilate e alcuni riferimenti a risultati parziali di suoi collaboratori, in particolare quelli di un tale colonnello C. Moreau derivati da una impostazione di C. A. Laisant, di non semplice applicazione.

⁸Lo pseudonimo *N. Claus de Siam* non è altro che l'anagramma di Lucas d'Amiens.

Un problema molto simile si era in realtà già presentato attorno al 1870 al fisico P. G. Tait nell'ambito della teoria dei nodi, come peraltro ricordato nell'accurata ricostruzione storica presentata in un gradevole articolo di J. Dutka del 1986 [Dut86]: vi avevano lavorato, tra gli altri, nomi famosi come Arthur Cayley e Thomas Muir. A quest'ultimo si deve la riformulazione del problema, nel caso particolare di $n = 4$, come il numero di termini non nulli nell'espansione del determinante seguente:

$$\begin{vmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{vmatrix}$$

Ciò equivale al calcolo del permanente della matrice data - la quale, con ogni evidenza, altro non è che la versione 4×4 della matrice dei vincoli per i *ménages*, che vedremo meglio al paragrafo 3.3.

Numerose sono le connessioni del problema con la teoria dei nodi e con la teoria dei *rook polynomials*, come già accennato in particolare al paragrafo 2.5. In estrema sintesi, se consideriamo la matrice data «sovraposta» ad una scacchiera, in modo che gli zeri marchino locazioni proibite, le soluzioni del problema coincidono con il numero di configurazioni che è possibile ottenere piazzando n torri in posizione mutuamente neutrale nelle sole caselle consentite. Sia detto $u(n)$ tale numero di configurazioni: Muir, dopo alcuni scambi con Cayley, arrivò a definire la ricorrenza semplificata $(n-2)u(n) = n(n-2)u(n-1) + n \cdot u(n-2) + 4 \cdot (-1)^{n-1}$ con le condizioni iniziali $u(3) = 1$ e $u(4) = 2$.

Lucas, tuttavia, non fece alcuna menzione di codesti lavori precedenti; d'altro canto, nel 1902 anche H. M. Taylor ripresentò il problema dei *ménages*, a sua volta senza menzionare alcuna fonte, fornendo una serie di ricorrenze equivalenti a quelle di Lucas⁹, che furono poi riprese in numerose pubblicazioni ludomateematiche nei decenni successivi. In particolare, si deve a Taylor la ricorrenza seguente:

$$n \cdot u(n+2) = (n^2 + n + 1)(u(n) + u(n+1)) + (n+1)u(n-1) \quad (20)$$

certamente non difficile da derivare da quella di Muir sopra riportata.

Un altro nome ben noto che compare nella storia del problema è quello del maggiore Percy A. MacMahon, padre dell'omonimo fondamentale *Master Theorem* in combinatorica enumerativa e algebra lineare, che contribuì con un tentativo di soluzione simile al metodo Laisant, purtroppo parimenti difficile da applicare in pratica.

La prima soluzione esplicita, completa e in forma chiusa al problema di Lucas venne fornita da Jacques Touchard molti anni più tardi, nel 1934 [Tou34]. Tuttavia, egli a sua volta non fornì una dimostrazione di correttezza di tale formula.

$$Me(n) = 2 \cdot Dc(n) \cdot n! \quad (21)$$

Il termine $Dc(n)$ esprime il numero di derangements che rispettano la condizione fondamentale del problema, e tale formulazione è nota anche come problema dei *ménages* ridotto.

$$Dc(n) = \sum_{j=0}^n \left[(-1)^j \frac{2n}{2n-j} \binom{2n-j}{j} (n-j)! \right] \quad (22)$$

La prima dimostrazione della formula di Touchard fu pubblicata solo nove anni più tardi, nel 1943, a cura di Irving Kaplansky [Kap43].

Circa quaranta anni dopo tale articolo di Kaplansky, nel 1981, il problema venne poi curiosamente ripreso (più correttamente, riscoperto) da D. S. Jones e P. G. Moore [JM81], che fornirono la seguente soluzione in forma chiusa:

$$Dc(n) = \sum_{j=0}^n \frac{(2n-j-1)! n (-1)^j \pi^{\frac{1}{2}}}{j! \alpha! 2^{2\alpha}} \quad (23)$$

Dove $\alpha = (n-j-\frac{1}{2})$. Con alcuni passaggi (usando in particolare la formula di Legendre per la duplicazione della funzione Γ) ci si riconduce comunque facilmente alla formula di Touchard (22). Ai medesimi autori Jones e Moore si deve l'esplicita dimostrazione del rapporto asintotico tra il numero di soluzioni al problema dei *ménages* e il totale delle permutazioni $n!$:

$$\lim_{n \rightarrow \infty} \frac{Dc(n)}{P(n)} = e^{-2} \quad (24)$$

Analogamente a quanto già visto con la formula (15), si ha infatti:

$$\lim_{n \rightarrow \infty} \frac{Dc(n)}{D(n)} = \lim_{n \rightarrow \infty} \frac{D(n)}{P(n)} = e^{-1} \quad (25)$$

⁹Vale la pena di notare che, nell'evoluzione storica delle idee risolutive, quella dovuta a Taylor è la prima (e per molti decenni rimarrà l'unica) nella quale non si assegnino cavallerescamente subito i posti alle signore nello sviluppo del calcolo.

Per tale motivo, come già notato, molte fonti indicano che grossolanamente una permutazione ogni tre è un derangement, e a sua volta un derangement ogni tre è una soluzione al problema dato.

Nel 1983 una breve nota di G. E. Thomas [Tho83] sul medesimo giornale attuariale che aveva pubblicato l'articolo di Jones e Moore (il *Journal of the Institute of Actuaries* di Cambridge) rilevava come il problema da essi sollevato fosse in realtà l'ottocentesco *problème des ménages* di Lucas, già risolto da Touchard.

Solo tre anni più tardi, nel 1986, K. Bogart e P. Doyle [BD86] proposero una dimostrazione elementare della formula di Touchard, poi riportata in numerosissimi testi di matematica discreta e combinatorica, ironizzando garbatamente fin dal titolo della pubblicazione («*Non-sexist Solution of the Ménage Problem*») sulla evitabile complessificazione dei calcoli introdotta storicamente nelle varie procedure risolutive dall'implicita assunzione dell'idea cavalleresca di preassegnare i posti alle signore.

Nel 2005 una giovane studentessa, Amanda Passmore, ha infine pubblicato online (An elementary solution to the Ménage problem) una rivisitazione della elegante dimostrazione di Kaplanski, riusandone i lemmi fondamentali ma impiegando unicamente concetti elementari di conteggio nella loro giustificazione.

Le sequenze dei numeri di Touchard e dei relativi derangements condizionati, derivanti dalle formule (21) e (22), sono ovviamente riportate nell'enciclopedia OEIS, rispettivamente con i codici A059375 e A000179. Eccone i primi valori:

A059375: 1, 0, 0, 12, 96, 3.120, 115.200, 5.836.320, 382.072.320, 31.488.549.120, 3.191.834.419.200, ...

A000179: 1, 0, 0, 1, 2, 13, 80, 579, 4.738, 43.387, 439.792, 4.890.741, 59.216.642, 775.596.313, 10.927.434.464, 164.806.435.783, 2.649.391.469.058, 45.226.435.601.207, ...

I valori della sequenza A059375 sono evidentemente dell'ordine di grandezza $O(n! \cdot n!)$, a meno di un fattore e^{-2} come evidenziato dalla (24).

3.3 Connotazione dei derangements ristretti.

La connotazione dei derangements ristretti che risolvono il problema dato è piuttosto intuitiva. Una volta assegnato il posto i -esimo ad uno dei coniugi di una coppia, l'altro può sedersi ovunque, tranne nelle locazioni $(i - 1) \bmod n$ e $(i + 1) \bmod n$. Tale vincolo è espresso efficacemente dalla seguente matrice binaria ciclica, esemplificata per $n = 8$, con la convenzione che ogni indice di riga identifica una coppia, le colonne individuano le sedie, e il valore nullo marca i posti «proibiti».

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \quad (26)$$

Naturalmente, nel rispetto del vincolo del problema, la matrice indica unicamente i posti pari (risp. dispari). Ne consegue che risolvere il problema significa innanzi tutto trovare tutti e soli quei derangements che rimangono tali anche dopo una rotazione a destra di una posizione.

In altri termini, passando alla **convenzione computazionale** sugli indici che include lo zero, dato l'usuale insieme finito dei primi n naturali $\{0, 1, \dots, n-1\}$, consideriamo tutte e sole le sue permutazioni nelle quali nessun elemento a_i occupi la posizione $(i - 1) \bmod n$ né la posizione i , per ogni $i \in [0, n - 1]$. La sommatoria nella formula (22) esprime appunto, come già ricordato, il numero di tali derangements condizionati. Consideriamo ad esempio i $D(4) = 9$ derangements di ampiezza pari a 4:

1 0 3 2, 1 2 3 0, 1 3 0 2, 2 0 3 1, 2 3 0 1, 2 3 1 0, 3 0 1 2, 3 2 0 1, 3 2 1 0

Solamente due di essi rispettano il vincolo appena esposto:

$$\begin{array}{rcc} 3 & 0 & 1 & 2 & \leftrightarrow & 2 & 3 & 0 & 1 \\ 2 & 3 & 0 & 1 & \leftrightarrow & 1 & 2 & 3 & 0 \end{array} \quad (27)$$

3.4 Generazione di una singola soluzione valida.

Si delinea la semplice procedura per la generazione di una singola soluzione al problema dei *ménages*, esemplificando il caso di $n = 4$.

Per la massima chiarezza, presumeremo di partire da liste ordinate distinte delle signore (*Ladies*) $L[\] = \langle A, B, C, D \rangle$ e dei rispettivi coniugi (*Gentlemen*) $G[\] = \langle a, b, c, d \rangle$. Useremo una notazione vettoriale esplicita,

con gli operatori [] e indici inizianti da zero, tale che ad esempio $L[0] = A$ etc., ed $S[]$ sarà l'array di dimensione pari a $2n$ che conterrà la soluzione.

Per comporre “manualmente” la nostra soluzione, avremo bisogno di scegliere una tra le $P(4) = 4! = 24$ permutazioni di ampiezza 4:

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3, & 0 & 1 & 3 & 2, & 0 & 2 & 1 & 3, & 0 & 2 & 3 & 1, & 0 & 3 & 1 & 2, & 0 & 3 & 2 & 1 \\ 1 & 0 & 2 & 3, & 1 & 0 & 3 & 2, & 1 & 2 & 0 & 3, & 1 & 2 & 3 & 0, & 1 & 3 & 0 & 2, & 1 & 3 & 2 & 0 \\ 2 & 0 & 1 & 3, & 2 & 0 & 3 & 1, & 2 & 1 & 0 & 3, & 2 & 1 & 3 & 0, & 2 & 3 & 0 & 1, & 2 & 3 & 1 & 0 \\ 3 & 0 & 1 & 2, & 3 & 0 & 2 & 1, & 3 & 1 & 0 & 2, & 3 & 1 & 2 & 0, & 3 & 2 & 0 & 1, & 3 & 2 & 1 & 0 \end{array}$$

Ci occorre inoltre uno dei due derangements di ampiezza 4 che rispettano i vincoli del problema, già elencati alla (27):

$$3 \ 0 \ 1 \ 2, \quad 2 \ 3 \ 0 \ 1$$

La procedura risolutiva, che chiameremo *MENAGE1*, è facilmente costruita come segue:

1. Si sceglie una qualsiasi permutazione e la si applica alle due liste, riordinandole: es. $1 \ 0 \ 2 \ 3 \Rightarrow L[] = \langle B, A, C, D \rangle, G[] = \langle b, a, c, d \rangle$
2. Si sceglie se assegnare alle signore i posti pari o dispari: es. **pari**.
3. In base alla decisione 2, si assegnano ordinatamente i posti **pari** alle signore nel vettore soluzione, lasciando vuoti gli altri: $S[] = \langle B, \square, A, \square, C, \square, D, \square \rangle$.
4. Si sceglie uno dei derangements della (27) e lo si applica alla lista dei signori, già riordinata al punto 1: es. $3 \ 0 \ 1 \ 2 \Rightarrow G[] = \langle d, b, a, c \rangle$.
5. Si intercalano ordinatamente i signori nei posti rimasti disponibili nel vettore soluzione: $S[] = \langle B, d, A, b, C, a, D, c \rangle$.

Risulta immediato verificare che la soluzione rispetta i vincoli del problema. La composizione (v. §2.6) del derangement scelto al punto 4 con la permutazione applicata ai nomi degli ospiti al punto 1 garantisce sempre il rispetto dei vincoli.

Naturalmente anche noi stiamo cavallerescamente facendo accomodare prima le dame, in questo caso. Tuttavia, osservando l'algoritmo risolutivo e le formule (21) e (22), appare chiaro come gli unici fattori che effettivamente distinguono una soluzione dalle altre sono:

- La scelta del derangement ristretto tra quelli validi, il che dà luogo a $Dc(n)$ variazioni (22);
- L'ordine generale imposto all'elenco degli invitati, ossia la permutazione scelta tra le $n!$ disponibili;
- La scelta se applicare il derangement all'elenco dei signori o a quello delle signore, connessa alla scelta a priori di una convenzione per i posti pari e dispari, che concorre a raddoppiare il totale delle soluzioni possibili.

Ne consegue, come da formula di Touchard (21), il totale di $2 \cdot n! \cdot Dc(n)$ soluzioni distinte e quindi la correttezza per costruzione della procedura.

3.5 Generazione esaustiva delle soluzioni al problema dei *ménages*.

Dall'algoritmo *MENAGE1* derivano in modo facile ed immediato i passi fondamentali di una procedura per la generazione *esaustiva* delle soluzioni, nella quale si esplicita in modo diretto il banale meccanismo di indicizzazione degli array degli invitati usando le permutazioni e i derangements di volta in volta generati:

Procedura *MENAGE2*: Passi fondamentali nella generazione esaustiva delle soluzioni.

1. Si generano tutti i $Dc(n)$ derangements di ampiezza n ;
- (a) **Per ogni** derangement d si generano tutte le $n!$ permutazioni $P(n)$;
 - i. **Per ogni** permutazione $\pi \in P(n)$, si generano le due soluzioni simmetriche, assegnando dapprima - per fissare le idee, con i opportuno indice - i posti pari a $L[\pi[i]]$ e i dispari a $G[\pi[d[i]]]$, e poi invertendo le signore e i signori.

Tale procedura garantisce di visitare una e una sola volta tutte le soluzioni al problema: anche se non si fornisce qui una dimostrazione formale di correttezza, la dimostrazione combinatoria è decisamente intuitiva. Si noti

che i passi **1** e **(a)** possono essere scambiati tra di loro in modo del tutto arbitrario, senza alcuna conseguenza sulle funzionalità della procedura.

Segue un esempio di output completo per $n = 3$: $Dc(3) = 1$, $P(3) = 3! = 6$, $Me(3) = 2 \cdot 6 \cdot 1 = 12$ soluzioni distinte. La lista degli invitati prevede le tre coppie $\langle Andre, Bernard, Calais \rangle$ - ovviamente in onore del luogo e del tempo in cui fu descritto il problema originale da Lucas. *Mesdames, messieurs: à la table!*

```
>> Problema dei menages:  
>> esempio didattico di procedura risolutiva esaustiva per 3 coppie.  
>> Si hanno in totale 1 derangements e 12 soluzioni.  
>>  
> Derangement 1 su 1: (201)  
> Permutazione 1 su 6: (abc)  
> Soluzione 1 (priorita' alle Signore):  
>> 1 M.me Andre      2 Msr. Calais  
>> 3 M.me Bernard    4 Msr. Andre  
>> 5 M.me Calais     6 Msr. Bernard  
> Soluzione 2 (priorita' ai Signori):  
>> 1 Msr. Andre      2 M.me Calais  
>> 3 Msr. Bernard    4 M.me Andre  
>> 5 Msr. Calais     6 M.me Bernard  
  
> Permutazione 2 su 6: (bac)  
> Soluzione 3 (priorita' alle Signore):  
>> 1 M.me Bernard    2 Msr. Calais  
>> 3 M.me Andre      4 Msr. Bernard  
>> 5 M.me Calais     6 Msr. Andre  
> Soluzione 4 (priorita' ai Signori):  
>> 1 Msr. Bernard    2 M.me Calais  
>> 3 Msr. Andre      4 M.me Bernard  
>> 5 Msr. Calais     6 M.me Andre  
  
> Permutazione 3 su 6: (cab)  
> Soluzione 5 (priorita' alle Signore):  
>> 1 M.me Calais     2 Msr. Bernard  
>> 3 M.me Andre      4 Msr. Calais  
>> 5 M.me Bernard    6 Msr. Andre  
> Soluzione 6 (priorita' ai Signori):  
>> 1 Msr. Calais     2 M.me Bernard  
>> 3 Msr. Andre      4 M.me Calais  
>> 5 Msr. Bernard    6 M.me Andre  
  
> Permutazione 4 su 6: (acb)  
> Soluzione 7 (priorita' alle Signore):  
>> 1 M.me Andre      2 Msr. Bernard  
>> 3 M.me Calais     4 Msr. Andre  
>> 5 M.me Bernard    6 Msr. Calais  
> Soluzione 8 (priorita' ai Signori):  
>> 1 Msr. Andre      2 M.me Bernard  
>> 3 Msr. Calais     4 M.me Andre  
>> 5 Msr. Bernard    6 M.me Calais  
  
> Permutazione 5 su 6: (bca)  
> Soluzione 9 (priorita' alle Signore):  
>> 1 M.me Bernard    2 Msr. Andre  
>> 3 M.me Calais     4 Msr. Bernard  
>> 5 M.me Andre      6 Msr. Calais  
> Soluzione 10 (priorita' ai Signori):  
>> 1 Msr. Bernard    2 M.me Andre  
>> 3 Msr. Calais     4 M.me Bernard  
>> 5 Msr. Andre      6 M.me Calais
```

```
> Permutazione 6 su 6: (cba)
> Soluzione 11 (priorita' alle Signore):
>> 1 M.me Calais      2 Msr. Andre
>> 3 M.me Bernard     4 Msr. Calais
>> 5 M.me Andre       6 Msr. Bernard
> Soluzione 12 (priorita' ai Signori):
>> 1 Msr. Calais      2 M.me Andre
>> 3 Msr. Bernard     4 M.me Calais
>> 5 Msr. Andre       6 M.me Bernard
>> Generati in totale 1 derangements e 12 soluzioni.
```

4 Algoritmi e implementazioni per il problema dei *ménages*.

La letteratura in materia di permutazioni e derangement è realmente sterminata, tanto che oggi sarebbe impossibile rendere giustizia all'argomento senza dedicare un intero tomo monografico ad una bibliografia esaustiva¹⁰. Si passa dal venerabile Knuth ([Knu97, Knu05]) che all'argomento dedica un intero capitolo nel terzo volume, per tornarvi poi nel secondo fascicolo del quarto volume, al recentissimo lavoro di Arndt (Matters Computational) che dedica ben due capitoli alla medesima questione, fino alle monografie interamente dedicate alle permutazioni e dintorni, come quella di Miklos Bona [Bon12]: per non dire delle centinaia di articoli. Non c'è testo di combinatorica computazionale, tra quelli qui citati e dozzine di altri, che non dedichi ampio spazio alla questione di generare permutazioni e loro restrizioni! Knuth osservava, con la sua usuale arguzia, che il numero di algoritmi per mettere in disordine una lista è almeno del medesimo ordine di grandezza di quello degli algoritmi di ordinamento.

4.1 Algoritmi generatori di permutazioni e affini.

In generale, codesti algoritmi sono facilmente classificabili in poche grandi famiglie:

- **Generatori esaustivi:** sovente ricorsivi, con alcune punte d'eccellenza iterative, producono una lista di oggetti combinatori in un ordine predefinito¹¹. Tra gli ordinamenti più diffusi troviamo lessicografico, colessicografico, a minima variazione, *plain change* o *bell ringing*, per numero di inversioni, per prefissi, eccetera. Tali generatori, specialmente quelli «universali» ma anche moltissimi dedicati, possono a loro volta essere «filtrati» con *early* o *late filtering*¹², per generare oggetti con particolari restrizioni.
- **Metodi di successione:** in grado di generare il «prossimo» oggetto combinatorio a partire da quello correntemente dato, secondo un ordinamento prestabilito. Di norma operano la trasformazione in tempo $O(n \cdot \log_2 n)$ o superiore, con ben poche eccezioni.
- **Ranking e unranking:** ampia famiglia di algoritmi che convertono direttamente un numero intero (l'ordinale, compreso tra zero e $P(n)$ o $D(n)$ ¹³) nella corrispondente permutazione, o viceversa, sempre secondo un ordinamento prestabilito. Le prestazioni variano ampiamente: si hanno esempi che eseguono la singola conversione in tempo ammortizzato (CAT)¹⁴ o $O(n \cdot \log_2 n)$, ma non mancano numerosi algoritmi con prestazioni quadratiche. Sebbene possano essere usati per una generazione esaustiva, le loro prestazioni medie inducono più spesso ad impiegarli per generare insiemi ristretti di oggetti combinatori. Un altro possibile impiego di tali algoritmi è la generazione di una singola permutazione o altro oggetto combinatorio analogo, dopo averne scelto l'ordinale corrispondente in modo pseudorandom: tuttavia, su questo terreno si trovano a competere direttamente (spesso in ampio svantaggio a priori) con la famiglia algoritmica descritta al punto immediatamente seguente.

¹⁰Quasi superfluo rimarcare qui che le note bibliografiche del presente articolo non hanno assolutamente la benché minima pretesa in tale senso, ma solo la funzione pratica di rimandare in modo immediato ad alcuni algoritmi e *survey* citati più o meno direttamente nel corso dell'esposizione.

¹¹In genere offrono prestazioni comprese tra $O(n^2 \cdot n!)$ e $O(n!)$: in quest'ultimo caso ogni singolo oggetto è generato in $O(1)$, come negli algoritmi *loopless*. Sovente taluni speciali tipi di ordinamento implicano una maggiore complessità computazionale, influendo negativamente sulle prestazioni. Sono altresì noti alcuni particolari algoritmi i quali, producendo in output codifiche binarie con ordinamenti appropriati, sono in grado di fornire prestazioni pari a $O((n - 1)!)$, es. [HRW12].

¹²Si parla di *early filtering* o *early rejection* (in combinatorica computazionale talora è usato anche l'acronimo BEST per *Backtracking Ensuring Success at Terminals*, es. [ER03]) quando la verifica dell'oggetto generato è anticipata il più possibile rispetto al termine della generazione, bloccando così quanto prima i rami improduttivi dell'albero computazionale (per questo un'altra espressione equivalente spesso usata è *early pruning*). Tipicamente, ad esempio, un algoritmo con *early filtering* per la generazione di derangements si arresterà, passando al successivo step, non appena si verificherà la condizione $a_i = i$ per l'indice corrente i ; per contro, un generatore di derangements con *late filtering* attenderà invece di avere generato l'intera permutazione prima di avviare il controllo, per ogni cifra a_i , che la relativa posizione sia diversa da i , per poi accettare o scartare l'oggetto generato. Nel mondo dell'Intelligenza Artificiale, tale paradigma è meglio noto come *generate and test*. Per quanto tendenzialmente svantaggioso, in alcuni casi il ricorso al *late filtering* è ingegneristicamente giustificato da altri vantaggi: semplificazione del codice, sostanziale limitatezza del numero di oggetti da generare, o altri fattori secondari, che in taluni frangenti possono bilanciare il generale maggior costo computazionale. In determinati casi, peraltro, si tratta dell'unica opzione ragionevole, a causa della peculiare natura dei dati o del problema che non consentono valutazioni anticipate sulla soluzione generata fino al completamento dell'elaborazione della stessa.

¹³Più in generale, l'ordinale ha ovviamente come limite superiore l'arietà o cardinalità della classe di oggetti combinatori considerata.

¹⁴Il concetto di CAT, tempo costante ammortizzato, è strettamente legato al mondo della generazione combinatoria. La sua introduzione è dovuta a Frank Ruskey in [Rus] (Combinatorial Generation), pag. 8: «...the amount of computation, after a small amount of preprocessing, is proportional to the number of objects that are listed. We do not count the time to actually output or process the objects; we are only concerned with the amount of data structure change that occurs as the objects are being generated.» Per definizione, si parla quindi di complessità CAT quando la computazione, a meno di un trascurabile *preprocessing* approssimativamente invariante, rimane direttamente proporzionale alla quantità di oggetti generati e quindi sostanzialmente *lineare*. L'idea è ripresa ed espansa nella IV parte del [CLRS09], al capitolo sull'analisi ammortizzata.

- **Metodi pseudorandom:** metodi diretti (es. Knuth's *shuffle*¹⁵, Sattolo [Sat86], Myrvold-Ruskey [MR01]), con prestazioni ampiamente accettabili per ottenere un singolo oggetto combinatorio, o al più per poche iterazioni. Occupano molto validamente la loro nicchia applicativa, risultando in genere altamente specializzati e verticalizzati.
- **Algoritmi paralleli e per GPGPU:** ormai da oltre quattro lustri la vera frontiera della ricerca nel campo della generazione combinatoria. Si tratta quasi sempre di algoritmi con basi concettuali innovative rispetto ai tradizionali lavori partoriti fino ad alcune decine di anni fa, e sovente richiedono un poderoso *framework* di supporto e hardware dedicato.

4.2 Algoritmi utilizzati nel software dimostrativo.

Come illustrato in *MENAGE2* (v. §3.5), la soluzione richiede sostanzialmente un generatore di permutazioni e uno di derangements. Potenzialmente si tratta anzi del medesimo algoritmo, eventualmente condizionato applicando diversi filtri per la generazione degli oggetti combinatori di volta in volta desiderati.

Dal punto di vista della strategia di computazione, siamo davanti ad un problema di generazione di un sottoinsieme di oggetti combinatori con restrizioni. Date le relazioni di cardinalità tra gli insiemi considerati (permutazioni, derangements e derangements ristretti) $P(n) > D(n) > Dc(n)$ quantificate come valori limite dalle formule (15), (25) e (24):

$$Dc(n) \approx \frac{D(n)}{e} \approx \frac{P(n)}{e^2} \quad (28)$$

e considerando che una singola soluzione viene convalidata in $O(n)$, si può scartare a priori per motivi prestazionali il ricorso ad algoritmi esaustivi per permutazioni con *late filtering*. Altre soluzioni previste in letteratura applicativa possono invece essere considerate per una valutazione in parallelo. In particolare:

- Algoritmi esaustivi per la generazione di permutazioni, adattati con *early filtering*;
- Algoritmi esaustivi per derangements, con filtraggio (*late* o *early*)¹⁶;
- Algoritmi esaustivi generici (*rule-based*), che consentono di generare una vasta gamma di oggetti combinatori con prestazioni mediamente buone.

Dato lo scopo ludico e illustrativo del presente articolo, si sono voluti presentare quattro diversi algoritmi, elaborati in epoche diverse e con scopi distinti, appartenenti alle categorie citate. Nella completa implementazione in linguaggio C disponibile per il download sono esemplificati algoritmi scelti secondo criteri computazionali e di engineering asseverati: non ultimi semplicità e leggibilità, oltre ad un lungo ed eccellente stato di servizio nella storia applicativa. In ordine alfabetico, gli algoritmi usati sono dovuti a:

Akl, S. G. (1980), §4.2.2: algoritmo ricorsivo esaustivo per derangements, al quale applichiamo un *late filtering* per convalidare solo i derangements ristretti che risolvono il problema dato.

Heap, B. R. (1963), §4.2.1: algoritmo esaustivo per permutazioni, iterativo. Usato per la sola fase (a) di *MENAGE2*.

Knuth, D. E. (2005), §4.2.4: algoritmo generico, iterativo, *rule based*.

Rohl, J. S. (1978), §4.2.3: algoritmo ricorsivo generico con *early pruning* dell'albero computazionale.

¹⁵ Breve ma doverosa nota filologica. L'algoritmo di shuffle “di Knuth” è stato pubblicato per la prima volta nella sua forma attuale, com’è ben noto, da R. Durstenfeld ([Dur64]). L’attribuzione “shuffle di Fisher e Yates” invece non è universalmente accettata senza discussioni, come molte fonti online paiono erroneamente ritenere (o, peggio, vorrebbero far credere). L’originale algoritmo pubblicato nel 1938 dai due statistici Sir Ronald A. Fisher (1890 – 1962) e Frank Yates (1902 – 1994), semplicemente, **non era computabile** e non si basava in modo specifico sull’aumento di efficienza dell’operazione di cancellazione da un array (realizzata in $O(1)$ tramite un semplice scambio con l’ultimo elemento e la riduzione di una unità della dimensione dell’array, per le mosse successive). Poiché esattamente in questa idea consiste l’asse portante dell’algoritmo e la chiave del suo successo applicativo, molti rifiutano la prassi di “cointestare” l’algoritmo stesso e considerano unicamente la pubblicazione di Durstenfeld come l’atto di nascita di tale procedura di shuffle in senso strettamente informatico: senza per questo voler togliere a Fisher e Yates una doverosa menzione (magari in una nota o commento) e senza sminuire in alcun modo i loro meriti. Meriti che peraltro sono e rimangono numerosi, anche in campo discretistico: a partire dai loro contributi allo studio dei *block designs*, dei quali rimane traccia ancor oggi nel fatto che la cardinalità dell’insieme degli elementi in un BD è curiosamente indicata con la lettera v , a memoria dell’originale “varieties of wheat” di cui si occupavano i due ricercatori britannici; per non dire della notissima disuguaglianza di Fisher per i *block designs* BIBD, intuitiva quanto fondamentale: in ogni $(v, b, r, k, \lambda) - BIBD$, $b \geq v$ ovvero il numero dei blocchi è non minore del numero di elementi dell’insieme di scelta.

¹⁶ Nel caso del *late filtering*, l’accettazione e convalida di una soluzione ogni tre offre prestazioni generalmente accettabili, in prima approssimazione.

Segue una sommaria descrizione di ciascuno, accompagnata da pseudocodice e stralci di implementazione in linguaggio C, nei quali si è dato risalto unicamente ai contenuti rilevanti, omettendo i dettagli minori¹⁷. Nella descrizione semiformale degli algoritmi si è cercato di privilegiare le convenzioni applicative più diffuse, usando in particolare la notazione vettoriale esplicita $A[i]$ in luogo di a_i e mantenendo la corrispondenza tra *posizione* e *valore* nella permutazione identità, il che implica l'uso dello zero come valore iniziale di indicizzazione e come valor minimo di ciascuna permutazione. Si ha quindi come permutazione identità: $P_{ID}[] = d_0d_1d_2 \dots d_{n-1} = 012 \dots n - 1$. Tuttavia, in almeno un caso ciò non risulta possibile, in quanto il valore zero è esplicitamente usato come *guard value*. L'unica conseguenza in casi del genere è comunque l'uso occasionale di un offset unitario per gli indici e/o i valori dell'array, ben riconoscibile e sostanzialmente innocuo dal punto di vista prestazionale e della leggibilità.

4.2.1 Algoritmo HEAP_PERM di B. R. Heap (1963).

L'algoritmo di Heap¹⁸ [Hea63] viene rapidamente discussso per primo in quanto, unico tra i quattro menzionati, è usato semplicemente per generare le permutazioni dell'elenco degli invitati, come passo (a) di *MENAGE2* (v. §3.5).

La versione utilizzata è quella iterativa, resa popolare da Robert Sedgewick nel suo fondamentale survey [Sed77] a fine anni Settanta, al quale si rimanda per ogni approfondimento. Il «trucco» del metodo Heap e di quelli analoghi dei suoi predecessori Wells e Boothroyd, come illustrato dal Sedgewick, sta nella semplificazione della tabella posizionale richiesta dal metodo odometrico¹⁹, di cui gli algoritmi appena menzionati costituiscono un affinamento.

Restando all'efficacissima ed intuitiva simbologia del Sedgewick, Heap usa per la funzione $B[N, c]$ la *parità*, e quindi il bit meno significativo (LSB, *Least Significant Bit*), dell'indice corrente per scegliere la posizione da permutare, eliminando così del tutto il ricorso a tabelle precalcolate (LUT, *Look-Up Tables*).

In questo specifico caso, il vettore $P[]$ di ampiezza n contiene la permutazione, e deve essere inizializzato in modo da contenere la permutazione identità, così che $P[i] = i$ per $0 \leq i < n$. L'algoritmo fa inoltre uso dell'array $w[]$, anch'esso di ampiezza n , inizialmente contenente solo valori nulli, usato per i “pesi” e introdotto durante il processo di eliminazione della ricorsione.

```

procedure HEAP_PERM
    output( $P[0], P[1], \dots, P[n - 1]$ )
     $i \leftarrow 0$ 
    while  $i < n$  do
        if  $w[i] < i$  then
            if (i è pari) then  $k \leftarrow 0$  else  $k \leftarrow w[i]$ 
             $P[i] \leftrightarrow P[k]; w[i] \leftarrow w[i] + 1; i = 0$ 
            output( $P[0], P[1], \dots, P[n - 1]$ )
        else
             $w[i] = 0; i \leftarrow i + 1$ 
    return

```

La semplicità strutturale è decisamente notevole e caratterizza in modo rimarchevole questo algoritmo. Davvero non occorrono approfondite descrizioni per illustrare il funzionamento di questo semplicissimo, intuitivo algoritmo odometrico basato su scambi.

Tenendo sempre ben presente che parliamo di complessità dell'ordine di $O(n!)$, con un coefficiente dominante²⁰ pari a circa $(e - 1) \cong 1,72$, l'algoritmo appena presentato con le sue numerose versioni micro-ottimizzate è da

¹⁷Ricordiamo che i codici sorgenti completi sono disponibili separatamente per il download, come indicato nel prologo al §1. I brani di sorgenti qui riportati sono stralci, modificati e semplificati a puro titolo illustrativo. Invitiamo il lettore a consultare sempre i codici sorgenti completi, anche durante la lettura del presente articolo, per un riferimento più accurato e aggiornato.

¹⁸Il cognome non ha la benché minima attinenza con la denominazione dello “heap” in quanto struttura dati astratta basata su alberi.

¹⁹Vale forse la pena di ricordare ai più giovani che tale metodo deve il suo nome agli odometri meccanici, con i quali il lettore ha già probabilmente una certa familiarità, anche se forse in modo poco consapevole: *odometro* infatti (dal greco ὁδός, strada e μέτρον, misura) è detta propriamente non solo la classica ruota metrica da agrimensori con indicazione meccanica della distanza percorsa, ma anche quella sezione fiscale del contachilometri automobilistico che non può essere azzerrata manualmente dal guidatore. Allo stesso modo, la maggior parte dei contatori tradizionali per gli allacciamenti urbani delle utenze alle reti di distribuzione pubbliche (es. acqua, gas...) erano basati su un contatore con analogo meccanismo, prima dell'avvento pervasivo dei moduli di misura digitali e della telemetria. Più in generale, gli algoritmi generatori “odometrici” sono ispirati al meccanismo dei più tradizionali contatori a ruote affiancate, congegnati in modo tale che ad una rotazione completa di una ruota corrisponde un avanzamento unitario della ruota immediatamente alla sua sinistra. Chiaramente con tale sistema è possibile rappresentare molto più che semplici conteggi decimali, predisponendo opportunamente le cifre su ciascuna ruota: il che, di norma, necessita appunto di tabelle precalcolate, se si parla di emulare informaticamente il semplice meccanismo.

²⁰In questo caso specifico si tratta di generare l'intero insieme delle $n!$ permutazioni, non solo i derangements o loro ulteriori restrizioni.

decenni uno dei migliori esistenti nello scenario applicativo per la generazione esaustiva di permutazioni. Verso la fine degli anni Settanta Robert Sedgewick ([Sed77]), proprio nel contesto della rigorosa analisi dalla quale è tratta la presente versione iterativa dell'algoritmo, considerava lo Heap il più efficace generatore esaustivo disponibile all'epoca: per la generalità dei casi, possiamo tranquillamente asserire che tuttora rimane tra i migliori, sebbene siano facilmente reperibili sul web variazioni e derivazioni anche più spinte prestazionalmente, ma che concettualmente non si discostano poi troppo rispetto alle idee originali di Heap e Sedgewick, salvo al più qualche euristica.

La relativa implementazione (in versione opportunamente semplificata) si configura come segue:

```

1 void HeapPerm(void)
2 {
3     unsigned int P[MAX_COUPLES];
4     unsigned int w[MAX_COUPLES];
5     size_t i;
6     for (i = 0; i < Len; ++i) {
7         w[i] = 0;
8         P[i] = i;
9         putchar(P[i] + '0');
10    }
11    puts("");
12    for (i = 0; i < Len;) {
13        if (w[i] < i) {
14            unsigned int tmp, k;
15            k = (i & 1) * w[i];
16            /* SWAP P[i], P[(i & 1) * w[i]] */
17            tmp = P[i]; P[i] = P[k]; P[k] = tmp;
18            for (k = 0; k < Len; ++k) {
19                putchar(P[k] + '0');
20            }
21            puts("");
22            w[i]++;
23            i = 0;
24        } else {
25            w[i++] = 0;
26        }
27    }
28 }
```

Il suggerimento di utilizzare il tipo *unsigned int* per il vettore di permutazione è legato alle migliori prestazioni generalmente offerte da tali tipi sulle architetture contemporanee e con i compilatori C mainstream, pur nella piena consapevolezza che la capienza richiesta non supera quella di un *char* - fatto implicitamente ribadito dall'uso illustrativo di *putchar()* per la stampa della singola cifra, perfettamente valido per piccoli ordini di derangement, che usano unicamente le cifre 0...9.

L'unica ulteriore nota implementativa riguarda l'espressione $k=(i\&1)*w[i]$ che consente elegantemente di eliminare una *if()* e quindi un salto隐式的, usando l'algebra booleana. Il valore dell'espressione è infatti pari a $w[i]$ se il bit meno significativo di i è pari ad uno (i dispari), e zero in caso contrario. Naturalmente sarebbe facile ottimizzare ulteriormente tale espressione eliminando la moltiplicazione²¹ tramite l'uso accorto di funzioni booleane: tuttavia, sui compilatori mainstream testati l'optimizer è sempre riuscito a produrre codice macchina privo di moltiplicazioni esplicite, senza necessità di ulteriori ottimizzazioni del sorgente, che rimane in tal modo probabilmente anche più leggibile.

4.2.2 Algoritmo DERANGE2 di S. G. Akl (1980).

Tale noto algoritmo è stato pubblicato su BIT **20** (1980), 2-7 [Akl80] ed ha conosciuto una notevole fortuna applicativa per tutti gli anni Ottanta e Novanta del secolo scorso, grazie alla sua inerente semplicità. Si tratta in origine di un algoritmo ricorsivo, e come tale viene qui presentato: dobbiamo quindi attenderci una tangibile

²¹Notoriamente l'istruzione IMUL ha ancor oggi prestazioni pessime sulle piattaforme Intel e AMD mainstream.

penalità prestazionale nella sua implementazione in linguaggio C, a causa delle convenzioni di chiamata di tale linguaggio e della totale mancanza di supporto alla *tail recursion*.

Come già visto per l'algoritmo di Heap, l'importanza storica, l'eleganza e la semplicità di questo algoritmo ne giustificano, da sole, la menzione e l'analisi, a prescindere da ogni ulteriore considerazione prestazionale e implementativa. La facilità estrema con la quale lo si è adattato a produrre i derangements ristretti per la risoluzione del problema, sia pure usando un *late filtering* certo non ottimale prestazionalmente, deve suggerire al lettore i motivi principali per cui un siffatto algoritmo ha dominato pressoché incontrastato la scena applicativa nell'ambito della generazione esaustiva di derangement e derivati fin dalla sua concezione all'inizio degli anni Ottanta e per almeno due decenni.

Il vettore $D[]$ di ampiezza n ospita il derangement, e inizialmente deve contenere la permutazione identità, in modo tale che $D[i] = i$ per $0 \leq i < n$.

```

procedure DERANGE2(m)
  if m ≠ 0 then
    if D[m] = m - 1 then l ← m - 2 else l ← m - 1
    for j = l to 0 do
      D[m] ↔ D[j]; DERANGE2(m - 1); D[m] ↔ D[j]
    else if D[0] ≠ 0 then
      validate(D[0], D[1], …, D[n - 1])
    return
  
```

Il funzionamento dell'algoritmo è decisamente intuitivo: perfino più semplice dello Heap visto al paragrafo precedente. Analizziamo la k -esima chiamata, con $0 < k < n$: quando viene chiamata `DERANGE2(k)`, gli elementi $D[k + 1] \dots D[n - 1]$ sono già al loro posto e costituiscono il suffisso del derangement corrente. Il derangement procede quindi a ritroso: per questo motivo, quando $D[k] = k - 1$, si varia semplicemente il valore iniziale della variabile di induzione j , in modo tale che il primo scambio (in realtà nullo, in quanto normalmente $j = k - 1$ alla prima iterazione) non produca una permutazione che viola il vincolo fondamentale del derangement.

Da notare il caratteristico procedimento con *backtracking*, che al termine del loop annulla lo scambio effettuato prima della chiamata ricorsiva, assolutamente tipico di una intera classe di algoritmi generativi esaustivi, come avremo modo di vedere nel seguito.

Con riferimento alla k -esima chiamata, è immediato calcolare che il numero totale di chiamate ricorsive per il livello successivo $k + 1$ è dato da:

$$d_{n,k} = \begin{cases} (n-1) \cdot d_{n-1,k} + (n-k-1) \cdot d_{n-2,k} & \text{per } 1 \leq k < n \\ 1 & \text{per } k = n \\ 0 & \text{per } k > n \end{cases} \quad (29)$$

Si tratta di una ricorrenza lineare omogenea di grado due a coefficienti variabili, tra le più semplici. Il totale delle chiamate ricorsive è dato quindi dalla sommatoria dei $d_{n,k}$ espressi dalla (29):

$$T_{Akl} = \sum_{k=1}^n [(n-1)d_{n-1,k} + (n-k-1)d_{n-2,k}] \quad (30)$$

Risolvendo la semplice ricorrenza in questione si ottiene il seguente risultato:

$$T_{Akl} = \sum_{k=1}^n \sum_{r=0}^{n-k} \frac{(-1)^r}{k!} \binom{n-k}{r} (n-r)! \approx (1 - e^{-1})n! \cong 0,63212 \cdot n! \quad (31)$$

La costante moltiplicativa è quindi decisamente superiore a quanto indicato dalla (24), come era lecito attendersi. La versione dell'algoritmo proposta è stata facilmente modificata per generare solamente i derangement condizionati che risolvono il problema dei *ménages*, come spiegato al §3.3. In questo caso, come accennato appena sopra, si è voluto mostrare un esempio di *late filtering* ossia del paradigma *generate-and-test*, nel quale il riconoscimento avviene solo dopo aver popolato l'intero array con il derangement corrente. Si noti come ciò, peraltro, non diminuisce il numero complessivo di chiamate ricorsive, risultando quindi irrilevante dal punto di vista del guadagno prestazionale teoricamente dato dal minor numero di oggetti computazionali da generare (circa 1/3 dei derangements totali).

```

1 void Akl_derange(const char len)
2 {
3     if (0 == len) {

```

```

4      /* Convalida della soluzione ed eventuale stampa. */
5      CheckMenage(D);
6  } else {
7      int j;
8      /*
9      ** In evidenza il calcolo del valore iniziale di j in funzione
10     ** del vincolo fondamentale per i derangements.
11     */
12    j = ((len -1) == D[len -1]) ? (len -2) : (len -1);
13    for ( ; j >= 0; --j) {
14        unsigned int t;
15        /* SWAP d[j], d[len-1] */
16        t = D[j]; D[j] = D[len -1]; D[len -1] = t;
17        Akl_derange(len - 1);
18        /* SWAP d[j], d[len-1] */
19        t = D[j]; D[j] = D[len -1]; D[len -1] = t;
20    }
21 }
22 }
```

Analizzando il sorgente esemplificativo si possono facilmente evidenziare alcuni capisaldi implementativi ed ingegneristici che caratterizzano tutti gli esempi qui proposti. In primo luogo, pur trattandosi di una prassi applicativa non incoraggiata (configge apertamente con i principi di buona modularizzazione basati sul criterio di *information hiding* di Parnas [Par72]), si è fatto il più ampio uso possibile di variabili *static* a livello di modulo, per evitare di aggravare le già pessime prestazioni in tempo e spazio di stack degli algoritmi ricorsivi presentati. L'unico parametro utilizzato è quello fondamentale per la ricorsione, che svolge il medesimo ruolo della variabile di induzione nella corrispondente implementazione iterativa. Inoltre, in questo caso specifico, in considerazione della dinamica della variabile indice j , si è tenuta la `if()` (per l'esattezza, un operatore ternario) rigorosamente fuori dal loop interno, evitando un marchiano errore implementativo - purtroppo molto più diffuso di quanto si possa ritenere. Anche qui, come nel caso dello Heap, si è scientemente rinunciato ad ottimizzare ulteriormente l'espressione²² appena discussa, per l'effettivamente scarsa rilevanza nel contesto prestazionale e, in subordine, per evitare il rischio di una potenziale *early optimization*, contoproducente con i compilatori mainstream più recenti.

4.2.3 Algoritmo di J. S. Rohl (1978).

Dopo un esempio di *late filtering*, non poteva mancare l'opportuna controparte: un altro algoritmo ricorsivo (generico), della medesima epoca ([Roh78]), riadattato per implementare una *early rejection*.

Similmente a quanto già visto con l'algoritmo di Heap (§4.2.1), l'algoritmo implementa un array ausiliario di pesi $We[]$, di dimensione n , inizialmente valorizzato con ogni locazione pari ad uno, in modo tale che $We[i] = 1$ per $0 \leq i < n$. Tale array, in questo caso, fa parte delle capacità estese dell'algoritmo, in grado di lavorare con multinsiemi. La versione qui presentata costituisce infatti un riadattamento semplificato dell'algoritmo originale: ne discuteremo nuovamente al paragrafo 5.3, parlando della sua più recente revisione polifunzionale. Uniformemente con gli altri esempi analoghi, anche qui l'array $D[]$ contiene il derangement e la variabile n ne rappresenta l'ampiezza: trattandosi nuovamente di routine ricorsiva, valgono appieno le considerazioni implementative sulle variabili condivise già viste al precedente paragrafo 4.2.2.

²²La differenza tra i due valori assegnati a j è pari ad una sola unità, dunque risulta facilmente ricavabile da un'espressione booleana di comparazione vincolata algebricamente a restituire null'altro che 1 o 0: il che, assieme ad una semplice somma, eliminerebbe del tutto la `if()`.

```

procedure ROHL(m)
    if m = n then
        output(D[0], D[1], ..., D[n - 1])
    else
        for j = 0 to n-1 do
            if We[j] ≠ j AND m ≠ j AND m ≠ (j + 1) mod n then
                D[m] ← j
                We[j] ← We[j] - 1
                ROHL(m + 1)
                We[j] ← We[j] + 1
    return

```

Per piccoli valori ($n < 12$), il numero totale di chiamate ricorsive effettuate in questo algoritmo semplificato è pari in media a:

$$T_{Rohl} \cong 0,45 \cdot n! \quad (32)$$

Si tratta di un netto miglioramento, almeno in teoria, rispetto al coefficiente dell'equazione (31) relativa all'algoritmo di Akl (§4.2.2), e ciò è dovuto quasi interamente all'adozione della *early rejection* che riduce tangibilmente il numero di chiamate non necessarie, come appena accennato. Tuttavia, sarà necessario prendere in considerazione ulteriori fattori per una più ampia valutazione complessiva, come vedremo meglio al paragrafo 4.3.

```

1 void Rohl_derange(const char len)
2 {
3     if (len == n) {
4         /* Stampa del derangement generato */
5     } else {
6         size_t i;
7         for (i = 0; i < n; ++i) {
8             if ((We[i] == 1)           &&
9                 (len != i)           &&
10                (((len + 1) % n) != i)) {
11                 D[len] = i;
12                 We[i]--;
13                 Rohl_derange(len + 1);
14                 We[i]++;
15             }
16         }
17     }
18 }
```

Si notano facilmente, alle righe 9 e 10, le condizioni per la generazione dei derangements ristretti (v. §3.3) nel loop principale dell'implementazione, che costituiscono il meccanismo di *early rejection* ed evitano di percorrere rami improduttivi dell'albero computazionale. Ritroveremo tali condizioni anche nell'algoritmo X di Knuth, al paragrafo successivo.

4.2.4 Algoritmo X di D. E. Knuth.

L'algoritmo in questione è basato su un'idea di M. C. Er pubblicata nel 1987, ed è apparso agli inizi del nuovo millennio, con la recente pubblicazione del quarto volume del TAoCP ([Knu05], pag. 334: «*Lexicographic permutations with restricted prefixes.*»). Si tratta di un algoritmo iterativo, in vari aspetti diverso dagli altri qui presentati²³.

L'algoritmo originale fa uso di due array²⁴ ausiliari: un array di link $L[]$, che contiene una lista circolare degli elementi ancora non utilizzati, consentendo l'accesso in $O(1)$, e un array di *undo* $U[]$ per implementare il

²³Non si può non fare cenno ai numerosi usi ludico-enigmistici di tale algoritmo: dalla risoluzione di *alphametics* alla generazione di soluzioni per Sudoku e quadrati latini, a ulteriore conferma della sua notevole flessibilità ed ingegnosità.

²⁴Si presti attenzione al fatto che l'algoritmo fa un uso peculiare degli indici ed usa lo zero come valore di arresto, quindi occorre gestire esplicitamente degli offset nell'implementazione al fine di usare per il derangement valori nel range $[0, n - 1]$ coerentemente con gli altri esempi. Per non compromettere eccessivamente la leggibilità, si è però rispettata la gestione delle posizioni originale, in modo tale che il vettore di output contiene la permutazione a partire dalla locazione 1, non già dalla zero. A tale scopo, l'implementazione fa uso di un ulteriore array ausiliario, $P[]$, usato durante la costruzione del derangement.

backtracking, già visto (in altra forma) nell'algoritmo di Akl (§4.2.2). L'array dei link (si veda anche [Knu00], il famoso articolo sui dancing links) è organizzato nel modo seguente. Se ad un determinato passo gli elementi ancora disponibili sono:

$$\{b[1], \dots, b[n-k]\} \quad \text{con } b[1] < \dots < b[n-k]$$

allora si avrà:

$$\begin{aligned} L[b[n-k]] &= 0 \\ L[b[j]] &= b[j+1] \quad \text{per } 0 \leq j < n-k \end{aligned}$$

L'algoritmo, come presentato da Knuth, genera tutte e sole quelle permutazioni che superano una sequenza di test $t_1(P[1]), t_2(P[1], P[2]), \dots$, i quali sono effettuati ad ogni iterazione su un numero crescente di elementi della permutazione stessa (*prefix acceptance test*), in ordine lessicografico. In questo caso, coerentemente con tutti gli altri esempi analoghi, la generazione è vincolata ai soli derangements ristretti, nell'array ausiliario $P[]$ di dimensione n .

Segue la descrizione dell'algoritmo, con le etichette originali impiegate da Knuth e riportate anche nei commenti dell'implementazione.

Algoritmo X di Knuth:

- X1.** [Initialize.] $L[n] \leftarrow 0$; $L[k] \leftarrow k + 1$ per $0 \leq k < n$; $k \leftarrow 1$.
- X2.** [Enter level k.] $p \leftarrow 0$, $q \leftarrow L[0]$.
- X3.** [Test $a_1 \dots a_k$.] $D[k] \leftarrow q - 1$. Se $t_k(D[1] \dots D[k])$ è falso, vai a **X5**. Altrimenti, se $k = n$, stampa la permutazione corrente e vai a **X6**.
- X4.** [Increase k.] $U[k] \leftarrow p$; $L[p] \leftarrow L[q]$; $k \leftarrow k + 1$. Torna a **X2**.
- X5.** [Increase a_k .] $p \leftarrow q$; $q \leftarrow L[p]$. Se $q \neq 0$ torna a **X3**.
- X6.** [Decrease k.] $k \leftarrow k - 1$. Se $k = 0$, termina. Altrimenti $p \leftarrow U[k]$; $q \leftarrow D[k] + 1$; $L[p] \leftarrow q$. Vai a **X5**.

L'implementazione segue pedissequamente l'algoritmo dato, facendo ampio uso delle `goto` laddove necessario, ciò che peraltro caratterizza numerosi tra i più efficienti algoritmi proposti da Knuth. Anche Joerg Arndt (Matters Computational) ha proposto una implementazione sostanzialmente identica, in C++, per la sua libreria FXT. Le caratterizzazione dell'algoritmo in termini di numero di esecuzioni in funzione di n è gentilmente fornita dallo stesso Knuth ([Knu05], 7.2.1.2-47, pag. 709). I passi $X1 \dots X6$ sono eseguiti rispettivamente:

$$(1, A, B, A - 1, B - N_N, A) \tag{33}$$

Dove si ha: $A = N_0 + \dots + N_{n-1}$, $B = nN_0 + (n-1)N_1 + \dots + N_{n-1}$, e N_k è, per ogni passo, il numero di prefissi di lunghezza k che superano i test.

```

1 void Knuth_X(void)
2 {
3     unsigned int P[MAX_COUPLES+1];           /* Array ausiliario per il derangement */
4     unsigned int Links[MAX_COUPLES+1];
5     unsigned int Undo[MAX_COUPLES+1];
6     size_t k;
7     unsigned int p, q;
8     /* X1. [Initialize.] */
9     for (k = 0; k < Width; ++k) {
10         Links[k] = k + 1;
11     }
12     Links[Width] = 0;
13     k = 1;
14     /* X2. [Enter level k.] */
15 X2:
16     p = 0;
17     q = Links[0];
18     /* X3. [test (P[1], ..., P[k])] */

```

```

19 X3:
20     P[k] = q -1;
21     if ((q == k) || (q == 1 + (k % Width))) {
22         goto X5;
23     } else if (k == Width) {
24         /*
25          ** Si ricordi la peculiarita' di questo
26          ** algoritmo: P[0] non e' usato.
27          */
28         memcpy(D, P+1, Width * sizeof(unsigned int));
29         PrintPerm(D);
30         goto X6;
31     }
32     /* X4. [Increase k.] */
33     Undo[k] = p;
34     Links[p] = Links[q];
35     ++k;
36     goto X2;
37     /* X5. [Increase P[k].] */
38 X5:
39     p = q;
40     q = Links[p];
41     if (q != 0)
42     {
43         goto X3;
44     }
45     /* X6. [Decrease k.] */
46 X6:
47     --k;
48     if (0 == k) return;
49     p = Undo[k];
50     q = P[k] +1;
51     Links[p] = q;
52     goto X5;
53 }
```

4.3 Confronto prestazionale dei tre algoritmi proposti.

L'idea di presentare ben tre distinti algoritmi per la risoluzione di un problema ludico è ovviamente illustrativa ed ha unicamente lo scopo di privilegiare la chiarezza per stimolare riflessioni e ulteriori ricerche nel lettore. Come d'uopo, abbiamo fornito una prima connotazione computazionale dei tre algoritmi con le equazioni (31), (32) e (33) (rispettivamente per Akl §4.2.2, Rohl modificato §4.2.3 e X di Knuth §4.2.4) in termini di numero di chiamate ricorsive generate e iterazioni.

La seguente tabella raffronta visivamente, per piccoli valori di n , il numero di chiamate ricorsive generate dai due algoritmi §4.2.2 e §4.2.3 e il numero di iterazioni della sezione **X3** (chiaramente preponderante) nell'algoritmo X di Knuth §4.2.4. Si tratta naturalmente di grandezze da usare solo per una prima stima asintotica del comportamento: una superficiale lettura della tabella potrebbe infatti suggerire conclusioni ingannevoli in ordine alle effettive prestazioni finali del codice su macchine mainstream odierne.

n	Rohl	Knuth	Akl
3	4	6	8
4	12	20	31
5	57	85	147
6	323	452	853
7	2.185	2.906	5.824
8	17.064	21.868	45.741
9	150.880	187.963	405.845
10	1.488.359	1.813.270	4.012.711
11	16.195.234	19.377.044	43.733.976
12	192.629.822	227.046.040	520.795.003

Nella nostra analisi dobbiamo partire da due pilastri fondamentali, sperabilmente ben chiari ad ogni lettore:

- In primo luogo, è chiaro a priori che l'*overhead* introdotto dalle chiamate ricorsive in C rende pressoché inutile il raffronto tra i due algoritmi ricorsivi e l'algoritmo X - sebbene si tratti di un algoritmo molto generico e quindi non particolarmente ottimizzato per la generazione di derangements ristretti, si può affermare a priori e senza tema di smentite che le prestazioni offerte a runtime dall'implementazione in linguaggio C dell'algoritmo iterativo X saranno nettamente superiori rispetto agli altri due.
- L'introduzione dell'*early reject* nell'algoritmo di Rohl ha sicuramente effetti tangibili sull'albero di esecuzione e quindi sul numero complessivo di chiamate generate, rendendolo in teoria assai superiore rispetto all'Akl modificato con *late filtering*.

Connotando intuitivamente i tempi di esecuzione con l'iniziale degli autori di ciascun algoritmo, sulla base delle considerazioni appena viste ci attenderemmo quindi una situazione del tipo $t_K \ll t_R < t_A$: tuttavia, non sempre tale ragionevole gerarchia rispecchia le effettive prestazioni del codice generato. Vale infatti la pena di rimarcare come, attivando tutte le opzioni di ottimizzazione più spinte su vari compilatori mainstream contemporanei o molto recenti, si ottiene al *profiling* sistematico una **inversione di prospettiva** rispetto al quadro teorico delineato dal numero complessivo di chiamate effettuate, in maniera apparentemente controiduitiva.

Ottimizzazioni	Akl	Knuth	Rohl
Standard/no	100%	37,66%	67,93%
Massime	90,96%	62,79%	100%

La tabella sintetizza e riassume le prestazioni medie rilevate per $3 \leq n \leq 20$ su cinque piattaforme hardware di test Intel e AMD mainstream, con tre diversi compilatori in ambiente Windows 7 e 8. In ciascuna riga la routine con le peggiori prestazioni funge da riferimento per i tempi di esecuzione relativi delle altre. La prima riga è, come descritto, relativa ad eseguibili prodotti con solo le opzioni standard di ottimizzazione, oppure nessuna ottimizzazione, secondo l'ambiente di sviluppo; la seconda riga si riferisce agli eseguibili prodotti dal medesimo sorgente, ma con il massimo livello di ottimizzazione abilitato.

Se da un lato è chiaramente e ampiamente confermata la prevista superiorità dell'algoritmo iterativo sugli altri, il resto appare assai meno lineare di quanto l'analisi asintotica consentisse di supporre. Come si vede, nei due scenari le prestazioni relative delle due implementazioni ricorsive variano drasticamente e solo nel primo caso rispecchiano le attese dell'analisi teorica basata sul numero di chiamate generate. Si noti inoltre l'appiattimento del divario tra routine ricorsive e iterazione quando sono attivate tutte le opzioni di ottimizzazione, compresa la convenzione di chiamata *register-based*.

In realtà, a prescindere dalla natura ricorsiva dell'implementazione (che può appunto essere mitigata, su architetture moderne, forzando l'uso di *calling conventions* basate sui registri, eliminando lo *stack frame* e attivando altre ottimizzazioni minori), nel caso dell'algoritmo di Rohl risulta un'idea controproducente attivare ottimizzazioni come *loop unrolling*, *coiling* e *branch prediction* a causa del salto implicito determinato dalla presenza strutturale della *if()* nel loop interno e della conseguente difficoltà di gestione da parte della *branch prediction unit*. Assume inoltre un peso rilevante la presenza di un operatore resto % nella condizione valutata ad ogni iterazione, e quindi il plausibile ricorso ad una istruzione IDIV da parte del compilatore sulle piattaforme Intel mainstream.

Pur senza passare direttamente all'Assembly inline o all'uso esplicito dei *builtins*, in taluni compilatori con un più fine controllo delle opzioni di ottimizzazione (in particolare Digital Mars) si possono in realtà ottenere situazioni intermedie di ottimizzazione che rispecchiamo ancora fedelmente le aspettative teoriche e in particolare la relazione d'ordine $t_K \ll t_R < t_A$ sopra delineata, a prezzo di un lieve decremento complessivo nelle prestazioni assolute rispetto a quanto ottenibile in regime di massima ottimizzazione.

In generale, tuttavia, algoritmi della natura del Rohl sono destinati a fornire sorprese non positive all'implementatore occasionale nel quadro di applicazioni contemporanee ad elevato *throughput* su hardware mainstream, mentre idee apparentemente molto ingenue come il *late filtering* applicato ad un classico algoritmo odometrico ricorsivo possono controiduitivamente risultare premiate dalla mera, acritica applicazione delle opzioni di ottimizzazione avanzate dei compilatori più recenti.

In definitiva, appare confermato ancora una volta il famoso adagio di Michael Abrash: «*The best optimizer is between your ears*». Trova altresì ennesima conferma il principio, mai ripetuto a sufficienza, di effettuare sistematicamente il *profiling* assieme all'analisi puntuale del codice assembly prodotto per gli *inner loops* e per tutte le sezioni di codice maggiormente critiche dal punto di vista prestazionale.

5 Bonus: altri tre algoritmi per i derangements.

In questa sezione, come gentile omaggio per i lettori, introduciamo brevemente altri tre algoritmi molto recenti (inclusa una evoluzione del Rohl §4.2.3) per la generazione di derangements, ciascuno caratterizzato da notevoli peculiarità. Viene fornita anche in questo caso una completa, originale implementazione in linguaggio C disponibile per il download.

Effler-Ruskey (2003), §5.1: algoritmo ricorsivo, in tempo CAT²⁵, pubblicato agli inizi del millennio ([ER03]), ha la particolarità di generare tutte le permutazioni con un dato numero di inversioni (v. §2.8). Facilmente adattato alla generazione dei soli derangements.

Mikawa-Tanaka (2014), §5.2: algoritmo iterativo recentissimo ([MT14]) che garantisce *ranking* e *unranking* (v. §4.1) in ordine lessicografico di derangements in tempo $O(n \cdot \log_2 n)$ usando alberi binari e le proprietà della notazione ciclica (v. §2.4).

Rohl-Ganapathi-Rama (2010), §5.3: evoluzione del già visto algoritmo generico ricorsivo di Rohl (v. §4.2.3), di recente pubblicazione ([GR10], arXiv:1009.4214v2), utilizzabile per numerose varietà di oggetti combinatori, è in grado tra l'altro di gestire estensivamente multinsiemi (quindi anche permutazioni con elementi ripetuti), ordinamenti totalmente arbitrari dell'insieme di output, e molto altro ancora.

5.1 Algoritmo di Effler-Ruskey.

L'algoritmo, pubblicato nel 2003 ([ER03]), è uno dei pochissimi dedicati alla generazione di permutazioni in base al numero di inversioni (v. §2.8). Si tratta di un algoritmo ricorsivo, in tempo CAT, fondamentalmente molto semplice, che tuttavia richiede alcuni accorgimenti non banali in fase implementativa.

Uniformemente al resto dello pseudocodice nel presente articolo, l'array $D[]$ di ampiezza n contiene il derangement, mentre k è il numero di inversioni desiderate, i cui limiti nel caso di derangement sono specificati dalla formula (17).

Gli autori definiscono $rank(x)$ come il numero di elementi che, ad ogni passo, precedono l'elemento x nella lista ordinata degli elementi ancora non utilizzati $[0, N - 1] \setminus \{D[n], \dots, D[N - 1]\}$.

```

procedure ER_derange(n, k)
    if n = 0 AND k = 0 then
        output(D[0], D[1], ..., D[n - 1])
    else
        for each x ∈ [0, N - 1] \ {D[n], ..., D[N - 1]} do
            if n - rank(x) ≤ k ≤ (n-1) + n - rank(x) then
                D[n] ← x
                ER_derange(n - 1, k - n + rank(x))
    return

```

Trattandosi di un algoritmo non esaustivo, limitato a particolari sottoinsiemi di derangements, si può già in prima approssimazione ritenere che le prestazioni complessive risultino ampiamente accettabili: specialmente in un contesto esemplificativo come il presente. Inoltre l'inerente semplicità strutturale, che ormai il lettore ha imparato ad associare agli algoritmi ricorsivi classici, rimane rimarchevole.

Tuttavia un simile algoritmo, se implementato ingenuamente in linguaggio C, potrebbe dare adito a codice scadente dal punto di vista qualitativo e prestazionale. Pur senza eliminare la ricorsione, vale la pena di suggerire alcune fondamentali strategie migliorative, lasciando la soluzione volutamente aperta ad ulteriori modifiche e migliorie, per stimolare ed invogliare il lettore alla sperimentazione.

La più importante variazione introdotta dall'implementazione d'esempio è l'eliminazione della `if()` dal loop principale: il controllo dei limiti per k è attuato indirettamente tramite la variabile di induzione j . In pratica viene invertito il concetto stesso di $rank(x)$: ben lungi dal ricorrere ad una qualche funzione esplicita di conteggio lineare o dall'usare una *Look-Up Table*, si usa invece una normalissima variabile di induzione per selezionare progressivamente gli elementi in lista, entro limiti di induzione assai facilmente calcolabili. Dal momento che il valore di k è in realtà fissato per ciascuna istanza ricorsiva, sostituendo $rank(x)$ con j nella coppia di disequazioni della `if()` sopra riportata e risolvendo in funzione di tale variabile, si ha infatti banalmente:

$$\begin{cases} j & \geq n - k \\ j & \leq \binom{n-1}{2} + n - k \end{cases}$$

²⁵Vedi nota 14 a pag. 14.

La presenza di una sottrazione nell'espressione del limite inferiore invita immediatamente alla prudenza, poiché parliamo (come quasi sempre) di valori di induzione strettamente non negativi. Infatti, come ben ricordiamo dalla (6), il numero di inversioni k assume sicuramente valori maggiori di n per ogni $n > 3$. Dunque è computazionalmente opportuno gestire la cosiddetta *saturazione a zero* del limite inferiore, per evitare problemi.

$$j \geq \begin{cases} n - k & \text{per } k \leq n \\ 0 & \text{per } k > n \end{cases}$$

Il codice tiene poi adeguatamente conto dell'off-by-one $n - k - 1$, poiché i nostri indici partono (come di consueto) da zero. Allo stesso modo, è facile verificare che l'espressione per il limite superiore può assumere valori maggiori di n e ciò non è desiderabile nel nostro loop principale. Anche in questo caso si gestisce opportunamente la relativa saturazione, isolandone il calcolo fuori dal loop (per evitare che qualche compilatore meno evoluto generi codice inutilmente soggetto a valutazioni multiple, ad ogni iterazione). Naturalmente, anche il coefficiente binomiale presente nella formula viene prelevato da una *LUT* precalcolata, per la massima efficienza.

Tutto ciò consente già un tangibile miglioramento prestazionale (ricordando che si parla pur sempre di una routine ricorsiva) rispetto ad una implementazione ingenua, e fornisce per l'occasione anche quel margine di manovra necessario all'introduzione del filtraggio che consente di limitare la generazione ai soli derangements. La lista degli elementi non ancora utilizzati è implementata con una *simply linked list (SLL)*, opportunamente iniziante con un *dummy node* che consente di eliminare in pratica ogni caso speciale nella gestione del primo nodo durante le normali operazioni sulla lista. La lista linkata semplice implementa, assieme ad uno stack ausiliario, la tecnica dei "dancing links" descritta da Knuth nel già citato articolo [Knu00] (arXiv:0011047) per le *doubly linked lists*, un'idea dovuta principalmente a Hitotumatu e Noshita a fine anni Settanta.

Nel loop principale, se il valore del "nodo corrente"²⁶ non coincide con l'indice corrente (*early filtering*), si procede con quella che sostanzialmente è la classica terna di istruzioni già vista negli algoritmi classici come Akl (v. §4.2.2), con qualche ovvia variazione contingente (indentata per evidenziare le operazioni principali):

- Si assegna il valore corrente alla locazione del derangement $D[n - 1]$.
 - La (macro)funzione `PushNode(*p)` salva il puntatore al "nodo corrente" in un apposito stack (implementato nel modo più efficiente tramite array e contatore).
 - Si elimina il "nodo corrente" dalla lista dei non utilizzati, semplicemente saltando il relativo link nella catena.
- Si invoca il successivo passo di ricorsione con nuovi opportuni valori per i parametri n e k .
- Al ritorno (*backtracking*) si reinserisce il nodo in lista, alla posizione originale, semplicemente ripristinando il puntatore con una chiamata a `PopNode()`. Si noti che il puntatore `next` del nodo ripristinato (`ptr->next->next` nel sorgente) non viene in alcun caso modificato, secondo l'idea portante dei *dancing links* sopra richiamata.

Come ultima ovvia operazione del loop principale, si passa al nodo successivo della lista.

I sorgenti delle funzioni accessorie qui non riportate, per brevità, sono ovviamente presenti nella versione integrale dei sorgenti disponibile per il download, che peraltro implementano esplicitamente *inline* le pseudofunzioni simboliche push/pop di nodo sopra richiamate. L'ottimizzazione prestazionale apportata dalla silloge di tecniche utilizzate è già notevolissima rispetto alla tipica implementazione scolastica, capace addirittura di ricorrere allegramente a devastanti (de)allocazioni dinamiche dei nodi ad ogni cancellazione e reinserimento, come purtroppo visto troppe volte nella prassi didattica e perfino su molti repositories. Il successivo passo di ottimizzazione consiste senz'altro nell'eliminazione della ricorsione o nel ricorso ad un linguaggio funzionale come Haskell, ML, OCaml e simili.

```

1 unsigned int ER_derange(const unsigned char n, const unsigned char k)
2 {
3     unsigned int cnt = 0;
4     size_t j;
5     Node_t *ptr = ListHead.first;
6     if ((0 == n) && (0 == k)) {
7         for (j = 0; j < Width; ++j) {

```

²⁶Trattandosi di *simply linked list*, per ovvi motivi pratici ciò che nel seguito descriviamo come "nodo corrente" in realtà è il successore del nodo effettivamente puntato da `ptr`, dal momento che parliamo di gestire sostanzialmente cancellazione e inserimento in lista, e quindi di lavorare sul puntatore "next" nel predecessore del nodo in questione.

```

8         putchar(D[j] +'0') ;
9     }
10    putchar( '\t' );
11    ++cnt ;
12 } else {
13     size_t lim , w = 0;
14     /*
15      ** Il calcolo dei limiti per j in funzione dei vincoli imposti a k
16      ** consente di eliminare la if() dal loop principale rispetto allo
17      ** pseudocodice proposto nell'articolo originale di Effler & Ruskey .
18      ** Si estrapola dalla for() il calcolo del limite massimo in funzione
19      ** di k ed n, per evitare potenziali valutazioni multiple nel binario
20      ** prodotto .
21     */
22     j = max(n -k -1, 0);
23     lim = min((BinCoef(n -1, 2) + n -k) , n);
24     /* Posizionamento iniziale nella LLS */
25     for (w = 0; w < j; ++w) {
26         ptr = ptr->next;
27     }
28     for ( ; j < lim; ++j) {
29         /*
30          ** Si noti la facilità estrema con la quale l'algoritmo viene
31          ** modificato per generare solo derangements .
32         */
33         if (ptr->next->data != n -1) {
34             D[n -1] = ptr->next->data;
35             PushNode(ptr->next);
36             ptr->next = ptr->next->next;
37             cnt += ER_derange(n -1, k + j -n +1);
38             ptr->next = PopNode();
39         }
40         ptr = ptr->next;
41     }
42 }
43 return cnt;
44 }
```

Il codice, compilato con le definizioni di default ed eseguito, produce il seguente output:

```

Derangements di 5 (44) , algoritmo di Effler-Ruskey ,
in ordine crescente di numero di inversioni :
 3: 10342 10423 12043 20143
 4: 12340 20341 13042 30142 12403 20413 14023 40123
 5: 23041 13402 30412 24013
 6: 13420 23140 23401 30421 32041 14302 34012 40312 24103 42013
 7: 23410 14320 32140 24301 32401 34021 40321 34102 43012 42103
 8: 24310 32410 34120 42301 43021 43102
 9: 42310 43120
** Totale: 44 **
```

Come già notato con l'eq. (16), il numero minimo di inversioni presenti in un derangement è pari a $k_{min} = \lceil \frac{n}{2} \rceil$ ovvero, per $n = 5$, $k_{min} = 3$. Avendo una lunghezza dispari, risulta inoltre escluso il caso limite con tutte le $\binom{5}{2} = 10$ coppie scambiate. Si invita il lettore a riflettere sulla controintuitiva relazione tra le formule (5) e (8) e la distribuzione dei derangements per numero di inversioni, e ad approfondirne per suo conto gli interessanti risvolti.

5.2 Algoritmi di Mikawa-Tanaka.

Si tratta di una coppia di algoritmi di *ranking* e *unranking*, pubblicati nel 2014 [MT14]. Gli algoritmi operano in $O(n \cdot \log_2 n)$ sia per il ranking che per l'unranking e sono, al momento attuale, gli unici disponibili in letteratura caratterizzati da una simile prestazione e da una richiesta di memoria $O(n)$ per generare derangements (in notazione ciclica, v. §2.4) in ordine lessicografico. L'unico algoritmo in competizione prestazionale risulta essere, ad oggi, quello di Myrvold-Ruskey [MR01]²⁷, che opera in tempo lineare, sebbene l'ordine di generazione non sia lessicografico.

5.2.1 Algoritmo di unranking di Mikawa-Tanaka.

Analizziamo innanzi tutto l'algoritmo di unranking per i derangements proposto da Mikawa e Tanaka. Iterativo, strutturalmente molto semplice, prende in input un numero intero non negativo (il *rank*) e genera il corrispondente derangement, espresso in notazione ciclica.

Le principali variabili utilizzate nello pseudocodice e nelle relative formule sono:

- Il rank r , un valore intero $0 \leq r < D(n)$;
- Il vettore d'inversione $V[]$, di ampiezza n ;
- Il vettore per la notazione ciclica $Cy[]$, di dimensione n ;
- Il vettore per il derangement $D[]$, ancora di dimensione n .

L'algoritmo si basa su due passi sequenziali, brevemente illustrati nei paragrafi successivi.

Step 1. Il *rank* r (ordinale) viene convertito nel corrispondente *vettore di inversione*, in una notazione peculiare elaborata dagli autori, che differisce strutturalmente dalle notazioni viste al §2.10 essendo basata sulle proprietà specifiche dei derangements e non su quelle delle permutazioni in generale. In particolare, per costruzione tale vettore d'inversione non conterrà in alcun caso due o più zeri consecutivi, e non gode della proprietà (12), dunque in definitiva la sommatoria delle sue cifre non indica il numero totale di inversioni nel derangement generato, diversamente da quasi ogni altra analoga tipologia.

Gli autori forniscono implicitamente due alternative per il calcolo dei valori nel vettore di inversione $V[]$: si è ovviamente scelta quella con prestazioni migliori, operante in $O(n)$. Tale algoritmo richiede il precalcolo del numero di derangements, qui denotato $De[k]$, in funzione della lunghezza k (13), implementato ovviamente tramite *LUT*, ed è basato sul calcolo incrementale del *rank parziale* x_i , stabilito dalle seguenti semplici equazioni:

$$x_i = \begin{cases} r & \text{se } i = 0; \\ x_{i-1} & \text{se } V[i-1] = 0; \\ x_{i-1} - (V[i-1] - 1)(De[n-i] + De[n-i-1]) & \text{se } i = 1 \text{ oppure } V[i-2] = 0; \\ x_{i-1} - (V[i-1] - 1)(De[n-i] + De[n-i-1]) - De[n-i] & \text{se } V[i-1] \neq 0 \text{ e } V[i-2] \neq 0; \end{cases}$$

Dal *rank parziale* x_i si calcola poi, in cascata, la corrispondente cifra del vettore di inversione $V[i]$:

$$V[i] = \begin{cases} 0 & \text{se } V[i-1] \neq 0 \text{ e } x_i < De[n-i-1] \\ \left\lfloor \frac{x_i}{De[n-i-1] + De[n-i-2]} \right\rfloor + 1 & \text{se } i = 0 \text{ oppure } V[i-1] = 0 \\ \left\lfloor \frac{x_i - De[n-i-1]}{De[n-i-1] + De[n-i-2]} \right\rfloor + 1 & \text{in ogni altro caso} \end{cases}$$

Il calcolo del vettore di inversione è ulteriormente facilitato per le posizioni 0, 1 ed $n - 1$ data la peculiare struttura del vettore stesso, la cui prima cifra è sicuramente non nulla²⁸, mentre l'ultima cifra è nulla per definizione. Ne scaturisce in modo pressoché immediato il seguente codice sorgente.

```
1 /* **** */
2 /* Algoritmo di unranking Mikawa-Tanaka, step 1: */
3 /* converte un rank in un vettore di inversione. */
```

²⁷Di tale algoritmo l'autore del presente articolo ha a suo tempo proposto una originale implementazione in questo post.

²⁸La prima cifra del vettore d'inversione viene a coincidere infatti, per costruzione, con la prima cifra del derangement - il quale, per definizione, in tale posizione non può avere uno 0, usando le universali convenzioni informatiche di numerazione e indicizzazione.

```

4  /*************************************************************************/
5  void TM_rank2inv(const unsigned int rank, unsigned int *v)
6  {
7      /*
8      ** Variabile automatica ridondante, per pura uniformita' di
9      ** notazione con l'articolo.
10     */
11    size_t n = Derange.Width;
12    size_t i, x, x1;
13    /* I tre valori notevoli vengono precalcolati fuori dal loop */
14    v[n - 1] = 0;
15    v[0] = 1 + rank / (De[n - 1] + De[n - 2]);
16    x = rank - (v[0] - 1) * (De[n - 1] + De[n - 2]);
17    if (x < De[n - 2]) {
18        v[1] = 0;
19    } else {
20        v[1] = 1 + (x - De[n - 2]) / (De[n - 2] + De[n - 3]);
21    }
22    /*
23    ** Le formule fornite consentono di ricavare ogni elemento
24    ** del vettore di inversione da una semplice terna di valori,
25    ** in tempo costante, incluso il precedente valore di  $X_i$ .
26    ** Tale valore viene qui salvato in  $x1$  per ogni passo successivo.
27    */
28    x1 = x;
29    for (i = 2; i < n - 1; ++i) {
30        /* Si ricava innanzi tutto il rank parziale  $X_i$ , grazie a  $x1$  */
31        if ((v[i - 1] != 0) && (v[i - 2] != 0)) {
32            x = x1 - (v[i - 1] - 1) * (De[n - i] + De[n - i - 1]) - De[n - i];
33        } else if (v[i - 1] == 0) {
34            x = x1;
35        } else if (v[i - 2] == 0) {
36            x = x1 - (v[i - 1] - 1) * (De[n - i] + De[n - i - 1]);
37        }
38        x1 = x;
39        /* Da tale rank si passa facilmente al numero di inversioni. */
40        if (0 == v[i - 1]) {
41            v[i] = 1 + (x / (De[n - i - 1] + De[n - i - 2]));
42        } else if ((0 != v[i - 1]) && (x < De[n - i - 1])) {
43            v[i] = 0;
44        } else {
45            v[i] = 1 + (x - De[n - i - 1]) / (De[n - i - 1] + De[n - i - 2]);
46        }
47    }
48 }

```

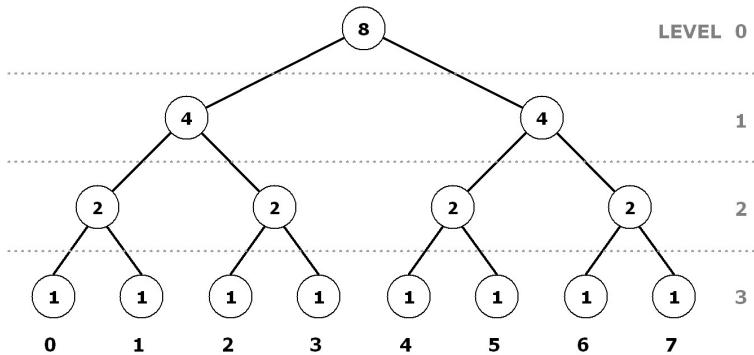


Figura 1: Esempio di albero di unranking Mikawa-Tanaka inizializzato.

Step 2. Il vettore di inversione $V[]$ ricavato allo **Step 1** viene convertito nel corrispondente derangement in *notazione ciclica* (v. §2.4), grazie ad un albero binario adeguatamente inizializzato. Tale parte dell'algoritmo risulta ispirata da altre simili realizzazioni già presenti in letteratura, in particolare [Bon08], alla quale gli autori rimandano per i dettagli della conversione basata su *BTree*.

Nello pseudocodice che segue, *Root* è il nodo radice dell'albero binario e si usa intuitivamente la *dot notation* per indicare gli elementi del nodo corrente, in particolare *node.val* per il valore e *node.left*, *node.right* per i puntatori ai rispettivi figli, quindi l'indicazione ricorsiva semplificata *node.left.val* indica il valore del figlio sinistro del nodo corrente, con semantica sostanzialmente identica a quella dei puntatori a strutture in C.

L'algoritmo richiede un albero binario completo con un numero di foglie non inferiore alla lunghezza n dei derangement da generare. In particolare, l'albero avrà $\lambda = 1 + \lceil \log_2(n) \rceil$ livelli: il numero di foglie dell'albero sarà quindi pari a $\mathcal{F}(\lambda) = 2^{\lambda-1} \geq n$, e vi saranno un totale di $\mathcal{N}(\lambda) = 2^\lambda - 1$ nodi. L'albero deve essere inizializzato in modo che ad ogni livello $0 \leq i < \lambda$, convenzionalmente ordinato in modo *crescente* a partire dal livello che contiene il nodo radice, il valore dei nodi sia $2^{\lambda-i-1}$, come illustrato in fig. 1. In tal modo ciascun nodo, eccetto le foglie, contiene la somma dei valori dei suoi figli.

La figura 2 a pag. 32 mostra invece l'intera successione dei percorsi di attraversamento e dei valori nell'albero per una conversione esemplificativa con $n = 7$, $\lambda = 1 + \lceil \log_2(7) \rceil = 4$, quindi un albero con $\mathcal{F}(4) = 2^3 = 8$ foglie (una delle quali resterà ovviamente inutilizzata, avendo $\mathcal{F}(\lambda) - n = 1$) e $\mathcal{N}(4) = 2^4 - 1 = 15$ nodi in totale. Nell'esempio si decodifica il *rank* 690, corrispondente al vettore d'inversione $V[] = \langle 3123010 \rangle$, nel derangement in notazione ciclica $Cy[] = \langle 31460 \rangle(52)$, che equivale a $D[] = \langle 3451620 \rangle$ in notazione 1-line²⁹.

Le figure mostrano inoltre come le foglie dell'albero siano in realtà caratterizzate anche da un attributo, riportato al di sotto del nodo terminale stesso, con valore compreso tra zero e $2^{\lambda-1} - 1$, che verrà usato per valorizzare il vettore dei cicli al termine di ogni cammino. Considerando ad esempio il primo step (a), si ha che le direzioni del cammino percorso dall'algoritmo partendo dal nodo radice sono SINISTRA, DESTRA, DESTRA. Al termine, come evidenziato dal commento, a $Cy[0]$ viene appunto assegnato il valore 3, ossia *l'attributo* della quarta foglia, raggiunta con tale cammino. Il valore contenuto in tutti i λ nodi (uno per ogni livello) attraversati dal percorso viene decrementato di una unità: questo influenza gli esiti dei confronti nei cicli successivi e, in definitiva, garantisce che ciascuna foglia sia raggiunta esattamente zero o una volta. Dallo pseudocodice e grazie all'esempio risulta infatti immediato rilevare che l'algoritmo itera n volte un attraversamento *top-down* dell'albero, il quale interessa ogni volta λ nodi, e quindi in definitiva lo **Step 2** lavora per costruzione in $O(n \cdot \lambda) = O(n \cdot \log_2 n)$.

```

procedure TM_inv2cycles(V, Cy)
    Init_Tree()
    for 0 ≤ i < n do
        node ← Root
        for 0 ≤ layer < λ do
            node.val ← node.val - 1
            if node.left.val > V[i] then
                node ← node.left
            else
                V[i] ← V[i] - node.left.val
                node ← node.right
            Cy[i] ← node.attrib
    return

```

Nel codice sorgente C proposto l'accesso all'albero binario è ottimizzato sotto due aspetti fondamentali. In primo luogo, l'albero completo è emulato tramite un *binary heap*, ossia un semplice array³⁰, in modo che valgano le seguenti formule di indirizzamento, ove si hanno gli ovvi limiti³¹ $0 \leq i < 2^\lambda - 1$ e $0 \leq h < \lambda$:

²⁹Si noti come le restrizioni imposte dalla definizione di derangement si applicano anche al corollario sulla leggibilità diretta delle notazioni cicliche come permutazioni, ricordato al §2.7: in questo esempio, come in molti altri, $Cy[]$ deprivato opportunamente delle parentesi contiene una permutazione valida, ma non può essere un derangement, avendo $Cy[1] = 1$. Il fatto che gli algoritmi di Mikawa e Tanaka lavorino direttamente sulla notazione ciclica può talora confondere gli studenti e i *practitioners* meno familiari con la generazione combinatoria.

³⁰L'array di piccoli interi usato per emulare l'albero impiega solo una frazione dello spazio richiesto da strutture nodo ed elimina i costi, i tempi ed i rischi di numerose allocazioni e deallocazioni, al costo di operazioni aritmetiche che normalmente si traducono in semplici coppie di istruzioni atomiche (*shift* e incrementi unitari di un registro o somme).

³¹Limiti chiaramente validi per tutti i linguaggi di programmazione con indicizzazione 0-based.

$Root = 0$ $Parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$ $Left(i) = 2i + 1$ $Right(i) = 2(i+1)$	$Sibling(i) = \begin{cases} i-1 & \text{se } i \text{ è pari (figlio destro)} \\ i+1 & \text{se } i \text{ è dispari (figlio sinistro)} \end{cases}$ $NpL(h) = 2^h$ $Level(h) = 2^h - 1$
----------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$Left(i)$ e $Right(i)$ sono intuitivamente gli indici dei due nodi figli rispetto al generico nodo i -esimo, con l'ovvia eccezione delle foglie, prive di figli, pertanto le rispettive equazioni sono applicabili per $0 \leq i < 2^{\lambda-1} - 1$.

$Parent(i)$ è il nodo padre e $Sibling(i)$ è il nodo fratello, figlio del medesimo padre, ovviamente ambedue esistenti per ogni nodo tranne $Root$, quindi per ogni $i \neq 0$.

$NpL(h)$ rappresenta il numero di nodi per ogni dato *layer* o livello h , mentre $Level(h)$ è l'indice dell'array dal quale inizia il livello dato (dunque 0, 1, 3, 7, 15, ... per i livelli 0, 1, 2, 3, 4, ...). Ogni nodo, tranne ovviamente $Root$, è figlio *destro* del proprio padre se e solo se è collocato ad un indice *pari* nell'array, e simmetricamente per i figli sinistri/indici dispari.

Tali banali formule algebriche consentono come noto un completo attraversamento dell'albero: in particolare gli attraversamenti *top-down*, *bottom-up*, *level-order* qui variamente necessari. In questa implementazione, l'attributo delle foglie è anch'esso facilmente ricavato per via algebrica dall'indice della foglia, sottraendo allo stesso la quantità $Level(\lambda-1) = 2^{\lambda-1} - 1$ che demarca appunto l'inizio dell'ultimo livello dell'albero.

Come seconda e subordinata ottimizzazione di design, l'inizializzazione per livelli dell'albero (teoricamente da ripetersi per ciascuna invocazione della funzione di unranking) è ottenuta per copia da un secondo array appositamente predisposto, in modo da minimizzare i tempi sfruttando la persistenza in cache e le numerose ottimizzazioni platform-dependent della copia per blocchi implementate nella `memcpy()` della libreria standard, in grado di offrire prestazioni sicuramente superiori rispetto al codice di inizializzazione con valori distinti per livelli. Ciò naturalmente al costo di un raddoppio della richiesta di memoria, che come abbiamo visto rimane dell'ordine di $O(n)$ e risulta complessivamente pari a $\mathcal{N}(\lambda) = 2^\lambda - 1$ volte il *footprint* di un (piccolo) intero per ogni albero allocato.

```

1  /*************************************************************************/
2  /* Algoritmo di unranking Mikawa-Tanaka , step 2:                  */
3  /* converte un vettore di inversione in un derangement , espresso in   */
4  /* notazione ciclica , usando un albero binario .                      */
5  /*************************************************************************/
6  void TM_inv2cycles(derange_t *v, derange_t *cy)
7  {
8      size_t n = Derange.Width;
9      size_t i;
10     /* Inizializzazione per copia. */
11     memcpy(Derange.BTree, Derange.Tree, Derange.Nodes * sizeof(derange_t));
12     for (i = 0; i < n; ++i) {
13         size_t j = 0;
14         size_t le;
15         /* Visita top-down del BTree. */
16         for (le = 0; le < Derange.Layers; ++le) {
17             /* Decrementa il valore al nodo corrente. */
18             Derange.BTree[j] -= 1;
19             if(v[i] < Derange.BTree[1 + (j << 1)]) {
20                 /* Visita il figlio a SINISTRA. */
21                 j = 1 + (j << 1);
22             } else {
23                 /*
24                  ** Sottrae da v[i] il valore del figlio SINISTRO,
25                  ** poi visita il figlio a DESTRA.
26                 */
27                 v[i] -= Derange.BTree[1 + (j << 1)];
28                 j = 2 + (j << 1);
29             }
30         }
31         /*
32          ** Decrementa anche il valore della foglia ,
33          ** sebbene a rigore non sia necessario .

```

```

34         */
35         Derange.BTree[ j ] == 1;
36         /* Assegna alla locazione corrente dell'array "l'attributo" della foglia
37         */
38         cy[ i ] = j - ((1 << (Derange.Layers)) -1);
39     }

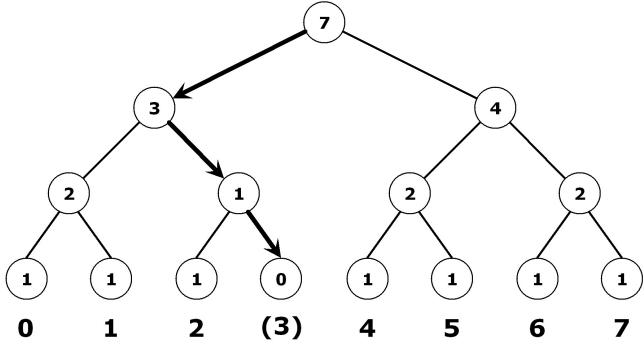
```

Come ultimo passo, non previsto dall'algoritmo originale, si propone il semplicissimo codice che converte la notazione ciclica in notazione lineare in tempo $O(n)$, con l'uso di due sole variabili ausiliarie. Si ponga attenzione al fatto che la notazione Mikawa-Tanaka richiede una scansione *right-to-left*, contrariamente a quella standard introdotta da Stanley (v. §2.4).

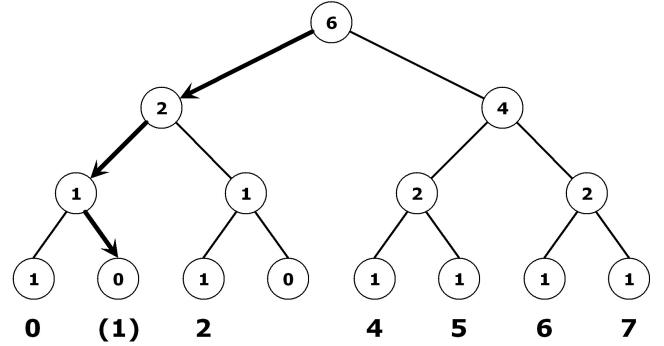
```

1   int i ;
2   unsigned int m; /* Valor minimo relativo , che chiude ogni ciclo . */
3
4   m = cy[ n -1];
5   i = n -2;
6   while ( i > -1) {
7       if( cy[ i ] < m) {
8           D[m] = cy[ i +1];
9           m = cy[ i ];
10      } else {
11          D[ cy[ i ] ] = cy[ i +1];
12      }
13      —i ;
14  }
15  D[m] = cy[ i +1];

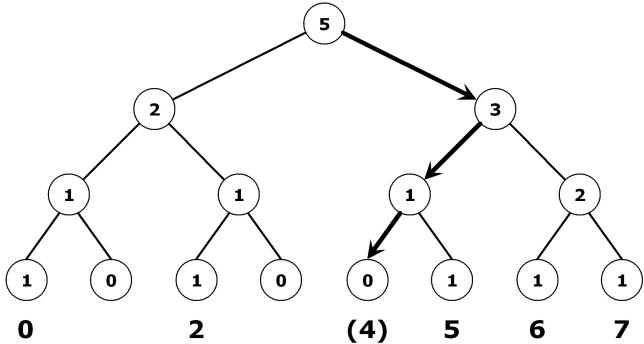
```



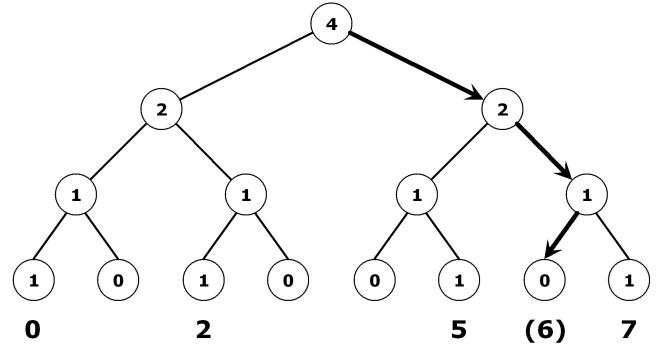
(a) $V[0] = 3$, $Cy[0] = 3$.



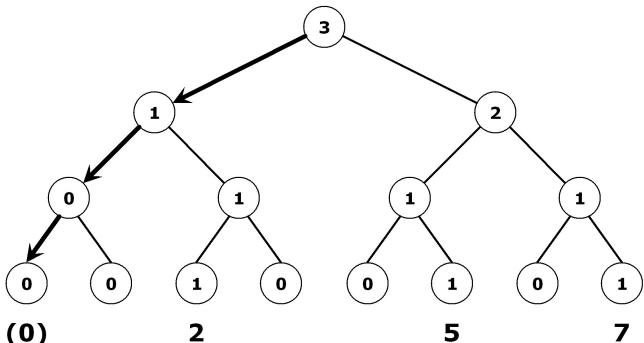
(b) $V[1] = 1$, $Cy[1] = 1$.



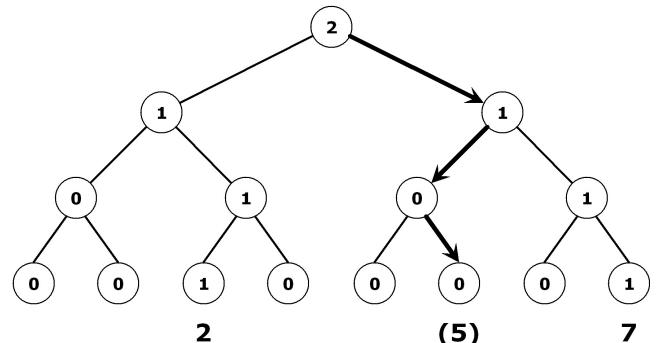
(c) $V[2] = 2$, $Cy[2] = 4$.



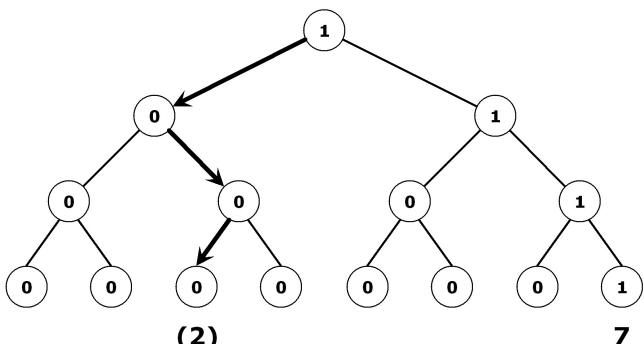
(d) $V[3] = 3$, $Cy[3] = 6$.



(e) $V[4] = 0$, $Cy[4] = 0$.



(f) $V[5] = 1$, $Cy[5] = 5$.



(g) $V[6] = 0$, $Cy[6] = 2$.

Figura 2: Esempio di unranking Mikawa-Tanaka: $n = 7$, $Rank = 690 \mapsto V[] = \langle 3123010 \rangle \mapsto Cy[] = (31460)(52)$.

5.2.2 Algoritmo di ranking di Mikawa-Tanaka.

L'algoritmo di ranking prende in input un array contenente il derangement espresso in *notazione ciclica* e ne restituisce il rank (ordinale), ossia in ultima analisi il numero di derangements che lo precedono nell'ordine

lessicografico, in tempo $O(n \cdot \log_2 n)$. Tale algoritmo iterativo risulta ancora articolato in due fasi ed è totalmente speculare a quello di unranking appena visto al paragrafo precedente.

Le principali variabili utilizzate sono a loro volta identiche a quelle dell'algoritmo di unranking:

- Il rank r , un valore intero $0 \leq r < D(n)$;
- Il vettore d'inversione $V[]$, di ampiezza n ;
- Il vettore per la notazione ciclica $Cy[]$, di dimensione n ;
- Il vettore per il derangement $D[]$, ancora di dimensione n .

Step 1. Come primo passo, l'algoritmo ricava il vettore d'inversione dalla notazione ciclica data, visitando *bottom-up* un albero binario completo, in tutto identico a quello già descritto per la procedura di unranking, ma inizializzato con ogni nodo a zero, come visibile in fig. 3.

Il vettore dei cicli viene inizialmente copiato nel vettore d'inversione, elemento per elemento, in modo tale che $V[i] = Cy[i]$ per $0 \leq i < n$. La pseudofunzione `ISRIGHT(node)` restituisce un valore logico TRUE se il nodo corrente è figlio destro del suo genitore. La pseudofunzione `GETNODE(attr)` restituisce il puntatore alla foglia corrispondente all'attributo `attr` con valori in $[0, n - 1]$: si veda la descrizione dell'indirizzamento aritmetico utilizzato nel *binary heap* nel punto precedente dedicato al ranking.

La visita è poi vincolata a due semplicissime operazioni: come già indicato in precedenza, al fine di semplificare la notazione dello pseudocodice, ogni nodo contiene figurativamente puntatori al proprio genitore, ai propri figli ed anche al fratello, ossia all'altro figlio del proprio nodo padre (i.e. `node.left`, `node.right`, `node.parent`, `node.sibling`) con le ovvie eccezioni in caso di *Root* e foglie³².

La figura 4 a pag. 36 mostra un completo esempio di conversione da notazione ciclica a vettore di inversione, per il derangement $D[] = \langle 3451620 \rangle$ in notazione 1-line, che corrisponde a $Cy[] = (31460)(52)$ e genera il vettore d'inversione $V[] = \langle 3123010 \rangle$ da cui si ricava, con lo **Step 2** di seguito descritto, il rank $r = 690$.³³

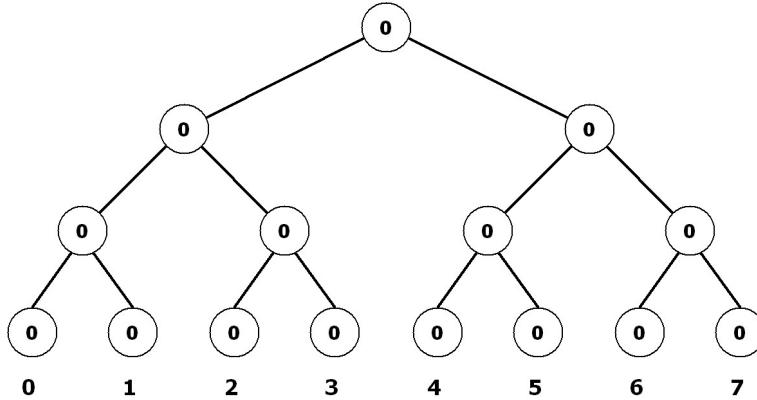


Figura 3: Esempio di albero di ranking Mikawa-Tanaka inizializzato.

```

procedure TM_cycles2inv(V,Cy)
  Init_Tree()
  for 0 ≤ i < n do
    node ← GETNODE(Cy[i])
    for 0 ≤ layer < λ do
      node.val ← node.val + 1
      if ISRIGHT(node) then
        V[i] ← V[i] - node.sibling.val
      node ← node.parent
  return
  
```

³²In realtà, come già visto in merito all'unranking, l'implementazione degli alberi avviene tramite un *binary heap*, ossia un opportuno array nel quale i nodi sono disposti in ordine *breadth-first* e indicizzati nel modo più efficiente con banali operazioni booleane e algebriche.

³³Si noti come, a rigore, non è strettamente necessario aggiornare il valore nel nodo radice al termine dell'attraversamento dell'albero: in questo caso il codice e la figura d'esempio ignorano tale passaggio. Tale valore potrebbe essere utile unicamente per un controllo ridondante di congruenza.

Step 2. Il vettore d'inversione $V[]$ ricavato al passo precedente viene convertito nel corrispondente rank, secondo la seguente semplice formulazione condizionale. Anche in questo caso, come per l'algoritmo di unranking Mikawa-Tanaka appena visto, è richiesto il precalcolo del numero di derangements, qui nuovamente denotato $De[k]$, in funzione della lunghezza k (13).

$$r = \sum_{i=0}^{n-1} \begin{cases} 0 & \text{se } V[i] = 0 \\ (V[i] - 1)(De[n-i-1] + De[n-i-2]) & \text{se } i = 0 \text{ oppure } V[i-1] = 0 \\ (V[i] - 1)(De[n-i-1] + De[n-i-2]) + De[n-i-1] & \text{in ogni altro caso} \end{cases}$$

In merito alla complessità computazionale, valgono le considerazioni già viste per la procedura di unranking: qui lo **Step 1** opera ancora in $O(n \cdot \lambda) = O(n \cdot \log_2 n)$, eseguendo n attraversamenti *bottom-up* dell'albero per una conversione, e **Step 2** opera in $O(n)$.

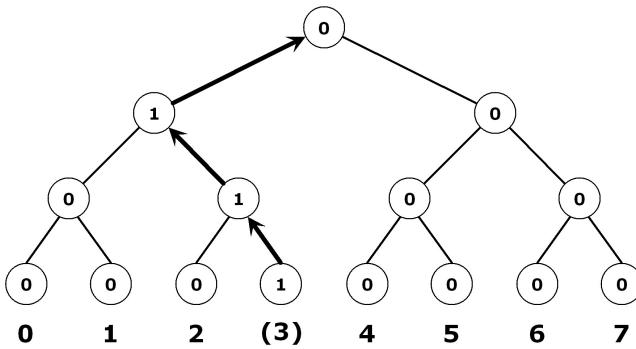
Segue un estratto del relativo codice sorgente.

```

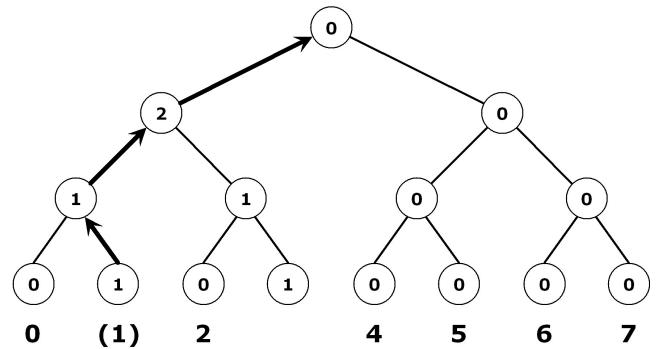
1  /*************************************************************************/
2  /* Algoritmo di ranking Mikawa-Tanaka. */
3  /* NB: Il derangement cy deve essere espresso in notazione ciclica. */
4  /*************************************************************************/
5  unsigned int TM_rank(derange_t *cy)
6  {
7      /*
8      ** Variabile automatica ridondante, per pura uniformita' di
9      ** notazione con l'articolo.
10     */
11    size_t n = Derange.Width;
12    size_t i;
13
14    unsigned int ra = 0L;
15    derange_t v[MAX_DERANGE];
16
17    memset(Derange.BTree, 0, Derange.Nodes * sizeof(derange_t));
18    memcpy(v, cy, MAX_DERANGE * sizeof(derange_t));
19
20    /* Conversione del derangement (cycle) in vettore di inversione. */
21    for (i = 0; i < n; ++i) {
22        size_t j = cy[i] + ((1 << Derange.Layers) - 1);
23        size_t le;
24
25        /* Visita bottom-up del BTREE. */
26        for (le = 0; le < Derange.Layers; ++le) {
27            /* Incrementa il valore al nodo corrente. */
28            Derange.BTree[j] += 1;
29            /*
30            ** Se si tratta del figlio DESTRO, sottrae da v[i] il valore
31            ** contenuto nel figlio SINISTRO.
32            */
33            if(0 == (j & 1)) {
34                v[i] -= Derange.BTree[j - 1];
35            }
36            /* Visita il padre. */
37            j = (j - 1) >> 1;
38        }
39    }
40
41    /* Calcolo del rank dal vettore di inversione. */
42    ra += (v[0] - 1)*(De[n - 1] + De[n - 2]);
43    for (i = 1; i < n; ++i) {
44        if (v[i] != 0) {
45            ra += (v[i] - 1)*(De[n - i - 1] + De[n - i - 2]);
46        }
47    }
48
49    return ra;
50}

```

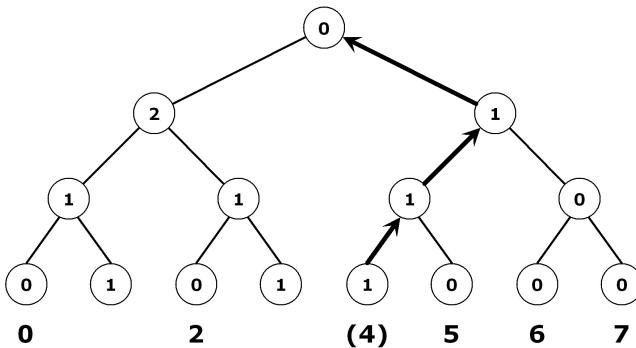
```
46         if (v[ i -1] != 0) {  
47             ra += De[n -i -1];  
48         }  
49     }  
50  
51     return ra;  
52 }  
53 }
```



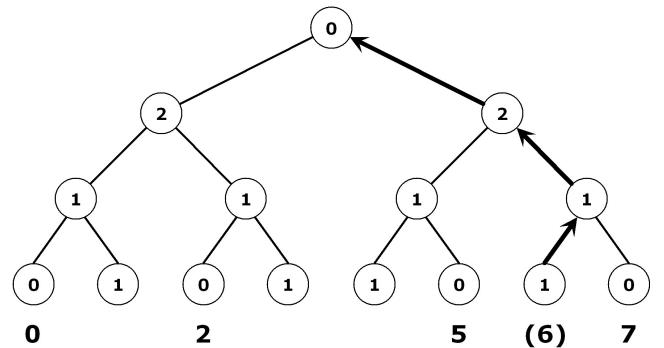
$$(a) Cy[0] = 3, V[0] = 3.$$



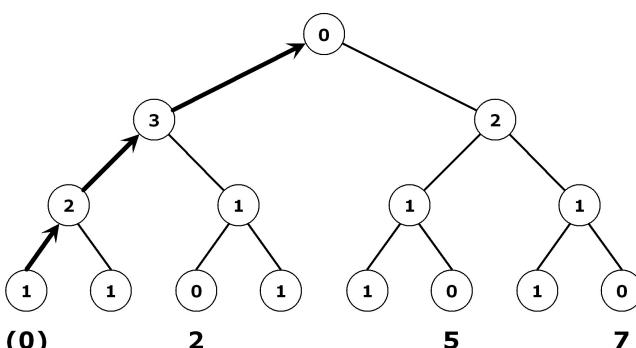
$$(b) Cy[1] = 1, V[1] = 1.$$



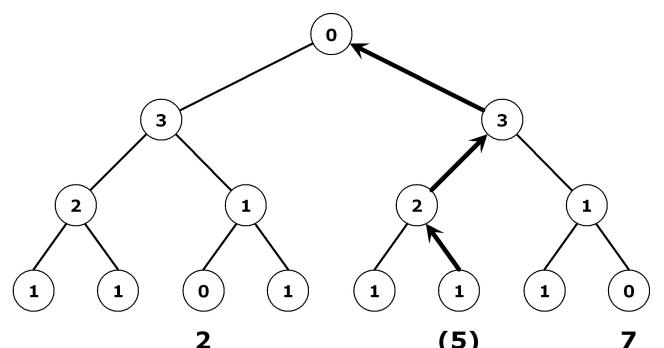
(c) $Cy[2] = 4, V[2] = 2.$



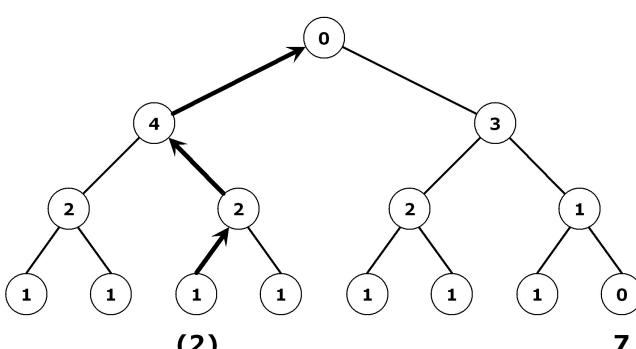
(d) $C_y[3] = 6, V[3] = 3.$



$$(e) Cv[4] = 0, V[4] = 0.$$



(f) $Cv[5] = 5, V[5] = 1.$



(g) $Cv[6] = 2$, $V[6] = 0$.

Figura 4: Esempio di ranking Mikawa-Tanaka: $n = 7$, $C_U[\cdot] = \langle 31460 \rangle (52) \mapsto V[\cdot] = \langle 3123010 \rangle \mapsto Rank = 690$.

5.2.3 Output dell'implementazione d'esempio.

Al fine di consentire una migliore comprensione dei due algoritmi e delle peculiarità della notazione ciclica scelta dagli autori, si è dotato il codice d'esempio di un output arricchito, che mostra in modo immediato le relazioni tra derangement, rank, e varie notazioni cicliche, incluse quelle generiche (riferite alle permutazioni in generale) già viste ed esemplificate al paragrafo 2.10.

Nel caso di $n = 5$, l'output si presenta come segue:

Derangements di 5 (44), algoritmo di Mikawa-Tanaka:										
Ra	Inv	#	Cycle	Derangement	b-inv	B-inv	c-inv	C-inv	##	
0	10110	3	10342	10342 (0)	10200	01002	10110	01011	3	
1	10210	4	10432	10423 (1)	10110	01011	10200	01002	3	
2	11010	3	12043	12043 (2)	20010	00201	11010	01101	3	
3	11110	4	12340	12340 (3)	40000	00004	11110	01111	4	
4	11210	5	12430	12403 (4)	30010	00031	11200	01102	4	
5	12010	4	13042	13402 (5)	30200	00032	12200	01022	5	
6	12110	5	13240	13420 (6)	40200	00024	12210	01122	6	
7	12210	6	13420	13042 (7)	20200	00202	12010	01021	4	
8	13010	5	14032	14320 (8)	40210	00124	13210	01123	7	
9	13110	6	14230	14302 (9)	30210	00132	13200	01023	6	
10	13210	7	14320	14023 (10)	20110	00211	13000	01003	4	
11	20110	4	20341	23041 (11)	23000	00203	22010	00221	5	
12	20210	5	20431	24013 (12)	22010	00221	23000	00203	5	
13	21010	4	21043	20143 (13)	11010	01101	20010	00201	3	
14	21110	5	21340	23140 (14)	42000	00204	22110	01221	6	
15	21210	6	21430	24103 (15)	32010	00231	23100	01203	6	
16	22010	5	23041	24301 (16)	33010	00133	23200	00223	7	
17	22110	6	23140	24310 (17)	43010	00134	23210	01223	8	
18	22210	7	23410	20341 (18)	13000	01003	20110	00211	4	
19	23010	6	24031	23410 (19)	43000	00034	22210	01222	7	
20	23110	7	24130	23401 (20)	33000	00033	22200	00222	6	
21	23210	8	24310	20413 (21)	12010	01021	20200	00202	4	
22	30110	5	30241	32401 (22)	33100	01033	32200	00232	7	
23	30210	6	30421	34102 (23)	32200	00232	33100	01033	7	
24	31010	5	31042	30412 (24)	12200	01022	30200	00032	5	
25	31110	6	31240	32410 (25)	43100	01034	32210	01232	8	
26	31210	7	31420	34012 (26)	22200	00222	33000	00033	6	
27	32010	6	32041	34021 (27)	23200	00223	33010	00133	7	
28	32110	7	32140	34120 (28)	42200	00224	33110	01133	8	
29	32210	8	32410	30421 (29)	13200	01023	30210	00132	6	
30	33010	7	34021	32140 (30)	42100	01204	32110	01231	7	
31	33110	8	34120	32041 (31)	23100	01203	32010	00231	6	
32	33210	9	34210	30142 (32)	11200	01102	30010	00031	4	
33	40110	6	40231	42310 (33)	43110	01134	42210	01224	9	
34	40210	7	40321	43120 (34)	42210	01224	43110	01134	9	
35	41010	6	41032	40321 (35)	13210	01123	40210	00124	7	
36	41110	7	41230	42301 (36)	33110	01133	42200	00224	8	
37	41210	8	41320	43021 (37)	23210	01223	43010	00134	8	
38	42010	7	42031	43012 (38)	22210	01222	43000	00034	7	
39	42110	8	42130	43102 (39)	32210	01232	43100	01034	8	
40	42210	9	42310	40312 (40)	12210	01122	40200	00024	6	
41	43010	8	43021	42103 (41)	32110	01231	42100	01204	7	
42	43110	9	43120	42013 (42)	22110	01221	42000	00204	6	
43	43210	10	43210	40123 (43)	11110	01111	40000	00004	4	

** Totale: 44 **

Osservando le varie colonne, si notano i seguenti campi:

Ra il rank, in ordine crescente, relativo al derangement corrente. Come già evidenziato, si tratta di un numero intero tale che $0 \leq Rank < D(n)$.

Inv il vettore d'inversione, costruito con i criteri di Mikawa e Tanaka sopra esposti.

la somma delle cifre del vettore d'inversione. Si noti come, in questo caso, non si applichi la proprietà dell'eq. (12) che invece connota altri tipi di *inversion vector*.

Cycle il vettore in notazione ciclica ricavato dal rank corrente.

Derangement il derangement corrispondente al campo Cycle e quindi a Rank, in notazione 1-line, seguito dal rank ricavato a ritroso da Cycle (come test per la routine di ranking).

b-inv e successivi: i quattro vettori di inversione generici per permutazioni (b , B , c e C), come definiti al paragrafo 2.10, seguiti dalla somma delle loro cifre, riportata una sola volta.

Questo genere di output tabulato rende immediato ed intuitivo ragionare sulle relazioni tra gli oggetti combinatori e le relative statistiche richiamate nella prima parte del presente articolo, oltre a costituire un collaudo sinottico della validità delle routine implementate.

5.3 Algoritmo di Rohl-Ganapathi-Rama.

Algoritmo di recente pubblicazione (2010), presente su arXiv [GR10], costituisce una rielaborazione del già visto algoritmo ricorsivo di Rohl (§4.2.3) e viene incluso nella presente selezione principalmente per la sua particolare flessibilità e genericità. Gli autori mostrano infatti come, con piccole modifiche e adattamenti, sia possibile utilizzare l'algoritmo per generare (come minimo) le seguenti famiglie di oggetti combinatori:

- Permutazioni;
- Derangements;
- Combinazioni e sottoinsiemi;
- N-parentesizzazioni (Catalan);
- Partizioni e composizioni vincolate.

Il tutto usando tipi di ordinamento totalmente arbitrari e con un singolo algoritmo di base, il quale arricchisce il novero degli algoritmi ricorsivi generici o “universali”. Sebbene per quasi tutte le categorie elencate esistano elaborati algoritmi iterativi specifici in grado di generare un singolo oggetto in tempo lineare, CAT e talora perfino in $O(1)$, in determinate occasioni la flessibilità e, di nuovo, l'inerente semplicità strutturale di questo algoritmo possono risultare vincenti in una accurata analisi ingegneristica costi/benefici: tipicamente per generare un numero ristretto di oggetti con un ordinamento non convenzionale, e più in generale entro una implementazione in linguaggi funzionali e/o con esplicito supporto all'ottimizzazione della ricorsione.

L'algoritmo nella sua versione generica richiede in input tre array e due interi, e genera l'oggetto combinatorio richiesto in un distinto array di output $\mathcal{D}[]$.

List: l'array $\mathcal{L}[] = \langle I_0, I_1, \dots, I_{n-1} \rangle$ contenente n elementi, non necessariamente distinti, che fungono da *generatori* per OrderSet e indirettamente per l'oggetto combinatorio finale. Tale array contiene $1 \leq p \leq n$ elementi unici;

OrderSet: l'array $\mathcal{O}[] = \langle u_0, u_1, \dots, u_{p-1} \rangle$ contenente tutti e soli i p elementi unici di $\mathcal{L}[]$, disposti in una sequenza che determinerà l'ordine complessivo di output;

CountArray: l'array $\mathcal{C}[] = \langle o_0, o_1, \dots, o_{p-1} \rangle$ contenente le effettive occorrenze in $\mathcal{L}[]$ relative a ciascuno dei p elementi unici di $\mathcal{O}[]$, ordinato e disposto in modo tale che, per ogni $0 \leq i < p$, $\mathcal{C}[i]$ rappresenta il numero di occorrenze di $\mathcal{O}[i]$ in $\mathcal{L}[]$;

p: l'intero privo di segno, tale che $0 < p \leq n$, che esprime il numero totale di elementi unici in $\mathcal{L}[]$ e quindi in definitiva la dimensione di $\mathcal{O}[]$ e $\mathcal{C}[]$;

r: un altro intero non segnato, esprimente l'ampiezza della presentazione, tale che $0 < r \leq n$.³⁴

A titolo di esempio, quindi, volendo gestire le permutazioni del multinsieme $\mathcal{L}[] = \langle a, a, a, b, b, c, c, c, c \rangle$ in semplice ordine lessicografico inverso, si avrà ovviamente $n = 9 = 3 + 2 + 4$ e $p = 3$. Gli array sopra descritti diventano $\mathcal{O}[] = \langle c, b, a \rangle$, $\mathcal{C}[] = \langle 4, 2, 3 \rangle$. Si noti, in particolare, la corrispondenza posizionale tra $\mathcal{C}[i]$ e $\mathcal{O}[i]$ per $0 \leq i < p$.

Dal punto di vista prestazionale, gli autori ricavano esplicitamente un *upper bound* non particolarmente entusiasmante: $O(|Obj| \cdot r \cdot p)$, dove $|Obj|$ è la cardinalità del (sotto)insieme di oggetti $\mathcal{D}[]$ generati, r è come appena visto l'ampiezza della presentazione e p l'ampiezza dell'insieme di scelta, il che si traduce ad esempio in $O(n! \cdot n^2)$ nel caso di generazione esaustiva di permutazioni, laddove si ha $r = p = n$. Da notare che, nel suo articolo originale [Roh78], lo stesso Rohl non forniva una caratterizzazione del suo algoritmo, giustificando la

³⁴Si noti come si può avere anche $r > p$, purché si abbia sempre $r \leq n$, il che consente di generare un'ampia gamma di restrizioni e prefissi di oggetti combinatori, come anche permutazioni e derangements di multinsiemi.

mancanza di una quantificazione della complessità computazionale con la connotazione di estrema genericità e flessibilità della procedura.

```

procedure APR(index)
    if index = r then
        output(D[0], D[1], ..., D[n - 1])
    else
        for each j ← 0 to p - 1 do
            if C[j] ≥ 1 then
                D[index] ← O[j]
                C[j] ← C[j] - 1
                APR(index + 1)
                C[j] ← C[j] + 1
    return

```

Si può ancora una volta notare la tipica struttura con *backtracking* attorno alla chiamata ricorsiva, che caratterizza praticamente tutti gli algoritmi classici qui analizzati, in particolare quelli di Akl (§4.2.2), Heap (§4.2.1) e ovviamente lo stesso Rohl (§4.2.3). L'implementazione dell'algoritmo in linguaggio C è decisamente elementare e pedissequa: in questo caso, si è scelto di implementare una versione ristretta ai soli derangements, per uniformità e coerenza tematica con gli altri algoritmi presentati. Semplicemente eliminando la seconda condizione dell'AND logico nella `if()` alle righe 14÷15, si riottiene comunque l'algoritmo generico originale, facilmente adattabile poi secondo le indicazioni dell'articolo ai più vari utilizzi.

Si noti inoltre come, anche in questo caso, nell'implementazione tutti gli array e le variabili acceduti dalla routine ricorsiva sono rigorosamente allocati nello *heap* come variabili globali (entro un singola struttura, per comodità e maggior ordine), in modo da ridurre all'assoluto indispensabile il passaggio di parametri nelle chiamate ricorsive, evitando così ulteriori penalità prestazionali.

```

1 unsigned int APR_derange(const unsigned char len)
2 {
3     size_t i;
4     unsigned int cnt = 0;
5     if (len == Derange.r) {
6         /* Attenzione alla peculiare gestione delle stampe! */
7         for (i = 0; i < Derange.r; ++i) {
8             putchar(Derange.D[i]);
9         }
10        putchar('\t');
11        ++cnt;
12    } else {
13        for (i = 0; i < Derange.Width; ++i) {
14            if ((Derange.CountArray[i] >= 1) &&
15                (Derange.List[len] != Derange.OrderSet[i])) {
16                Derange.D[len] = Derange.OrderSet[i];
17                Derange.CountArray[i]--;
18                cnt += APR_derange(len + 1);
19                Derange.CountArray[i]++;
20            }
21        }
22    }
23    return (cnt);
24 }

```

Preliminarmente all'algoritmo vero e proprio, potrebbe risultare talora necessario inizializzare a *runtime* l'array $\mathcal{C}[]$ delle occorrenze. A tale scopo, gli autori ipotizzano la presenza di una funzione di *reverse lookup* denominata $\mathcal{I}_\mathcal{O}()$ per gli elementi di $\mathcal{L}[]$, tale che $\mathcal{I}_\mathcal{O}(\mathcal{L}[i]) = m \iff \mathcal{O}[m] = \mathcal{L}[i]$. Tale funzione (tipicamente implementata tramite *hashtables* o *LUT*) non è presente nell'implementazione di riferimento, in quanto non necessaria, ma può essere utile per sperimentare con variazioni dell'algoritmo.

```

procedure PopulateArray
    for each  $j \leftarrow 0$  to  $p - 1$  do
         $C[j] \leftarrow 0$ 
    for each  $j \leftarrow 0$  to  $p - 1$  do
         $pos \leftarrow \mathcal{I}_{\mathcal{O}}(\mathcal{L}[j])$ 
         $C[pos] \leftarrow C[pos] + 1$ 
    return

```

A titolo di singolo esempio della notevole flessibilità dell'algoritmo, si riporta lo pseudocodice delle semplicissime modifiche necessarie per generare composizioni ristrette di interi. In questo caso, la variabile n contiene il valore della composizione.

```

comment APR6 - Restricted Integer Compositions
procedure APR6(index)
    if  $n = 0$  then
        output( $\mathcal{D}[0], \mathcal{D}[1], \dots, \mathcal{D}[n - 1]$ )
    else
        for each  $j \leftarrow 0$  to  $p - 1$  do
            if  $n \geq \mathcal{O}[j]$  then
                 $\mathcal{D}[index] \leftarrow \mathcal{O}[j]$ 
                 $n \leftarrow n - \mathcal{O}[j]$ 
                APR6(index + 1)
                 $n \leftarrow n + \mathcal{O}[j]$ 
    return

```

Nell'algoritmo appena visto vengono formate tutte le composizioni soggiacenti al vincolo che la somma degli elementi sia uguale al valore dato n , usando solo i valori presenti nell'array $\mathcal{O}[]$. In pratica, si seleziona un sottoinsieme $\mathcal{S} = \{s_0, s_1, \dots, s_m\} \subseteq \mathcal{O}$ tale che $\sum_{i=0}^m s_i = n$. Con una semplice ulteriore modifica, è possibile implementare anche vincoli di molteplicità sugli addendi della composizione, facendo opportuno uso di `CountArray` come già visto in precedenza.

Si invita il lettore a sperimentare ampiamente le numerose possibilità del presente algoritmo secondo le indicazioni dell'articolo originale, il quale (al di là di alcune trascurabili incoerenze notazionali) risulta di facile e scorrevole lettura.

6 Conclusioni e ringraziamenti.

Dopo alcuni succinti richiami a concetti e definizioni elementari in campo combinatorio, si è fornita una brevissima descrizione del problema dei *ménages* e della sua storia, ed in tale contesto si sono presentati in totale sei distinti algoritmi generatori correlati³⁵, ben noti in letteratura, pubblicati lungo un arco temporale di oltre mezzo secolo:

Anno	Algoritmo	Paragrafo	Pag.
1963	Heap	4.2.1	16
1978	Rohl	4.2.3	19
1980	Akl	4.2.2	17
2003	Effler-Ruskey	5.1	24
2005	X di Knuth	4.2.4	20
2010	Rohl-Ganapathi-Rama	5.3	38
2014	Mikawa-Tanaka	5.2	27

Tali algoritmi sono stati accompagnati da brevissime note sulle principali caratteristiche computazionali di ciascuno e dalle relative implementazioni esemplificative in forma adeguatamente ridotta, con sorgenti in linguaggio C89 (ANSI/INCITS X3.159-1989, ISO/IEC 9899:1990). Si è particolarmente insistito sulla elegante semplicità strutturale che caratterizza gli algoritmi “classici”, ricorsivi, odometrici, con *backtracking* concepiti ben prima della svolta del millennio, indicandone al contempo i principali pregi e difetti prestazionali ed implementativi. Si sono quindi presentate alcune delle idee fondanti per le nuove generazioni di algoritmi combinatori, evidenziando per contrasto la relativa sofisticazione delle strutture dati impiegate e delle corrispondenti tecniche implementative per l’efficiente gestione (in particolare i “*dancing links*” e i *BTrees* emulati tramite array) rispetto agli approcci classici. Il tutto senza dimenticare l’esistenza di algoritmi “universali”, a più riprese analizzati in letteratura, come quello di Rohl e, in misura differente, lo stesso algoritmo X di Knuth.

Come già ricordato in apertura, il materiale qui presentato costituisce sostanzialmente una rielaborazione e riorganizzazione di contenuti già pubblicati sul *blog* dell’autore «Titolo provvisorio...» nella serie di articoli aperta da «Il valzer delle coppie». Tutti i codici sorgenti, in forma completa e immediatamente compilabile con i più diffusi ambienti mainstream (Digital Mars, Open Watcom, Visual Studio C++ Express 2010, CMake...) sono disponibili per il download direttamente su tale *blog*.

In chiusura, l’autore desidera ringraziare sentitamente tutti gli autori qui citati, direttamente e indirettamente, per i loro contributi.

L’autore ringrazia anche Dave Brubeck, Larry Carlton, John Coltrane, Paul Desmond, Danny Gatton, Allan Holdsworth, Illinois Jacquet, Waylon Jennings, Eric Johnson, Paul Kossof e i Free, Frank Marino e i Mahogany Rush, Mike Stern, Joey Tafolla, Pat Travers, Carlos Santana, Stevie Ray Vaughan, Johnny Winter, i Deep Purple, gli Uriah Heep nonché Johann Sebastian Bach, Ludwig Van Beethoven, Frédéric Chopin, Wolfgang Amadeus Mozart, Antonio Vivaldi e Richard Wagner per l’immortale e variegata colonna sonora che, accompagnandolo nei suoi viaggi, ha sovente fatto da sottofondo alla stesura del presente articolo.

Riferimenti bibliografici

- [Akl80] Selim G. Akl, *A new algorithm for generating derangements*, BIT **20** (1980), 2–7.
- [BD86] K. Bogart and P. Doyle, *Non-sexist solution of the ménage problem*, American Math. Monthly **93** (1986), no. 7, 514–519.
- [Bon08] B. Bonet, *Efficient algorithm to rank and unrank permutations in lexicographic order*, AAAI - Workshop on Search in AI and Robotics, 2008.
- [Bon12] Miklos Bona, *Combinatorics of permutations*, 2nd ed., Discrete Mathematics and Its Applications, Chapman and Hall/CRC Press, New York, USA, 2012.
- [CLRS09] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed., MIT Press, 2009.

³⁵Più una recente variazione su uno di essi, per un totale di sette algoritmi implementati.

- [Dur64] R. Durstenfeld, *Algorithm 235: Random permutation*, Communications of the ACM **7** (1964), no. 7, 420.
- [Dut86] Jacques Dutka, *On the problème des ménages*, The Mathematical Intelligencer **8** (1986), no. 3, 18–25.
- [ER03] S. Effler and F. Ruskey, *A cat algorithm for generating permutations with a fixed number of inversions*, Information Processing Letters **86** (2003), 107–112.
- [GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete mathematics: A foundation for computer science*, 2nd ed., Addison-Wesley, Reading, MA, 1994.
- [GR10] P. Ganapathi and B. Rama, *A versatile algorithm to generate various combinatorial structures*, Sett. 2010.
- [Hal56] Marshall Hall, Proceedings of Symposia in Applied Mathematics (FL, USA), vol. 6, AMS - American Math. Soc., 1956, p. 203.
- [Hea63] B. R. Heap, *Permutations by interchanges*, The Computer Journal **6** (1963), no. 2, 293–8.
- [HRW12] A. E. Holroyd, F. Ruskey, and A. Williams, *Shorthand universal cycles for permutations*, Algorithmica **64** (2012), no. 2, 215–245.
- [JM81] D. S. Jones and P. G. Moore, *Circular seating arrangements*, Journal of the Institute of Actuaries **108** (1981), 405–411.
- [Kap43] Irving Kaplansky, *Solution of the 'problème des ménages'*, Bulletin of the American Mathematical Society **49** (1943), 784–785.
- [Knu97] Donald E. Knuth, *The art of computer programming, volume 3 (3rd ed.): sorting and searching*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1997.
- [Knu00] ———, *Dancing links*, Nov. 2000.
- [Knu05] ———, *The art of computer programming, volume 4, fascicle 2: Generating all tuples and permutations*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2005.
- [Leh60] D. H. Lehmer, Proceedings of Symposium on Applications of Mathematics to Combinatorial Analisys (FL, USA), vol. 10, AMS - American Math. Soc., 1960, pp. 179–193.
- [Luc91] Edouard Lucas, *Théorie des nombres*, Gauthier-Villars, Paris, 1891, Ristampa Blanchard, 1961.
- [Mak65] David C. Makinson, *The paradox of the preface*, Analysis **25** (1965), no. 6, 205–207.
- [MR01] Wendy Myrvold and Frank Ruskey, *Ranking and unranking permutations in linear time*, Information Processing Letters **79** (2001), no. 6, 281–284.
- [MT14] K. Mikawa and K. Tanaka, *Lexicographic ranking and unranking of derangements in cycle notation*, Discrete Applied Mathematics **166** (2014), 164–169.
- [Par72] David L. Parnas, *On the criteria to be used in decomposing systems into modules*, Communications of the ACM (CACM) **15** (1972), no. 12, 1053–1058.
- [Roh78] J.S. Rohl, *Generating permutations by choosing*, The Computer Journal **21** (1978), no. 4, 302–305.
- [Rus] Frank Ruskey, *Combinatorial generation*, Preliminary working draft. University of Victoria, Victoria, BC, Canada 11, 20.
- [Sat86] S. Sattolo, *An algorithm to generate a random cyclic permutation*, Information Processing Letters **22** (1986), 315–317.
- [Sed77] Robert Sedgewick, *Permutation generation methods*, ACM Computing Surveys (CSUR) **9** (1977), no. 2, 137–164.
- [Sta86] Richard P. Stanley, *Enumerative combinatorics*, vol. 1, Wadsworth Publ. Co., Belmont, CA, 1986.
- [Tho83] D. E. Thomas, *Note*, Journal of the Institute of Actuaries **110** (1983), 396–398.
- [Tou34] Jacques Touchard, *Sur un problème des permutations*, Comptes Rendus de L'Acad. des Sciences (Paris), vol. 198, 1934, pp. 631–633.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

Un problemino con le addizioni...

Sommario

Si parla di partizioni di numeri naturali, cioè dei modi per scrivere un numero dato come somma di altri numeri interi positivi: queste somme finite vengono studiate sistematicamente in matematica da almeno tre secoli, hanno proprietà importanti in Teoria dei Numeri e matematica discreta, e costituiscono inoltre un valido esercizio elementare di programmazione combinatoria. Nel seguito si introduce l'argomento, senza la benché minima pretesa di esaustività e rigore ma con l'esplicito scopo di incuriosire il lettore e presentare un risultato storico eclatante - decisamente controintuitivo e sorprendente per i più - che riguarda le «banali» partizioni. Sono inoltre presentati opportuni esempi di algoritmi combinatori implementati nei linguaggi più idonei e richiami ad innovativi procedimenti di calcolo recentemente apparsi in letteratura, seppure di utilità limitata dal punto di vista strettamente computazionale.

«È più facile quadrare un circolo che arrotondare un Matematico»
(Augustus de Morgan, 1806-1871)

1 Introduzione.

Il materiale qui presentato costituisce una riorganizzazione ed un ampliamento di quanto già discusso sul *blog* dell'autore «Titolo provvisorio...», nella serie di articoli aperta da «Un problemino con le addizioni...». Ne mantiene volutamente inalterati sia il tono colloquiale e didascalico che l'impostazione strettamente divulgativa.

Capita piuttosto sovente sui forum di programmazione di incontrare *thread* creati da qualche utente alle prese con qualche problemino di partizioni numeriche: in varie forme, ciò ricorre abbastanza spesso tra gli esercizi di combinatorica elementare, come anche in numerosi altri campi «insospettabili». Le partizioni sono un concetto fondamentale non solo in teoria del numero, ove esistono da decenni rispettatissime monografie dedicate all'argomento come [And84], ma anche in innumerevoli altri campi, ad esempio:

- Gruppi simmetrici [BMSW54, Com55];
- Polinomi gaussiani [AE04];
- Algebre di Schur e teoria della rappresentazione [Mar07];
- Funzioni ellittiche modulari [Sch86];
- Derivate [Yan00];

...nonché una sterminata lista di applicazioni in fisica matematica e nelle scienze sperimentali. Risulta quindi assolutamente improponibile l'idea di affrontare seriamente l'argomento in un breve articolo divulgativo: tuttavia, vale la pena di dare qualche cenno sulla questione, in modo da suscitare almeno un po' di curiosità e offrire qualche minimo spunto di approfondimento. Si tratta di un problema fortemente rappresentativo dello spirito dell'intera Teoria del Numero, che in generale si occupa di problemi semplici, dall'apparenza elementare ed innocua, i quali però talora celano vette di complessità matematica impressionanti: tanto che normalmente gli approfondimenti in materia non fanno neppure parte dei normali piani di studio nei corsi di laurea, a maggior ragione nelle odierne lauree «brevi».

2 Preliminari e definizioni.

Per gli scopi della presente trattazione introduttiva, può essere accettabile partire dalla seguente definizione semplificata.

Definizione 1. Dato un numero $n \in \mathbb{N}$ non nullo, ossia un intero strettamente positivo¹, chiamiamo *partizione del naturale n* (in k parti) la successione finita non crescente² di naturali $\lambda = \{\lambda_1, \lambda_2, \dots, \lambda_k\}$, propriamente detti *parti*, per la quale vale:

$$\begin{cases} n \geq \lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_k > 0 \\ \lambda_1 + \lambda_2 + \dots + \lambda_k = n \end{cases}$$

Quindi una partizione, in termini intuitivi, è un modo per scrivere il naturale n dato come somma di due o più addendi (al limite, un singolo termine), anch'essi naturali e ovviamente non nulli. Vi sono alcune altrettanto intuitive conseguenze che discendono in modo immediato da questa semplice definizione. Ad esempio:

Lemma 1. Il numero di parti k non sarà in alcun caso superiore al numero dato n , in quanto si ha, al più, $n = \underbrace{1 + 1 + \dots + 1}_{n \text{ volte}}$. Allo stesso modo, esiste una sola possibilità per scrivere un numero n come unico termine, ed è l'identità $n = n$. Per ogni $n > 1$ è unica e distinta dalla precedente anche la scrittura $n = (n - 1) + 1$, il che consente di giungere con facilità ad una intuitiva definizione ricorsiva.

2.1 Generazione ricorsiva di partizioni per parentesizzazione vincolata.

I lettori più computazionalmente smaliziati avranno già iniziato a pensare ad altre proprietà combinatorie di queste «semplici» somme. Prendiamo in considerazione, ad esempio, la partizione in una sola parte del numero 6 e mostriamo un modo intuitivo e ricorsivo nel quale tutte le altre dieci partizioni di tale numero possono essere da essa derivate, tramite *parentesizzazione vincolata*. Tralascio l'enunciazione esplicita delle banali regole del gioco, lasciandole all'intuizione dei miei lettori, e lascio quindi «parlare» l'esempio.

$$\begin{aligned} 6 &= \underbrace{(1 + 1 + 1 + 1 + 1 + 1)}_{6} & (1) \\ 5 + 1 &= \underbrace{(1 + 1 + 1 + 1 + 1)}_{5} + 1 & (2) \\ 4 + 2 &= \underbrace{(1 + 1 + 1 + 1)}_{4} + \underbrace{(1 + 1)}_{2} & (3) \\ 4 + 1 + 1 &= \underbrace{(1 + 1 + 1 + 1)}_{4} + 1 + 1 & (4) \\ 3 + 3 &= \underbrace{(1 + 1 + 1)}_{4} + \underbrace{(1 + 1 + 1)}_{3} & (5) \\ 3 + 2 + 1 &= \underbrace{(1 + 1 + 1)}_{3} + \underbrace{(1 + 1)}_{2} + 1 & (6) \\ 3 + 1 + 1 + 1 &= \underbrace{(1 + 1 + 1)}_{3} + 1 + 1 + 1 & (7) \\ 2 + 2 + 2 &= \underbrace{(1 + 1)}_{3} + \underbrace{(1 + 1)}_{2} + \underbrace{(1 + 1)}_{2} & (8) \\ 2 + 2 + 1 + 1 &= \underbrace{(1 + 1)}_{2} + \underbrace{(1 + 1)}_{2} + 1 + 1 & (9) \\ 2 + 1 + 1 + 1 + 1 &= \underbrace{(1 + 1)}_{2} + 1 + 1 + 1 + 1 & (10) \\ 1 + 1 + 1 + 1 + 1 + 1 &= 1 + 1 + 1 + 1 + 1 + 1 & (11) \end{aligned}$$

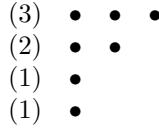
La natura di quest'approccio, squisitamente ricorsiva, invoglia a codificare immediatamente una soluzione pratica nel proprio linguaggio preferito - magari con un generatore Python. Naturalmente l'algoritmica in ambito combinatorio ci ha consegnato anche altri metodi più efficienti e maggiormente adatti alla generazione esaustiva delle partizioni, ma tale esempio così intuitivo mantiene comunque un grande valore didattico, nonché un suo peculiare fascino.

¹Per pura pedanteria, ricordiamo che in altri contesti è utile estendere la definizione ad n non negativo, in modo che valga $n \geq 0$ (nel caso limite esiste comunque una unica partizione, derivata in modo ovvio dall'identità $0 = 0$); allo stesso modo, a volte è utile e comodo considerare la partizione stessa come una serie infinita definitivamente nulla dopo il k -esimo termine. Ma ciò trascende ampiamente gli scopi della nostra chiacchierata.

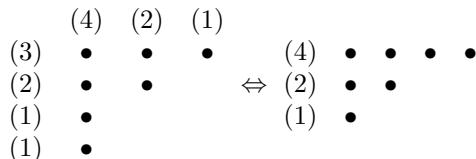
²Sovente gli studenti obiettano, invero un po' ingenuamente, che le partizioni «in quanto somme» non dovrebbero essere caratterizzate da alcun ordine intrinseco particolare, dal momento che l'addizione è commutativa nel rassicurante mondo di cui ci occupiamo per default, ossia il semianello dei naturali. Rimane infatti vero che due partizioni p_1, p_2 a rigore differiscono se e solo se contengono un numero di addendi differenti oppure se almeno uno degli addendi in p_1 differisce da tutti quelli in p_2 , o viceversa, indipendentemente dall'ordine degli addendi. Tuttavia, per definizione, una partizione **non è una somma** ma è costituita dalla mera *successione delle parti*: è in sostanza una lista di numeri naturali (per quel che ci riguarda, strettamente positivi), e come tale possono esservi applicati diversi ordinamenti di default. Nell'ambito della generazione combinatoria (e in quello più generale della matematica discreta) è invalso l'uso di applicare un ordine convenzionale, quello non crescente, come si può riscontrare anche in tutti i testi e gli articoli qui citati.

2.2 Diagrammi di Ferrers e Young.

Sempre restando in tema di rappresentazioni semplici, uno degli strumenti grafici più interessanti ed efficaci sviluppati per trattare la teoria delle partizioni sono i diagrammi di Ferrers e quelli di Young (a quest'ultimi in particolare è dedicata l'interessante monografia [Ful96], alla quale si rimanda caldamente il lettore interessato). In tali diagrammi, semplici e intuitivi, ogni addendo (parte) viene rappresentato con un numero di simboli (tipicamente pallini o quadratini) pari al suo valore, allineati ai vertici di una griglia quadrata rispettando un ordinamento non crescente dall'alto verso il basso. Ecco, ad esempio, una delle partizioni di $7 = 3 + 2 + 1 + 1$ così rappresentata secondo Ferrers:



Usando tali diagrammi risulta naturale e intuitivo definire anche la *partizione coniugata*: è sufficiente scambiare righe e colonne, ossia «contare i pallini» per colonne, anziché per righe, e riscrivere il nuovo diagramma con tale ordine:



In questo modo è immediato, ad esempio, far intuire al lettore la dimostrazione di un fondamentale teorema: ogni partizione di n il cui elemento massimo sia m (in questo caso, 3) corrisponde ad una partizione del medesimo numero n che abbia esattamente m parti; il che appare ovvio contando le «colonne» di una partizione, e quindi le «righe» della sua coniugata, in rappresentazione Ferrers o Young.

3 Generare tutte le partizioni di n .

Una volta fornita almeno una ectoplasmatica idea della inherente (ma solo apparente!) semplicità della nozione di partizione di un numero naturale, nessun lettore si stupirà nell'apprendere che praticamente tutti i testi di algoritmica combinatoria trattano l'argomento «generazione delle partizioni di un naturale» nelle primissime pagine, tra gli oggetti combinatori elementari.

L'esercizio di generare tutte le partizioni di n è in effetti fin troppo banale. Tutti i vari testi citati in bibliografia (es. [Rei77, Nij78, PW79, Sta86b, Knu05, Rus, Ski09]) contengono algoritmi e/o esempi di codice per la generazione esaustiva e vincolata di partizioni (es. partizioni con valore massimo k , ovvero con esattamente k parti). Sono facilmente disponibili ottime ed efficienti soluzioni in C (alcune anche in C++, come quelle proposte da J. Arndt in [Arn10]) dalle quali il lettore potrà apprendere molto. Invito anzi caldamente tutti i lettori a familiarizzare con il codice sovente disponibile nelle varie homepages correlate, e ricordo che buona parte dei testi citati è oggi liberamente disponibile per il download presso i siti dei rispettivi autori, a volte con blande limitazioni (del tutto irrilevanti ai fini dello studio individuale).

Parlando di Python, una soluzione ricorsiva piuttosto elegante (sebbene dalle prestazioni limitate) viene dagli archivi delle "recipes" di Python sul sito ActiveState ed è dovuta a David Eppstein:

```
def partitions(n):
    # base case of recursion: zero is the sum of the empty list
    if n == 0:
        yield []
        return

    # modify partitions of n-1 to form partitions of n
    for p in partitions(n-1):
        yield [1] + p
        if p and (len(p) < 2 or p[1] > p[0]):
            yield [p[0] + 1] + p[1:]
```

Ma naturalmente le soluzioni più eleganti in assoluto sono quelle offerte dai paradigmi non convenzionali. Ecco una splendida soluzione in J, derivata dall'articolo [Bos08], liberamente disponibile in formato PDF.

```

parts=.monad define ([] ,~ >:@i .@[ <@;@:(([ ,.( >:{."1})#])&.>)
{.)&.>/(<<i .1 0),~<"0 ([] ( -<.>:@) i .) @<:) y )

```

Una volta superata l'impressione che tale apparentemente insensata sequenza di caratteri sia il risultato della pigra passeggiata di un micio sulla tastiera, è facile convincersi - magari anche compulsando con attenzione l'articolo relativo - che si tratta di una delle soluzioni più eleganti in assoluto per la generazione esaustiva di partizioni, peraltro anche ragionevolmente efficiente. Segnalo inoltre questo articolo, di facile lettura, che compara i più noti algoritmi di generazione delle partizioni e le relative implementazioni in J.

3.1 Due algoritmi generatori altamente efficienti.

Il primo algoritmo di generazione esaustiva delle partizioni che proponiamo è stato pubblicato da Zoghbi e Stojmenovic nel 1994 [ZS94] ed è tuttora uno dei più efficienti noti in letteratura per l'uso generale. Come si può immediatamente notare, si tratta di un algoritmo iterativo estremamente semplice che esegue in tempo costante ammortizzato (CAT) e genera tutte le partizioni del naturale n in ordine lessicografico inverso. L'algoritmo fa uso unicamente di variabili di induzione e contatori, oltre all'array destinato a contenere la partizione corrente, e consta sostanzialmente solo di assegnazioni, incrementi, decrementi. Si noti come la partizione in una singola parte, ossia n , viene stampata prima del loop principale.

Algoritmo 1. *AccelDesc di Zoghbi e Stojmenovic.*

```

PROCEDURE AccelDesc
Require:  $n \geq 1$ 
 $k \leftarrow 1$ 
 $q \leftarrow 1$ 
 $d_2 \dots d_n \leftarrow 1 \dots 1$ 
 $d_1 \leftarrow n$ 
visit  $d_1$ 
while  $q \neq 0$  do
    if  $d_q = 2$  then
         $k \leftarrow k + 1$ 
         $d_q \leftarrow 1$ 
         $q \leftarrow q - 1$ 
    else
         $m \leftarrow d_q - 1$ 
         $n' \leftarrow k - q + 1$ 
         $d_q \leftarrow m$ 
        while  $n' \geq m$  do
             $q \leftarrow q + 1$ 
             $d_q \leftarrow m$ 
             $n' \leftarrow n' - m$ 
        end while
        if  $n' = 0$  then
             $k = q$ 
        else
             $k \leftarrow q + 1$ 
            if  $n' > 1$  then
                 $q \leftarrow q + 1$ 
                 $d_q \leftarrow n'$ 
            end if
        end if
    end if
    visit  $d_1 \dots d_k$ 
end while

```

Il secondo algoritmo generatore preso in considerazione per le sue peculiarità è uno dei più recenti in assoluto: la seconda revisione dell'articolo di Kelleher e O'Sullivan [KO09] è stata infatti pubblicata nel 2014. L'algoritmo

è simile al precedente, ma ha la particolarità di generare le partizioni in ordine opposto rispetto agli algoritmi classici, sovertendo l'usuale definizione (1) come elenco di parti non crescenti e l'ordine lessicografico utilizzato dalla maggioranza degli algoritmi. Gli autori ritengono di ravvisare nel ricorso a tale inusuale ordinamento un leggero vantaggio teorico rispetto all'algoritmo (1).

Algoritmo 2. *AccelAsc di Kelleher e O'Sullivan.*

```

PROCEDURE AccelAsc
Require:  $n \geq 1$ 
 $k \leftarrow 2$ 
 $a_1 \leftarrow 0$ 
 $y \leftarrow n - 1$ 
while  $k \neq 1$  do
     $k \leftarrow k - 1$ 
     $x \leftarrow a_k + 1$ 
    while  $2x \leq y$  do
         $a_k \leftarrow x$ 
         $y \leftarrow y - x$ 
         $k \leftarrow k + 1$ 
    end while
     $\ell \leftarrow k + 1$ 
    while  $x \leq y$  do
         $a_k \leftarrow x$ 
         $a_\ell \leftarrow y$ 
        visit  $a_1 \dots a_\ell$ 
         $x \leftarrow x + 1$ 
         $y \leftarrow y - 1$ 
    end while
     $y \leftarrow y + x - 1$ 
     $a_k \leftarrow y + 1$ 
    visit  $a_1 \dots a_k$ 
end while

```

3.2 L'implementazione in COMAL 80.

L'inerente semplicità degli algoritmi (1) e (2) invoglia ad una (retro)implementazione in COMAL 80 (collaudata con la vers. 2.01 su cartridge per Commodore 64), quantomai lineare e divertente. Il lettore in vena di sperimentazioni potrebbe voler considerare l'aggiunta di statement come **SELECT OUTPUT "lp:"** per inviare l'output su stampante (virtuale) e dovrà comunque armarsi di pazienza: il programma, pur essendo in grado di generare senza errori tutte le 8.349 partizioni di $n = 32$, richiede tuttavia circa mezz'ora per portare a termine tale elaborazione su un Commodore 64...

```

PAGE
PRINT "*****"
PRINT "** EnumParts **"
PRINT "*****"
DIM part(32)
DIM table'parts(33)
total:=1
INPUT "Immetti un valore (3-32): ": n
num'parts(n)
PRINT "P(",n,") = ",table'parts(n+1)
acceldesc(n)
accelasc(n)
END "Fine lavoro."
// ****

```

```

//** Subroutines
//***** ****
PROC acceldesc(n)
//***** ****
//** Algorithm AccelDesc by
//** Zoghbi & Stojmenovic 1994
//***** ****
PRINT "AccelDesc(" ,n ,")"
IF n<3 THEN
    PRINT "n deve essere maggiore di 3"
    RETURN
ENDIF
total:=1
i:=1
j:=1
FOR i:=2 TO n DO
    part(i):=1
ENDFOR i
part(1):=n
display 'part(part(),1)
WHILE j<>0 DO
    IF part(j)=2 THEN
        i:=i+1
        part(j):=1
        j:=j-1
    ELSE
        part(j):=-1
        m:=part(j)
        k:=i-j+1
        WHILE k>=m DO
            j:=j+1
            part(j):=m
            k:=k-1
        ENDWHILE
        IF k=0 THEN
            i:=j
        ELSE
            i:=j+1
            IF k>1 THEN
                j:=j+1
                part(j):=k
            ENDIF
        ENDIF
    ENDIF
    display 'part(part(),i)
ENDWHILE
ENDPROC acceldesc

PROC accelasc(n)
//***** ****
//** Algorithm AccelAsc by
//** Kelleher e O'Sullivan 2014
//***** ****
PRINT "AccelAsc(" ,n ,")"
IF n<3 THEN
    PRINT "n deve essere maggiore di 3"
    RETURN
ENDIF
total:=1
i:=2

```

```

j:=n-1
part(1):=0
WHILE i<>1 DO
    i:=1
    k:=part(i)+1
    WHILE 2*k<=j DO
        part(i):=k
        j:=k
        i:=i+1
    ENDWHILE
    m:=i+1
    WHILE k<=j DO
        part(i):=k
        part(m):=j
        display 'part(part(),m)
        k:=k+1
        j:=j-1
    ENDWHILE
    j:=j+k-1
    part(i):=j+1
    display 'part(part(),i)
ENDWHILE
ENDPROC accelasc

PROC num' parts(m)
//*****
//** Calculta la funzione P(m)
//*****
table'parts(1):=1
table'parts(2):=1
FOR i:=3 TO m+1 DO
    sum:=0
    sign:=1
    omega:=2
    j:=1
    k:=1
    WHILE omega<=i DO
        omega2:=omega+j
        sum:=sum+sign*table'parts(i-omega+1)
        IF omega2<=i THEN
            sum:=sum+sign*table'parts(i-omega2+1)
        ENDIF
        j:=j+1
        k:=k+3
        omega:=omega+k
        sign:=-sign
    ENDWHILE
    table'parts(i):=sum
ENDFOR i
ENDPROC num' parts

PROC display'part(a(),k)
PRINT USING "#####": total,
PRINT ":" ,
total:=1
FOR i:=1 TO k DO
    PRINT USING "###": a(i),
ENDFOR i
PRINT ""
ENDPROC display'part

```

Il listato, a prescindere da qualche insignificante modifica nella nomenclatura delle variabili di induzione, è sostanzialmente autoesplicativo e ricalca fedelmente i semplicissimi passaggi illustrati nei due algoritmi sopra presentati, con l'aggiunta di un terzo algoritmo che vedremo meglio nella prossima sezione 4. L'offset di una unità nell'indicizzazione dell'array `table'parts()` è dovuto ovviamente al fatto che in COMAL gli indici partono da 1, per cui si ha $P(n)=table'parts(n+1)$.

3.3 Implementazione in C99.

Si riporta per confronto una analoga implementazione pediseguamente didattica in C99, senza dilungarci in ulteriori commenti o considerazioni. Si suggerisce solo che, volendo produrre un ulteriore esempio implementativo in Python, la soluzione migliore consiste nell'utilizzare direttamente Sympy: una consolidata libreria di calcolo simbolico che compendia anche numerosi generatori combinatori, tra i quali un valido generatore di partizioni.

```
#include <stdio.h>
#include <stdlib.h>

#define PART_SIZE 128

// Contatore globale delle partizioni generate
size_t cnt;

/*********************************************************/
void DisplayPart(unsigned int c[], size_t s) {
    size_t i;

    printf("%5u: ", ++cnt);

    for (i = 1; i <= s; ++i) {
        printf("%3u", c[i]);
    }

    puts("");
}

/*********************************************************/
void AccelDesc(unsigned int n) {
    size_t i, j, k, m;
    unsigned int part[PART_SIZE];

    cnt = 0L;

    for (i = 2; i <= n; ++i) {
        part[i] = 1;
    }

    i = 1;
    j = 1;
    part[1] = n;

    printf("\n** AccelDesc(%u)\n", n);

    DisplayPart(part, 1);

    while (j != 0) {
        if (part[j] == 2) {
            part[j] = 1;
            --j;
            ++i;
        } else {
            part[j] -= 1;
            m = part[j];
            k = i - j + 1;
```

```

while ( k >= m) {
    part[++j] = m;
    k -= m;
}

if (k == 0) {
    i = j ;
} else {
    i = j +1;

    if (k > 1) {
        part[++j] = k;
    }
}
DisplayPart( part , i );
}

<*/*****/>
void AccelAsc(unsigned int n) {
    size_t i , j , k , m;
    unsigned int part[PART_SIZE];

    cnt = 0L;

    i = 2;
    j = n -1;
    part [1] = 0;

    printf ("\n** AccelAsc(%u)\n" , n);

    while ( i != 1) {
        --i ;
        k = part [ i ] + 1;

        while (k<<1 <= j ) {
            part [ i ] = k;
            j == k;
            ++i ;
        }

        m = i +1;

        while (k <= j ) {
            part [ i ] = k;
            part [m] = j ;
            DisplayPart( part , m);
            ++k;
            --j ;
        }

        j += k -1;
        part [ i ] = j +1;
        DisplayPart( part , i );
    }
}

<*/*****/>
int main(void) {
    size_t i ;
}

```

```

const unsigned test_vect[] = {5, 12, 32};

for (i = 0; i < 3; ++i) {
    AccelDesc(test_vect[i]);
    AccelAsc(test_vect[i]);
}

return (EXIT_SUCCESS);
}
/* EOF: AccAsc.c
 */
/*
 */

```

3.4 Un ulteriore algoritmo (e una simpatica digressione).



Figura 1: 964° «Quesito con la Susi» pubblicato sulla rivista N° 4577 del 12 dicembre 2019 come 4513° concorso settimanale.

Le partizioni di interi (come anche quelle di insiemi...) sono davvero dovunque, anche dove meno ce le aspettiamo. Tra le innumerevoli possibilità, sceglieremo una testimonial d'eccezione: la bionda Susi, instancabile propalatrice di enigmi logico-numericci della quasi novantenne Settimana Enigmistica. Si tratta di ottimo pretesto per implementare in linguaggio C uno degli algoritmi più belli ed efficienti per la generazione di partizioni vincolate, risalente addirittura a Carl Friedrich Hindenburg (1741-1808) e illustrato dal venerabile Knuth nell'arcinoto TAoCP, volume 4 [Knu05]: l'algoritmo H, in questo caso un «analogo con modifica» per limitarsi alle partizioni con esattamente 3 elementi e valore massimo pari a 9. Per l'esattezza, la scelta è ricaduta sul 964° «quesito con la Susi», come visibile in fig. 1.

Nel seguito useremo per mero gusto personale lettere minuscole in luogo delle maiuscole impiegate dal vignettista, sottintendendo una implicita corrispondenza dei simboli. Risulta immediato rilevare che le lettere a, b, c così disposte formano un *quadrato latino*: in particolare, uno dei dodici possibili di ordine 3. Infatti ognuno dei tre simboli appare esattamente una volta per ogni riga e per ogni colonna.

a	b	c
c	a	b
b	c	a

Tra le meravigliose caratteristiche di queste matrici c'è anche quella di avere *somma costante* per colonne e per righe: in questo caso, la cosiddetta *costante magica* è data simbolicamente da $s = a + b + c$. Ne consegue che il valore della somma a quattro cifre dei tre numeri, ricordando dalle scuole elementari come si addizionano i numeretti in colonna per unità, decine, centinaia etc., sarà dato da $x = 100s + 10s + s$. Raccogliendo i termini omogenei, si ha quindi in definitiva che la somma dei tre numeri di tre cifre incogniti è pari a:

$$x = (100 + 10 + 1) \cdot s = 111 \cdot s = 111 \cdot (a + b + c) \quad (1)$$

Il problema si riduce pertanto a cercare tra alcuni multipli a quattro cifre di 111 quello composto da tutte cifre distinte. Tali multipli saranno generati da un moltiplicatore s compreso tra³ 11 e rispettivamente 24 o 27, secondo che vogliamo considerare a, b e c necessariamente *distinti* $a \neq b \neq c$ o meno (il testo non lo specifica in modo chiaro ed esplicito, ricorre solo implicitamente alla classica formula enigmistica "a simbolo uguale corrisponde numero uguale"). Si tratta quindi di effettuare una manciata di operazioni elementari alla ricerca di un numero (l'unico in quell'intervallo con le caratteristiche desiderate), per scoprire banalmente che $s = 19$, da cui il valore incognito cercato $x = 2109 = 111 \cdot 19$.

³Inutile perdere tempo a cercare cifre distinte nel risultato di $111 \cdot 10$, anche se a rigore il limite inferiore sarebbe dato da $10 = [1000 \div 111]$.

Ovviamente fino qui abbiamo lavorato unicamente in *pencil-and-paper mode*, o se si preferisce *back of envelope*, anche se l'algoritmo ricavato è in pratica immediatamente automatizzabile. Il fatto è che la somma delle cifre desiderata, $s = 19$, si può ottenere banalmente assegnando in vari modi diversi i valori nell'intervallo naturale $[1, 9]$ ad a, b e c . In altri termini, non è univoco il modo per ottenere quel totale di 2109. Si noti che escludiamo la presenza di cifre nulle per il banale motivo che $19 > 9 + 9$, quindi anche nel caso limite la minore delle cifre sarà necessariamente non inferiore all'unità. Ecco allora che le partizioni entrano prepotentemente in gioco nel momento in cui una mente matematica cerca di rispondere alla domanda: *in quanti modi è possibile assegnare le singole cifre tra 1 e 9 ad a, b e c per comporre quella costante 19 e ottenere l'agognata somma?*

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX_PARTS 16
#define FMT_PERMS "\n    %02d: "

/*
** Totalizzatori per le soluzioni:
** Partizioni, Partizioni in parti distinte, Permutazioni delle
** parti su a-b-c.
*/
unsigned Cnt_parts, Cnt_unique, Cnt_perms;

/*
** Tabella precalcolata delle permutazioni di tre elementi
*/
unsigned perms[6][3] = {{0,1,2},{0,2,1},{1,0,2},
                       {1,2,0},{2,0,1},{2,1,0}};

/*
** Tabella precalcolata delle permutazioni di un multiset
** con occorrenze {1,2}
*/
unsigned mperms[3][3] = {{0,1,2},{1,0,2},{1,2,0}};

/*****************/
/*
** Stampa la partizione corrente e le relative permutazioni su a,b,c.
*/
/*****************/
void Visit(unsigned a[], const unsigned m) {
    size_t i, j;

    Cnt_parts++;
    printf("%02d: %d ", Cnt_parts, a[0]);

    for (i = 1; i < m; ++i) {
        printf("+ %d ", a[i]);
    }

    /*
    ** Il caso di partizione in tre elementi identici non e'
    ** consentito dai vincoli del problema. Si esaminano i
    ** restanti tre casi.
    */
    if (a[0] == a[1]) {
        /* Partizione di tipo j+j+k, j>k */
        for (i = 0; i < 3; ++i) {
            Cnt_perms++;
            printf(FMT_PERMS, Cnt_perms);
        }
    }
}
```

```

        for (j = 0; j < 3; ++j) {
            printf("%c=%d ", 'a'+j, a[2-mperms[i][j]]);
        }
    } else if (a[1] == a[2]) {
        /* Partizione di tipo  $j+k+k$ ,  $j>k$  */
        for (i = 0; i < 3; ++i) {
            Cnt_perms++;
            printf(FMT_PERMS, Cnt_perms);

            for (j = 0; j < 3; ++j) {
                printf("%c=%d ", 'a'+j, a[mperms[i][j]]);
            }
        } else {
            /* Partizione in tre elementi distinti */
            Cnt_unique++;

            for (i = 0; i < 6; ++i) {
                Cnt_perms++;
                printf(FMT_PERMS, Cnt_perms);

                for (j = 0; j < 3; ++j) {
                    printf("%c=%d ", 'a'+j, a[perms[i][j]]);
                }
            }
        }
    puts(" ");
}
//****************************************************************************
/*
** Genera tutte le partizioni di n in esattamente m parti, con valore
** massimo non superiore ad u.
**
** Basato sull'algoritmo H dal volume 4 del TAOCP di Knuth.
*/
//****************************************************************************
void GenParts(const unsigned n, const unsigned m, const unsigned u) {
    size_t j, s, x;
    unsigned a[MAX_PARTS];

H1: /* [Initialize.] */
    a[0] = n -m +1;
    a[m] = m -1;

    for (j = 1; j < m; ++j) {
        a[j] = 1;
    }

H2: /* [Visit.] */
    if (a[0] > u) {
        goto H20;
    }

    Visit(a, m);
H20:
    if (a[1] > (a[0] - 2)) {
        goto H4;
    }
}

```

```

    }

H3: /* [Tweak a[1] and a[2].] */
    a[0] == 1;
    a[1] += 1;
    goto H2;
H4: /* [Find j.] */
    j = 2;
    s = a[0] + a[1] -1;

    while (a[j] > (a[0] -2)) {
        s += a[j];
        ++j;
    }

H5: /* [Increase a.] */

    if (j > m-1) {
        return;
    } else {
        x = a[j] +1;
        a[j] = x;
        --j;
    }

H6: /* [Tweak a[1]...a[j].] */

    while (j) {
        a[j] = x;
        s -= x;
        --j;
    }

    a[0] = s;
    goto H2;
}

/*
** Risolve il 964° "Quesito con la Susi", pubblicato sulla rivista
** "La settimana enigmistica" N° 4577 del 12 dicembre 2019 come 4513°
** concorso settimanale.
** Calcola inoltre tutte le possibili attribuzioni di valori ai simboli
** a, b e c usati nella vignetta per caratterizzare le singole cifre dei
** numeri incogniti da sommare.
*/
int main(void) {
    const unsigned mul = 111;
    unsigned s, p;
    bool unique[10];
    char val[10];

    Cnt_parts = 0;
    Cnt_perms = 0;
    Cnt_unique = 0;

    for (s = 11; s < 28; ++s) {
        size_t i;
        bool valid = true;

        p = mul * s;

```

```

itoa(p, val, 10);

for (i = 0; i < 10; ++i) {
    unique[i] = false;
}

for (i = 0; i < 4; ++i) {
    if (unique[val[i] - '0']) {
        valid = false;
        break;
    } else {
        unique[val[i] - '0'] = true;
    }
}

if (valid) {
    printf(">> Totale: %d = %d x %d\n\n"
           "> Elenco delle partizioni di %d in 3 parti:\n",
           p, mul, s, s);
    GenParts(s, 3, 9);
    printf("*****\n"
           "Esistono %d diverse partizioni del numero %d che "
           "risolvono il problema,\n"
           "un totale di %d distinte attribuzioni di valori "
           "ai>> simboli a, b e c. Di queste partizioni, "
           "%d sono semplici ossia constano\n di parti tutte "
           "distinte.",
           Cnt_parts, s, Cnt_perms, Cnt_unique);
    break;
}
}

return (0);
}
/** EOF: susy77.c ***/

```

La banalissima modifica introdotta nel codice di Knuth consiste nell'uso di un terzo parametro e nella sbrigativa implementazione di un *late filtering* per visualizzare le sole partizioni la cui parte massima è non superiore a 9, come da vincoli del problema. Quasi inutile rimarcare che tale soluzione non costituisce certo la scelta più efficiente per una generazione vincolata, pur essendo banalissima da implementare: in una applicazione *real world* sarebbe necessario sforzarsi di filtrare le cifre maggiori di 9 il più presto possibile durante la fase di calcolo e spostamento. Una volta generata ciascuna partizione in formato standard (non crescente), vi sono poi per ognuna *sei distinti modi* di assegnare i valori ad a, b e c , per le partizioni semplici (in tre parti distinte), oppure *tre modi* se vi sono due parti di identico valore (altri casi non sono contemplati). Il codice qui proposto in C99 è estremamente ridondante e semplificato, la logica di gestione è del tutto elementare, anche nella discriminazione delle tipologie di partizione (sarebbe banale renderlo più compatto e parametrizzato, ma anche molto meno leggibile), e sostanzialmente autodocumentante. Si usano sbrigativamente delle tabelle precalcolate per generare le permutazioni, per la massima efficienza e perché non è questo il focus del codice didattico. Si fa ovviamente uso di un array di booleani per controllare efficientemente l'unicità delle quattro cifre nel risultato della moltiplicazione che risolve il problema posto: a ciascuna locazione dell'array corrisponde una delle cifre tra 0 e 9, e la relativa cella contiene il valore della *funzione caratteristica* applicato a tale cifra.

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

$$f_{\mathcal{S}}(a) := \begin{cases} 1 & \text{se } a \in \mathcal{S} \\ 0 & \text{se } a \notin \mathcal{S} \end{cases} \quad (2)$$

In letteratura la funzione caratteristica è sovente indicata anche come $1_{\mathcal{S}}$, $I_{\mathcal{S}}$ o $\chi_{\mathcal{S}}$. Si noti per inciso che la cardinalità del sottoinsieme \mathcal{S} è data dalla somma dei valori delle sua funzione caratteristica:

$$|\mathcal{S}| = \sum_{a \in \mathcal{A}} f_{\mathcal{S}}(a) \quad (3)$$

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*, che in questo caso sfruttiamo per determinare se una singola cifra presa dall'insieme $\mathcal{A} = \{0, 1, \dots, 9\}$ compare o meno nel numero generato (il sottoinsieme \mathcal{S} considerato), potendo così controllare in $O(1)$ se tale cifra è effettivamente unica. Il report finale prodotto al termine del `main()` è estremamente esplicativo e dettagliato.

4 Il numero di partizioni di un naturale $P(n)$.

Ci accordiamo esplicitamente per indicare con $P(n)$, come quasi universalmente in letteratura, il numero totale di partizioni del naturale n (la cosiddetta *funzione di partizione*), e con $P(n, k)$ uno specifico numero di partizioni di n in esattamente k parti, laddove $1 \leq k \leq n$. Vale allora la semplice relazione di ricorrenza:

$$\begin{cases} P(n, k) = P(n - 1, k - 1) + P(n - k, k) \\ P(n, 1) = P(n, n) = 1 \end{cases} \quad (4)$$

Qui riprendiamo appieno il tono didascalico e confidenziale a suo tempo utilizzato nel blog per avvicinarci, con un sorriso, all'aspetto più nerd-folkloristico della questione. Sì, va bene: generare le partizioni di un numero naturale è in effetti un lavoretto banale, ma si riesce a sapere quante sono codeste partizioni senza enumerarle tutte con una ricorrenza additiva? Così sembrano invariabilmente ragionare i giovani programmati che periodicamente si presentano sui forum: un po' stupiti e un po' contrariati perché, in uno scenario così semplice, il loro libro preferito «sembra aver dimenticato di menzionare la formuletta per calcolare quante sono in totale le partizioni $P(n)$ di un numero n dato». Probabilmente, aggiungono, perché deve essere una formula «talmente puerile» da esser lasciata come *semplice esercizio per il lettore interessato*, anche se «stranamente» non sono riusciti a ricavarla da soli dopo vari tentativi (a volte anche ingegnosi, a onor del vero). Le cose non vanno diversamente se si apre uno dei tanti testi di matematica discreta, ad esempio il bel lavoro [MZ97] che pure all'argomento dedica un intero capitulo, trattando anche temi relativamente avanzati, come i reticolati di dominanza e quelli di Young. Se sul ben noto Kreher-Stinson si giunge addirittura a scrivere che «*Although partitions have been studied by mathematicians for hundreds of years and many interesting results are known, there is no known formula for the values of $P(m)$.*» ([KS98], pag. 67), gli altri testi se la cavano con una più dignitosa omertà, dopo avere frettolosamente ricordato che $P(n)$ cresce in modo estremamente veloce; oppure si limitano a presentare tabelline varie, come la seguente nella quale $P(n)$ rappresenta il numero totale di partizioni, $q(n)$ il numero di partizioni semplici ovvero in parti tutte distinte:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$P(n)$	1	2	3	5	7	11	15	22	30	42	56	77	101	135
$q(n)$	1	1	2	2	3	4	5	6	8	10	12	15	18	22

(5)

A questo punto, messo di fronte - libri alla mano - a cotanta generale evidenza, di solito il giovane programmatore sbotta impaziente: «Ma insomma, che avrà mai di strano 'sta dannata formuletta, che tutti se la nascondono nella manica? Per tirar fuori quei quattro numeretti, non sarà mica così complicata... insomma... voglio dire, se esiste la formuletta di Binet per trovare l'ennesimo numero di Fibonacci in $O(1)$, ci sarà ben qualcosa anche per questo caso... o no?!?!?!»

4.1 La formula di Hardy-Ramanujan-Rademacher.

Ebbene, la risposta è che una formula per determinare il numero di partizioni $P(n)$ di un naturale dato n in effetti esiste. Diciamo però che non è una formula chiusa e che è *un tantino complicata*. A dire il vero, è opinione popolare che si tratti di una delle formule più mostruosamente cervellotiche, astruse e macchinose dell'intera matematica pura... La stima asintotica originariamente trovata da Godfrey H. Hardy (1877-1947) e Srinivasa Ramanujan (1887-1920) un secolo fa aveva già un aspetto «poco rassicurante»:

$$P(n) \underset{n \rightarrow \infty}{\cong} \frac{1}{4n\sqrt{3}} \cdot e^{\left(\pi\sqrt{\frac{2n}{3}}\right)} \quad (6)$$

Ma questo è nulla. Il numero di partizioni di un naturale n è dato dalla valutazione parziale (con approssimazione all'intero più vicino) della serie asintotica convergente che segue:

$$P(n) = \frac{1}{\pi\sqrt{2}} \sum_{k=1}^{\infty} A_k(n) \sqrt{k} \frac{d}{dx} \left[\frac{\sinh\left(\frac{\pi}{k}\sqrt{\frac{2}{3}}(x - \frac{1}{24})\right)}{\sqrt{(x - \frac{1}{24})}} \right]_{x=n} \quad (7)$$

dove

$$A_k(n) = \sum_{h=1}^k \delta[MCD(h, k), 1] \exp\left[i\pi\left(s(h, k) - \frac{2nh}{k}\right)\right]$$

Nella definizione qui sopra compare anche la familiare funzione discreta Delta di Kronecker, poiché contribuiscono al totale solo quei termini nei quali h e k sono primi tra loro (coprimi), il che è verificato quando il loro massimo comun divisore è pari a uno:

$$\delta(a, b) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{se } a \neq b \\ 1 & \text{se } a = b \end{cases} \quad (8)$$

Inoltre, si definisce come segue la somma di Dedekind:

$$s(h, k) = \sum_{j=1}^{k-1} \frac{j}{k} \left(\frac{hj}{k} - \left\lfloor \frac{hj}{k} \right\rfloor - \frac{1}{2} \right) \quad (9)$$

La notazione è conforme a quella usata su Wolfram MathWorld, in [AW95] e nella maggioranza delle pubblicazioni moderne⁴. Il venerabile Donald E. Knuth, che all'argomento dedica le prime 25 pagine nel fascicolo 3b del quarto volume [Knu05], fa invece uso di una notazione leggermente diversa, peraltro con l'uso della funzione sferica di Bessel modificata (si vedano ad esempio [Leb72, Bow03]), rigorosamente coerente con la sua esposizione dello sviluppo storico della formula e di altri risultati correlati.

La stupefacente formula, alla quale tra l'altro si fa cenno anche in lavori divulgativi di largo successo come [dS04], è dovuta come accennavo ai teorici dei numeri Hardy e Ramanujan, con sostanziali migliorie a posteriori da parte di Hans Rademacher (1892-1969), e porta quindi il nome di tutti e tre. Le pubblicazioni originali sono la [HR18] e la [Rad38], rispettivamente.

Orbene, a questo punto il lettore può facilmente immaginare l'espressione del nostro apoftegmatico giovane programmatore: giunto sul forum ansioso di abbellire il suo programmino col calcolo del numero di partizioni del naturale dato in input, si trova di fronte a questo «spaventoso formulone». Più di qualcuno ha osservato letteralmente che invece di una «vera formula», di quelle che «si studiano a scuola», questa sembra piuttosto una di quelle apocalittiche stupidaggini matematiche che gli scenografi fanno copiare a qualche scagnozzo sulle finte lavagne dei finti laboratori del solito improbabile scienziato pazzo cinematografico (per chi fosse interessato, c'è in rete un'ampia quanto oziosa letteratura su formule mal copiate e intere lavagnate di *nonsense* apparse a vario titolo in pellicole più o meno famose).

In effetti è una formula *sbalorditiva* secondo la qualificata opinione della maggioranza dei matematici, computazionali e non: Knuth la definisce senza mezzi termini «...surely one of the most astonishing identities ever discovered» ([Knu05], fasc. 3b, pag. 8). Sembra davvero provenire da un'altra galassia, per quanto è sproporzionalmente distante dalla banalissima natura di questo umile problemino di addizioni: una sfilza di radicali, pigreco, l'unità immaginaria, funzioni trascendenti, differenziali, operatore *floor*, una delta discreta applicata ad un massimo comun divisore e una storia d'implicazioni che rimanda facilmente a cerchi di Ford, serie di Farey, funzione sferica di Bessel...

4.1.1 Una formula «soddisfacente»?

Dopo avere brevemente visto la formula di Hardy-Ramanujan-Rademacher (7) per determinare il numero complessivo di partizioni di un naturale n , è forse opportuno un brevissimo *excursus* con l'autorevole compagnia del professor Richard P. Stanley del MIT. Cercheremo di tradurre e condensare in poche righe l'intera filosofia che è magistralmente riassunta nelle prime pagine del suo capolavoro [Sta86a], che non ha alcun bisogno di presentazioni o di indici di citazione (a meno che non vogliamo indulgere nel manipolare numeri a cinque cifre, beninteso).

Quella formula strabiliante, al di là dell'ammirazione e dello stupore, ci trova anche riluttanti in veste di matematici computazionali: proviamo a capirne il perché. Stanley porta quattro esempi fondamentali per spiegare meglio cosa si possa realmente considerare una *formula soddisfacente*, in forma chiusa ed esplicita:

1. L'insieme delle parti (o insieme potenza) di un insieme finito con cardinalità espressa dal naturale n , cioè l'insieme di tutti i possibili sottoinsiemi che si possono formare dall'insieme di partenza, ha un numero totale di elementi dato da $\phi(n) = 2^n$...e tutti, profani inclusi, sono d'accordo nel considerarla una formula perfettamente soddisfacente!

⁴Solo una pennellata di pedanteria formale. L'uso della notazione funzionale $\exp(x)$ per l'esponenziale è una scelta puramente dettata da opportunità pratica, perché ci consente di rendere molto più leggibili esponenti lunghi e complicati, senza forzature di font: $\exp(x) \stackrel{\text{def}}{=} e^x$.

2. Si supponga che n signori consegnino i loro n cappelli ad una guardarobiera. Sia $f(n)$ il totale dei modi in cui costoro possono riavere indietro quei cappelli, facendo però sì che nessuno riceva il proprio cappello, quello cioè che aveva in testa al suo arrivo. L'espressione della nostra $f(n)$, in questo caso, è data da:

$$f(n) = n! \sum_{k=0}^n \frac{(-1)^k}{k!}$$

Tale formula, sebbene marcatamente meno elegante della precedente, è comunque accettabile «in mancanza di una risposta più semplice», anche perché è in grado di semplificare (in un senso ben quantificabile) il calcolo dei valori di $f(n)$. Inoltre, la sua aderenza ad un importante principio (quello detto di **inclusione/esclusione**) la rende comunque «intuitiva» a chi abbia un minimo di familiarità con la materia.

3. Qui Stanley invita a considerare come $f(n)$ il numero di matrici booleane quadrate $n \times n$ tali che ciascuna riga e colonna contiene esattamente tre valori pari a 1. La formula migliore nota ad oggi è:

$$f(n) = \frac{(n!)^2}{6^n} \sum \frac{(-1)^{\beta}(\beta + 3\gamma)!2^{\alpha}3^{\beta}}{\alpha!\beta!(\gamma!)^26^{\gamma}}$$

Dove la sommatoria copre tutte le soluzioni naturali all'equazione $\alpha + \beta + \gamma = n$, ossia (lupus in fabula!) le partizioni di n in esattamente tre parti, che in totale assommano a $\frac{(n+2)(n+1)}{2}$. L'unico motivo per accettare (con riluttanza) una simile formula, scrive Stanley, è di natura computazionale - sebbene essa non ci dica quasi alcunché di significativo riguardo alla natura della funzione studiata.

4. Ultimo e peggiore esempio: esistono in letteratura «funzioni» di conteggio la cui valutazione richiede nulla meno che un'enumerazione esaustiva (o quasi) degli oggetti contati. Una simile «formula» è decisamente priva di valore e poco utile, conclude l'esimio professore.

Con queste autorevolissime parole, spero vivamente di aver trasmesso almeno un'idea dello spirito della matematica discreta e costruttiva, che si esprime sovente in uno spiccato senso estetico per le formule, con forti valenze anche etiche e pragmatiche dal punto di vista computazionale e algoritmico.

4.2 La ricorrenza euleriana.

Dopo questa doverosa introduzione, passiamo alle *buone notizie*, informaticamente parlando: sfruttando la relazione di ricorrenza già nota a Leonhard Euler (1707-1783) e sue semplici derivate, è comunque possibile costruire algoritmi elementari ed accettabilmente efficienti per il calcolo esatto di $P(n)$ e valori correlati, che (dopo lo scontro con la «spaventosa» formula di Hardy-Ramanujan-Rademacher) fanno tornare il sorriso ai programmati, giovani e meno giovani, che si trovino ad affrontare l'argomento. Ovviamente chiariamo subito che si tratta di una ricorrenza, nella quale cioè occorre calcolare iterativamente per accumulo tutti i valori compresi tra $P(1)$ e $P(n)$: pertanto la sua collocazione ideale, così come per gli algoritmi di crivello [CM05], è nella generazione di tabelle LUT eseguita una tantum e di conseguenza l'aspetto prestazionale passa in secondo piano.

La relazione di ricorrenza utilizzata è la seguente, già nota come accennato al grande matematico svizzero Leonhard Euler:

$$\begin{aligned} P(0) &= 1 \\ P(n) &= \sum_{\substack{j \geq 1 \\ \omega \leq n}} (-1)^{j+1} P(n - \omega) + \sum_{\substack{j \geq 1 \\ \omega' \leq n}} (-1)^{j+1} P(n - \omega') \end{aligned} \tag{10}$$

dove $\omega = j(3j - 1)/2$, $\omega' = j(3j + 1)/2$. La formulazione qui proposta è strettamente analoga a quella originariamente riportata da Kreher & Stinson [KS98] per i loro scopi, con un'unica aggiunta: l'esplicitazione delle variabili intermedie omega, che in realtà rappresentano numeri pentagonali generalizzati⁵ - variabili da loro stessi introdotte e utilizzate nell'algoritmo. La complessità computazionale di questa soluzione è accettabile, a grandi linee: l'algoritmo esegue solo $\Theta(n^{\frac{3}{2}})$ somme.

⁵Un numero pentagonale è un numero figurato poligonale generato da un numero intero positivo n secondo la seguente formula: $p_n = n(3n - 1)/2$. I primi numeri pentagonali sono: 1, 5, 12, 22, 35, 51, 70, ... (OEIS A000326). Vale anche la pena di rilevare che il numero pentagonale i -esimo è pari a $1/3$ del corrispondente numero triangolare di posizione $3i - 1$. Un numero pentagonale generalizzato assume la forma $q_n = n(3n - 1)/2$ con $n \in \mathbb{Z}$. La successione generalizzata assume pertanto il seguente aspetto: 0, 1, 2, 5, 7, 12, 15, 22, 26, 35, 40, 51, ... (OEIS A001318). Equivalentemente, preso $m = |n|$, si può scrivere:

$$q_n \stackrel{\text{def}}{=} \begin{cases} \frac{3m^2 - m}{2} & \text{quando } n \geq 0 \\ \frac{3m^2 + m}{2} & \text{quando } n < 0 \end{cases}$$

Con l'evidente vantaggio che $m \in \mathbb{N}$. Si vedano, ad esempio, [Sil97, Pap93].

4.3 L'implementazione di Kreher & Stinson.

```

/*
** IntegerPartitions.c
** October 1, 1997
** This program implements Algorithms 3.1 – 3.9
** by Kreher & Stinson
*/

#include <stdio.h>
#include <stdlib.h>

int P[25];

void EnumPartitions2(int m)
/*
** Algorithm 3.6
** compute the partition numbers P[i] for i <= m
*/
{
    int i;

    P[0] = 1;
    P[1] = 1;

    for (i = 2; i <= m; ++i) {
        int j, sum, omega, omega2, sign;

        sign = 1;
        omega = 1;
        sum = 0;
        j = 1;

        while (omega <= i) {
            omega2 = omega + j;

            sum += sign * P[i - omega];
            if (omega2 <= i) {
                sum += sign * P[i - omega2];
            }

            omega += 3 * j + 1;
            ++j;
            sign = -sign;
        }

        P[i] = sum;
    }
}
***** */
int main(void) {

```

Non posso qui esimermi dal citare uno splendido ed elegante teorema, la cui tesi è una delle numerose congettura di monsieur Pierre de Fermat (1601-1665): qualsiasi numero naturale può essere espresso come somma di tre numeri triangolari oppure di cinque numeri pentagonali, e più in generale si può esprimere qualsiasi naturale sommando n numeri n -gonali. A questa congettura si riferiva Carl Friederich Gauss (1777-1855) quando annotava nei suoi diari, nel 1796, la famosa espressione:

$$\epsilon\nu\rho\eta\kappa\alpha \text{ num} = \Delta + \Delta + \Delta$$

Tale apparentemente criptica espressione, rifacendosi a sua volta al leggendario «Eureka!» di Archimede, sta ad indicare che il Princeps Mathematicorum aveva appunto dimostrato il caso della congettura relativo ai numeri triangolari: dimostrazione poi puntualmente riportata nelle sue «Disquisitiones Arithmeticae» pubblicate due anni dopo, nel 1798. Per una moderna traduzione in inglese, si veda [GC65].

```

int i, n = 22;

printf("\nTest of Algorithm 3.6 with n = %d\n\n", n);

EnumPartitions2(n);

printf(" n | ");

for (i = 1; i <= n; i = i + 1) {
    printf("%5d", i);
}

printf("\n-----|");

for (i = 1; i <= n; i = i + 1) {
    printf("-----");
}

printf("\nP[n] | ");

for (i = 1; i <= n; i = i + 1) {
    printf("%5d", P[i]);
}

printf("\n\nEnd of all tests.\n\n");
return (0);
}
/** EOF: IntPart.c ***/

```

Non tentate di utilizzare il codice, chiaramente didattico, per numeri troppo grandi: al di là dell'ovvio limite nell'array statico $P[25]$, la funzione di partizione cresce in modo esplosivo al crescere di n , con ovvie conseguenze in termini di rappresentabilità hardware del risultato con i tipi di default. Per andare oltre occorrono apposite librerie, come la nota MIRACL: oppure il buon vecchio Python.

Il codice qui presentato è un semplice riadattamento dell'algoritmo euleriano 3.6 del Kreher-Stinson: il loro sorgente C `IntegerPartitions.c` è contenuto nell'archivio `GEN.tar.gz`, liberamente scaricabile presso la sezione "source code" della home page di supporto al testo. L'algoritmo euleriano proposto in [KS98] originariamente era stato sviluppato in Pascal (come il resto del loro codice): per la stesura del testo è stato reimplementato dagli autori in linguaggio C e compilato, all'epoca, con la versione 2 di GCC. Tuttavia, è bene sottolineare che si tratta pur sempre di un funzione che opera in $\Theta(n^{\frac{3}{2}})$, ben lontana dall'ideale $O(1)$ garantito dalle formule chiuse esistenti per numerosi altri problemi analoghi. Le implementazioni interne dei grandi framework di calcolo sono ovviamente table-based per i piccoli valori di n , e per grandi numeri fanno quasi sempre uso di algoritmi basati sulla laboriosa valutazione della formula di Rademacher modificata. Ad esempio, citerei volentieri l'ottimo lavoro di Jonathan Bober incluso nel noto ambiente SAGE.

Tornando ad Euler, come funziona in realtà l'algoritmo di K&S? Vediamo di delinearne almeno l'idea generale. Sicuramente è migliorativo rispetto all'algoritmo 3.5 presente nel medesimo testo (e nel sorgente indicato), che opera in $\Theta(n^2)$ se utilizzato per il calcolo di $P(n)$. Il codice riportato, tuttavia, non somiglia alla sommatoria (10), almeno a prima vista: cerchiamo di analizzare brevemente queste differenze. Come d'abitudine in questi casi, la principale preoccupazione degli autori è stata quella di eliminare dal loop le operazioni computazionalmente più costose sulla piattaforma di riferimento Intel x86: tipicamente le *moltiplicazioni tra variabili*. Per cominciare, scriviamo esplicitamente la differenza tra due generici valori consecutivi di omega:

$$\omega_j - \omega_{j-1} = \frac{3j^2 - j}{2} - \frac{3(j-1)^2 - (j-1)}{2} = 3j - 2$$

$$\omega_{j+1} - \omega_j = \frac{3(j+1)^2 - (j+1)}{2} - \frac{3j^2 - j}{2} = 3j + 1$$

Ovvero:

$$\omega_j = \omega_{j-1} + 3j - 2 \tag{11}$$

O equivalentemente:

$$\omega_{j+1} = \omega_j + 3j + 1 \quad (12)$$

La semplificazione ottenuta è già utile: scelta ad esempio la (11), accumulando in ω la quantità $3j - 2$ ad ogni iterazione, si ottiene la successione desiderata, eliminando operazioni relativamente (e misurabilmente!) costose.

Ma non è tutto: applicando ricorsivamente il procedimento, si può determinare allo stesso modo il passo di incremento di codesta espressione, partendo nuovamente dalla (11) per fissare le idee:

$$\Delta_j = 3(j + 1) - 2 - 3j + 2 = 3$$

Questo elimina definitivamente ogni moltiplicazione nella sezione di aggiornamento delle variabili connesse a quella di induzione. Ricaviamo banalmente anche il secondo indice (nel codice originale viene denominato `omega2`) in funzione di ω :

$$\omega'_j - \omega_j = \frac{3j^2 + j}{2} - \frac{3j^2 - j}{2} = j$$

Da cui:

$$\omega'_j = \omega_j + j \quad (13)$$

Leggendo il codice originale, ci sono ancora almeno due aspetti che spesso lasciano perplessi giovani sviluppatori e studenti nonostante questa spiegazione (omessa dagli autori per la sua evidenza), e riguardano proprio l'applicazione delle semplici relazioni appena ricavate. Risulta piuttosto ovvio che, una volta eliminato aritmeticamente il ricorso a moltiplicazioni, divisioni, modulo/resto e potenze (quest'ultime con l'incombente spettro dell'uso di FP da parte del compilatore) ad ogni passo di iterazione, le opzioni di riscrittura e ridistribuzione delle operazioni per queste equazioni sono varie, ma quasi tutte computazionalmente equivalenti, a meno di minime differenze da evidenziare con misure puntuali su un preciso target.

Tuttavia, nel codice originale, invece di ricorrere ad un idioma immediato come il seguente:

$$\begin{cases} j & \leftarrow j + 1 \\ \omega & \leftarrow \omega + 3j - 2 \\ \omega' & \leftarrow \omega + j \end{cases}$$

o, meglio ancora:

$$\begin{cases} j & \leftarrow j + 1 \\ k & \leftarrow k + 3 \\ \omega & \leftarrow \omega + k \\ \omega' & \leftarrow \omega + j \end{cases}$$

gli autori hanno optato per una forma diversa, con l'idea di minimizzare il numero complessivo di addizioni e incrementi unitari generati dal compilatore di riferimento, ossia GCC. Ciò, all'epoca in cui è stato concepito e sperimentato il codice, nella seconda metà dei Novanta, aveva ampiamente senso sulle CPU x86 (e non solo). Tenendo opportunamente conto della dinamica della variabile di induzione j , si giunge facilmente all'implementazione suggerita, che utilizza la formula (12) per l'incremento di ω a monte dell'incremento della variabile di induzione, e la formula (13) per `omega2` a valle di tale incremento.

$$\begin{cases} \omega & \leftarrow \omega + 3j + 1 \\ j & \leftarrow j + 1 \\ \omega' & \leftarrow \omega + j \end{cases} \quad (14)$$

Ulteriori accorgimenti implementativi presenti nel codice originale, probabilmente ancora più importanti con la combinazione di compilatore e hardware scelta all'epoca dagli autori, erano volti ad eliminare l'uso di ogni possibile istruzione IMUL in tutto il loop interno, facendo uso in particolare di `if()` per scegliere il segno del termine e affidandosi alla gestione della eventuale BPU (Branch Prediction Unit) nell'optimizer. Ritengo tuttavia doveroso ricordare ai miei lettori che, prima di ricorrere a «ottimizzazioni» di quest'ultimo tipo nel codice C, è sempre indispensabile valutare varie soluzioni e combinazioni di flag di ottimizzazione, usando correttamente il profiler e verificando con cura il codice assembly generato dal compilatore per la macchina target, per non andare incontro a strane «sorprese».

4.4 Brevi cenni sull'ottimizzazione.

Abbiamo già acclarato che l'algoritmo presentato, come tutti i suoi omologhi e come i vari crivelli numerici, in definitiva nasce solo per creare delle tabelle LUT da utilizzare poi in un secondo momento: pertanto di fatto

non è molto rilevante se opera in pochi istanti o piuttosto se richiede intere giornate di elaborazione, come accadeva con talune tabelle numeriche (alcune delle quali famose protagoniste di vere e proprie «bibbie» del calcolo applicativo, ancor oggi in uso) ancora ai tempi gloriosi dell'IBM S/360 o dei VAX 11/750. Tuttavia, sul blog abbiamo preso a pretesto questa semplicissima routine per mostrare con quanta facilità ancor oggi si può aggirare o piegare ai propri scopi la presunta «intelligenza» dei compilatori mainstream, e come - con altrettanta facilità - è facile inciamparvi a proprio detrimento, per eccesso di fideismo.

I test condotti su XP Pro SP3 all'epoca della prima pubblicazione sul blog non vengono riproposti in questa sede, essendo sostanzialmente oggi di mero interesse museale. Tuttavia, quel che interessava mostrare all'epoca rimane perfettamente valido: pur attivando tutte le opzioni più ragionevoli di ottimizzazione dei vari compilatori mainstream, certe «ottimizzazioni» nel sorgente finiscono per sortire risultati perfino controproducenti, e per contro basta pochissimo per ottenere sorprendenti miglioramenti nel runtime. A costo di apparire monotoni, quanto sopra sottolinea ancora una volta l'importanza di procedere in modo sistematico quando, lavorando in C ed avendo a che fare con un algoritmo in $\Theta(n^{\frac{3}{2}})$, le prestazioni diventano fondamentali e, per i più vari motivi, è escluso il ricorso a librerie ottimizzate, linguaggi dedicati, ambienti verticali:

1. Verificare le implementazioni di riferimento;
2. Attivare tutte le opzioni di ottimizzazione del compilatore;
3. Leggere il codice Assembly prodotto dal compilatore, almeno nelle sezioni più critiche;
4. Eseguire profiling sistematici;
5. Ricorrere alla micro-ottimizzazione e ai *builtins*, laddove disponibili.

Non si dia per scontato che il compilatore C/C++ sia sempre in grado di operare miracoli, perché (come ampiamente mostrato e documentato all'epoca dei profiling test sistematici a cui si fa cenno) è veramente banale mettere in crisi anche i più moderni optimizer con poche righe di codice e una maldestra, pedissequa «traduzione» di formule analitiche prese da qualche testo teorico senza tenere conto delle implicazioni a livello di codice e prestazioni (per non parlare di accuratezza e stabilità). Peraltro, sulle piattaforme embedded il break-even point tra costante necessità di ottimizzazione estrema e limitate capacità dei cross-compiler sposta molto verso il low level e il ricorso all'Assembly (inline o meno) l'area di intervento del programmatore; mentre è invece generalmente vero che sui PC mainstream, pur con tutto il loro gravame di retrocompatibilità e microcodice, la necessità di ottimizzare anche una «semplice» espressione algebrica al cuore di un inner loop è sempre meno sentita: ma non per questo si deve ignorare tale possibilità. Al di là degli aspetti sperimentali e del banale esempio, rimane l'importanza di riasserire che anche agli inizi del terzo millennio «*il miglior ottimizzatore rimane sempre situato tra le vostre orecchie*», secondo il sacrosanto aforisma di Michael Abrash. Sebbene le motivazioni della scuola di pensiero che mette in guardia dalle *early optimizations* siano di norma ragionevoli e sensate (ma è ben altro rispetto a ciò di cui si parla qui), sovente le puerili obiezioni del tipo «con i processori e i compilatori di oggi è inutile ottimizzare» sono solo la moderna traduzione del «monum matura est», non è ancora matura, come disse la volpe di Fedro davanti al grappolo d'uva troppo alto per lei. Il vero problema è che l'ottimizzazione in tutti i suoi aspetti (numerica, del codice, locale, globale...) esula sempre più dalla preparazione impartita oggi allo «sviluppatore quadratico medio», sulla base di una illusione di onnipotenza della «modernità» eccessivamente diffusa in ambito didattico.

5 Altre formule per $P(n)$.

Alcuni anni dopo la pubblicazione della prima serie di interventi sul blog, due nuovi articoli di ricerca hanno portato una ventata di novità in questo settore sostanzialmente fermo da decenni (per non dire da secoli, pensando alla ricorrenza euleriana ancor oggi largamente utilizzata). Anche se l'interesse computazionale di tali formulazioni è limitato, in special modo relativamente al lavoro di Ken Ono e soci, vale comunque la pena di fornire qualche accenno: soprattutto al metodo di Jerome Malenfant [Mal11] che comunque ha ancora solidissime radici euleriane e, peraltro, si applica anche a numerose altre funzioni generatrici, incluse quelle dei numeri di Euler (ancora lui), Bernoulli e Stirling.

Vale la pena di menzionare brevemente il fatto che gli stessi Kelleher e O'Sullivan, nel medesimo articolo [KO09] in cui hanno definito l'algoritmo (2), hanno fornito una loro espressione in forma chiusa come sommatoria finita per $P(n)$:

$$P(n) = \frac{1}{2} \left(1 + \sum_{a_1 \dots a_k \in \mathcal{A}'(n)} \left\lfloor \frac{a_{k-1} + a_k}{a_{k-1} + 1} \right\rfloor \right) \quad (15)$$

dove $\mathcal{A}'(n) = \{0a_1 \dots a_k \mid a_1 \dots a_k \in \mathcal{A}(n)\}$ e $\mathcal{A}(n)$ è l'insieme di tutte le partizioni di n . Tuttavia, come è immediato rilevare, tale «formula» richiede di fatto un conteggio esaustivo di tutte le partizioni per poterne

sommare sistematicamente le parti più grandi λ_1 e λ_2 ⁶, pertanto essa manca di effettiva utilità computazionale, come peraltro apertamente riconosciuto dagli stessi autori.

5.1 La formula di Malenfant.

La formula fa uso di numeri pentagonali generalizzati (vedi nota 5). Per facilitare la lettura e snellire il formalismo, definiamo una semplice funzione accessoria $gp(m)$, che ci fornisce direttamente l' m -esimo numero pentagonale generalizzato, per $m \in \mathbb{N}$:

$$gp(m) \stackrel{\text{def}}{=} \begin{cases} \frac{3m^2+2m}{8} & \text{per } m \text{ pari} \\ \frac{3m^2+4m+1}{8} & \text{per } m \text{ dispari} \end{cases} \quad (16)$$

Rimarcando *en passant* che lo zero è considerato pari, l'uso di tale notazione arbitraria ci consente di referenziare un qualsiasi numero pentagonale generalizzato, pur utilizzando unicamente valori naturali per m , per comodità di scrittura e uniformità computazionale.

Ricordiamo inoltre brevemente la definizione di coefficiente multinomiale. Si abbiano n interi positivi k_1, \dots, k_n , con $n > 1$:

$$\binom{k_1 + k_2 + \dots + k_n}{k_1, k_2, \dots, k_n} \stackrel{\text{def}}{=} \frac{(k_1 + k_2 + \dots + k_n)!}{k_1! k_2! \dots k_n!} = \frac{\left(\sum_{j=1}^n k_j \right)!}{\prod_{j=1}^n k_j!} \quad (17)$$

La formula che ci interessa fa anche uso della funzione delta di Kronecker (8), ideata da Leopold Kronecker (1823-1891) con una semantica che può facilmente essere compresa dall'informatico quadratico medio dei giorni nostri, in quanto si comporta esattamente come un operatore booleano di confronto entro un costrutto condizionale, supportato da tutti i linguaggi di programmazione degni di tale definizione [BJ66]. Armati di queste sole semplicissime definizioni, è già possibile dare una prima occhiata - in una versione opportunamente semplificata - alla formula più recente, proposta nel maggio del 2011 dal fisico Jerome Malenfant, la quale pare adattissima da un punto di vista didattico. La formula in oggetto, al contrario della formula di Hardy-Ramanujan-Rademacher (7), esprime il numero di partizione $P(n)$ tramite una somma *finita* di termini. In prima approssimazione, la scriviamo come segue:

$$P(n) = \sum_{0 \leq k_1, k_2, \dots, k_M \leq n} (-1)^S \binom{K}{k_1, k_2, \dots, k_M} \delta \left(n, \sum_{m=1}^M k_m \cdot gp(m) \right) \quad (18)$$

Dove M è il numero naturale tale che $gp(M)$ restituisce il più grande numero pentagonale generalizzato non maggiore di n , e vale inoltre:

$$K = k_1 + k_2 + \dots + k_M = \sum_{i=1}^M k_i$$

In estrema sintesi, «cosa fa» codesta formula? Tale sommatoria multipla enumera le disposizioni con ripetizioni⁷ dei coefficienti in una somma di numeri pentagonali generalizzati. Talune disposizioni - quelle che non corrispondono a *partizioni pentagonali* del numero n dato - vengono semplicemente scartate tramite la delta di Kronecker, che in tali casi annulla l'intero prodotto. Le disposizioni rimanenti, usate nel calcolo dei coefficienti multinomiali, generano i risultati intermedi che vengono infine sommati algebricamente per ottenere il risultato finale desiderato, ossia il numero complessivo di partizioni dell'intero n dato.

⁶Si ricordi che i due autori utilizzano un ordinamento *ascendente* delle parti, contrariamente alle convenzioni utilizzate nel resto della letteratura. Pertanto, nella loro notazione, a_k e a_{k-1} sono rispettivamente la più grande parte e la seconda più grande. L'aggiunta dell'elemento nullo a ciascuna partizione è un banale accorgimento per evitare l'eccezione della partizione con una singola parte $\lambda_1 = n$. In ogni caso, gli autori dimostrano un semplice teorema dal quale ricavano l'espressione della sommatoria $\left[\frac{\lambda_2+\lambda_1}{\lambda_2+1} \right]$, riscritta secondo la notazione impiegata in questo articolo.

⁷Dati n simboli distinti e un intero $0 < k \leq n$, si dicono *disposizioni con ripetizioni* di questi n elementi in classe k tutte le possibili presentazioni tali che:

- Ciascuna presentazione contenga esattamente k simboli, non necessariamente distinti;
- Ciascun simbolo possa essere ripetuto fino a k volte;
- Due presentazioni differiscano per qualche simbolo, oppure per l'ordine in cui i simboli sono disposti.

Il totale delle possibili disposizioni con ripetizioni di n elementi in classe k è pari a $DR(n, k) = n^k$. Ad esempio, i $DR(2, 4) = 16$ numeri binari a 4 bit 0000, 0001, ..., 1111 sono uno dei più classici esempi di disposizione con ripetizione, e le disposizioni con ripetizione $DR(3, 2) = 9$ degli elementi dell'insieme $A = \{a, b, c\}$ sono tutte le coppie del relativo prodotto cartesiano $aa, ab, ac, ba, bb, bc, ca, cb, cc$.

Per poter usare operativamente la formula di Malenfant «con carta e penna» occorre ora solo vedere insieme un paio di banali dettagli. L'espressione S che compare all'esponente del -1 , il quale a sua volta determina il segno di ciascun coefficiente multinomiale nella sommatoria, è una *somma selettiva* dei coefficienti k_i . In particolare, partecipano a codesta somma solamente i coefficienti relativi ai numeri pentagonali generalizzati generati da un intero relativo *pari*, zero escluso: $\pm 2, \pm 4, \pm 6, \dots$. Nella nostra notazione, a causa del comodo cambio di indice arbitrario, ciò implica soltanto che dovremo includere in tale somma tutti e soli quei coefficienti che corrispondono a posizioni successive alla prima e congruenti a 0 oppure a 3 in modulo 4, quindi in sostanza:

$$S = k_3 + k_4 + k_7 + k_8 + k_{11} + k_{12} + \dots$$

Aggiungo solamente che, a rigore, noi siamo interessati unicamente al bit meno significativo di tale risultato e potremmo parimenti voler scrivere $S(\text{mod } 2)$ nella formula, in un contesto più formale.

In ultimo - ma certo non meno importante - resta da definire come far variare i coefficienti k_i . Sarebbe ingenuo, ingiustificato e inefficiente far variare ogni coefficiente in $0 \leq k_i \leq n$. Non è infatti necessario esaminare tutto lo spazio delle possibili disposizioni con elementi ripetuti, che consta di n^M elementi: sappiamo già a priori che in taluni intervalli nessuna delle disposizioni generate potrà essere una partizione del numero n dato. Dobbiamo ora quantificare meglio l'espressione lasciata vaga, per amor di sintesi, nella (18).

Sappiamo che una partizione di un numero naturale n può avere, al massimo, n parti (1). Estendiamo il ragionamento ad alcuni semplici esempi⁸:

$$\begin{aligned} n &\geq \underbrace{2 + 2 + \cdots + 2}_{\lfloor n/2 \rfloor} \\ n &\geq \underbrace{5 + 5 + \cdots + 5}_{\lfloor n/5 \rfloor} \\ n &\geq \underbrace{7 + 7 + \cdots + 7}_{\lfloor n/7 \rfloor} \end{aligned}$$

Generalizzando, si avrà che - con ovvia corrispondenza degli indici - ciascun generico coefficiente è limitato all'intervallo seguente, determinato dalla parte a cui il coefficiente medesimo è associato:

$$k_i \in [0, \left\lfloor \frac{n}{\lambda_i} \right\rfloor] \quad (19)$$

In particolare, per ogni parte il cui valore supera $\lfloor n/2 \rfloor$ i relativi coefficienti associati saranno, al più, unitari. Questa definizione più restrittiva dell'intervallo di variazione per ciascun singolo coefficiente consente evidentemente di migliorare un po' l'efficienza del calcolo. Le disposizioni con elementi ripetuti dei coefficienti così vincolati, con ovvio significato dei simboli, risultano ovviamente in numero inferiore rispetto a quanto stabilito dalla formula generale $DR_n(M) = n^M$:

$$\begin{aligned} Dq(n, M) &= \left(1 + \left\lfloor \frac{n}{gp(1)} \right\rfloor\right) \cdot \left(1 + \left\lfloor \frac{n}{gp(2)} \right\rfloor\right) \cdots \left(1 + \left\lfloor \frac{n}{gp(M)} \right\rfloor\right) \\ &= \prod_{m=1}^M \left(1 + \left\lfloor \frac{n}{gp(m)} \right\rfloor\right) \end{aligned} \quad (20)$$

A questo punto possiamo riscrivere l'espressione della sommatoria multipla in modo meno conciso e un po' più preciso:

$$P(n) = \sum_{k_1=0}^{\lfloor n/gp(1) \rfloor} \sum_{k_2=0}^{\lfloor n/gp(2) \rfloor} \sum_{k_3=0}^{\lfloor n/gp(3) \rfloor} \cdots \sum_{k_M=0}^1 (-1)^S \binom{K}{k_1, k_2, \dots, k_M} \delta \left(n, \sum_{m=1}^M k_m \cdot gp(m) \right) \quad (21)$$

Siamo dunque pronti a calcolare manualmente, come diceva il buon Gottfried Wilhelm von Leibniz (1646-1716) con il suo motto «*Calculemus!*», con un semplicissimo esempio. Scegliamo $n = 5$, ricordando dalla tabella (5) che $P(5) = 7$. I numeri pentagonali generalizzati non maggiori di 5 sono: $gp(1) = 1$, $gp(2) = 2$, $gp(3) = 5$ e ovviamente $M = 3$, quindi i coefficienti in ciascuna somma saranno k_1, k_2 e k_3 mentre l'espressione all'esponente S in questo caso coincide semplicemente, per ciascun termine della somma, col solo coefficiente k_3 : $S = k_3$. Calcoliamo gli intervalli per i tre coefficienti secondo la formula (19):

⁸La funzione floor $\lfloor n \rfloor \stackrel{\text{def}}{=} \max\{m \in \mathbb{N} \mid m \leq n\}$ restituisce il più grande naturale non maggiore di n . Curiosità: la notazione e il nome di questa funzione discreta e di altre analoghe sono stati ideati da Kenneth Iverson (1920-2004), il «padre» di APL e J. Si vedano in particolare [Ive62] e il capitolo 3 del fondamentale [GKP94].

$$0 \leq k_1 \leq 5$$

$$0 \leq k_2 \leq 2$$

$$0 \leq k_3 \leq 1$$

Ricordiamo infine che le possibili disposizioni vincolate con elementi ripetuti, secondo la formula (20), assommano a $Dq(5, 3) = (1+5)(1+2)(1+1) = 36$. Ovviamente, sarebbe possibile scartare a priori la disposizione $\langle 0, 0, 0 \rangle$, ma pedantesamente generiamo comunque tutte le disposizioni con ripetizioni di tali coefficienti, tenendo conto unicamente dei rispettivi vincoli superiori. Sappiamo dalla OEIS A095699 che le partizioni pentagonali del numero cinque sono solamente quattro.

k_1	k_2	k_3	δ	k_1	k_2	k_3	δ
0	0	0		0	0	1	★
1	0	0		1	0	1	
2	0	0		2	0	1	
3	0	0		3	0	1	
4	0	0		4	0	1	
5	0	0	★	5	0	1	
0	1	0		0	1	1	
1	1	0		1	1	1	
2	1	0		2	1	1	
3	1	0	★	3	1	1	
4	1	0		4	1	1	
5	1	0		5	1	1	
0	2	0		0	2	1	
1	2	0	★	1	2	1	
2	2	0		2	2	1	
3	2	0		3	2	1	
4	2	0		4	2	1	
5	2	0		5	2	1	

(22)

Sono marcate con un carattere ★ tutte e sole le disposizioni dei tre coefficienti che risultano in partizioni di $n = 5$, tali cioè che $5 = \sum_{j=1}^3 k_j \cdot gp(j) = k_1 \cdot 1 + k_2 \cdot 2 + k_3 \cdot 5$. In tutti e soli i casi marcati, la delta di Kronecker nella (21) assume valore unitario, mentre vale zero altrove, di fatto eliminando dalla sommatoria i termini che non contribuiscono al totale.

La tabella è stata compilata in ordine tale da evidenziare un altro aspetto. Laddove, come in questo caso, il numero n coincide con l'M-esimo numero pentagonale generalizzato, è ovvio che tutte le disposizioni nelle quali k_M è pari ad uno (tranne chiaramente quella che rappresenta la partizione di n in una singola parte) possono essere scartate a priori, dimezzando di fatto il numero totale di disposizioni con ripetizioni da generare. Tuttavia, da un punto di vista algoritmico, anche un tale dimezzamento non appare particolarmente significativo.

Prendiamo ora in considerazione le sole disposizioni valide della (22), e ricordando che $S = k_3$ e che $K = \sum_{j=1}^3 k_j = k_1 + k_2 + k_3$:

k_1	k_2	k_3	K	$(-1)^S$
0	0	1	1	-1
1	2	0	3	+1
3	1	0	4	+1
5	0	0	5	+1

Si avrà quindi, con ovvi passaggi:

$$P(5) = -\binom{1}{0, 0, 1} + \binom{3}{1, 2, 0} + \binom{4}{3, 1, 0} + \binom{5}{5, 0, 0} = -1 + 3 + 4 + 1 = 7 \quad (23)$$

Giunti a questo punto, i miei lettori avranno già compreso che la formula (21) presa in esame riveste notevole importanza dal punto di vista concettuale, perché fornisce una formula chiusa che esprime il numero di partizione $P(n)$ come somma finita, ma non appare direttamente applicabile. Per scartare del tutto l'idea di tradurre in algoritmo la procedura fin qui seguita, è sufficiente uno sguardo alla seguente tabella, decisamente autoesplicativa. Si presti attenzione in particolare all'andamento della colonna $Dq(n, M)$. Le colonne relative ai primi dieci numeri pentagonali generalizzati $1, 2, \dots, 40$ riportano, per ogni riga, il valore del limite superiore indicato dalla (19) aumentato di una unità, in modo tale che il prodotto di tali valori equivale a $Dq(n, M)$.

n	M	gp(M)	p(n)	Dq(n,M)	1	2	5	7	12	15	22	26	35	40
1	1	1		1	2	2								
2	2	2		2	6	3	2							
3	2	2		3	8	4	2							
4	2	2		5	15	5	3							
5	3	5		7	36	6	3	2						
6	3	5		11	56	7	4	2						
7	4	7		15	128	8	4	2	2					
8	4	7		22	180	9	5	2	2					
9	4	7		30	200	10	5	2	2					
10	4	7		42	396	11	6	3	2					
11	4	7		56	432	12	6	3	2					
12	5	12		77	1.092	13	7	3	2	2				
13	5	12		101	1.176	14	7	3	2	2				
14	5	12		135	2.160	15	8	3	3	2				
15	6	15		176	6.144	16	8	4	3	2	2			
16	6	15		231	7.344	17	9	4	3	2	2			
17	6	15		297	7.776	18	9	4	3	2	2			
18	6	15		385	9.120	19	10	4	3	2	2			
19	6	15		490	9.600	20	10	4	3	2	2			
20	6	15		627	13.860	21	11	5	3	2	2			
21	6	15		792	19.360	22	11	5	4	2	2			
22	7	22		1.002	44.160	23	12	5	4	2	2	2		
23	7	22		1.255	46.080	24	12	5	4	2	2	2		
24	7	22		1.575	78.000	25	13	5	4	3	2	2		
25	7	22		1.958	97.344	26	13	6	4	3	2	2		
26	8	26		2.436	217.728	27	14	6	4	3	2	2	2	
27	8	26		3.010	225.792	28	14	6	4	3	2	2	2	
28	8	26		3.718	313.200	29	15	6	5	3	2	2	2	
29	8	26		4.565	324.000	30	15	6	5	3	2	2	2	
30	8	26		5.604	624.960	31	16	7	5	3	3	2	2	
31	8	26		6.842	645.120	32	16	7	5	3	3	2	2	
32	8	26		8.349	706.860	33	17	7	5	3	3	2	2	
33	8	26		10.143	728.280	34	17	7	5	3	3	2	2	
34	8	26		12.310	793.800	35	18	7	5	3	3	2	2	
35	9	35		14.883	2.239.488	36	18	8	6	3	3	2	2	2
36	9	35		17.977	3.239.424	37	19	8	6	4	3	2	2	2
37	9	35		21.637	3.326.976	38	19	8	6	4	3	2	2	2
38	9	35		26.015	3.594.240	39	20	8	6	4	3	2	2	2
39	9	35		31.185	3.686.400	40	20	8	6	4	3	2	2	2
40	10	40		37.338	8.926.848	41	21	9	6	4	3	2	2	2
41	10	40		44.583	9.144.576	42	21	9	6	4	3	2	2	2
42	10	40		53.174	11.442.816	43	22	9	7	4	3	2	2	2
43	10	40		63.261	11.708.928	44	22	9	7	4	3	2	2	2
44	10	40		75.175	18.779.040	45	23	9	7	4	3	3	2	2
45	10	40		89.134	28.439.040	46	23	10	7	4	4	3	2	2
46	10	40		105.558	30.320.640	47	24	10	7	4	4	3	2	2
47	10	40		124.754	30.965.760	48	24	10	7	4	4	3	2	2
48	10	40		147.273	41.160.000	49	25	10	7	5	4	3	2	2
49	10	40		173.525	48.000.000	50	25	10	8	5	4	3	2	2
50	10	40		204.226	56.010.240	51	26	11	8	5	4	3	2	2

Per essere realmente migliorativo rispetto all'algoritmo euleriano, l'algoritmo di Malenfant dovrebbe operare asintoticamente in meno di $\Theta(n^{\frac{3}{2}})$, mentre è facile argomentare che una versione - già ottimale rispetto all'approcchio di calcolo ingenuo qui illustrato - basata su una generazione vincolata delle partizioni con *early filtering* per la selezione di quelle pentagonali si attesta comunque su $O(n^2)$ o peggiore.

5.2 La matrice di Malenfant.

Nel medesimo articolo [Mal11], il fisico Malenfant ha dimostrato per altra via una generalizzazione di derivazioni in parte già note⁹ ed ha altresì proposto una formulazione alternativa, basata sul calcolo del determinante della seguente matrice di Toeplitz¹⁰:

$$P(n) = \begin{vmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 1 & -1 \\ -1 & 0 & -1 & 0 & 0 & 1 & 1 \\ \vdots & & & & & & \ddots \end{vmatrix}_{nxn} \quad (24)$$

Tale matrice, in realtà, può essere caratterizzata completamente dalla sua prima colonna, i cui elementi $a_{r,c}$ possono assumere unicamente valori appartenenti all'insieme $\{-1, 0, 1\}$ secondo la semplice regola seguente:

$$a_r = \begin{cases} (-1)^{|m|+1} & \text{se esiste } m \in \mathbb{Z} \text{ tale che } q_m = r \\ 0 & \text{altrimenti} \end{cases} \quad (25)$$

Qui con q_m indichiamo il numero pentagonale generalizzato m -esimo, e come da definizione l'indice m assume, in successione, i valori interi $0, \pm 1, \pm 2, \pm 3, \dots$. In altri termini, acquisiscono il valore ± 1 (secondo la parità dell'indice, preso in valore assoluto) tutte e sole quelle righe della prima colonna il cui indice è un numero pentagonale generalizzato.

Vale la pena di sottolineare che in questo contesto seguiamo le usuali convenzioni dell'algebra lineare piuttosto che quelle ordinarie in informatica applicativa: pertanto gli indici partono da 1 e l'interpretazione di default degli indici stessi segue la convenzione r, c ovvero «Riga, Colonna». I valori della prima colonna si propagano poi diagonalmente verso il basso, secondo la definizione di matrice di Toeplitz. Nel caso che ci interessa, abbiamo a che vedere con una matrice che è sostanzialmente diagonale inferiore, nella quale cioè la parte al di sopra della diagonale principale¹¹ è nulla, eccetto ovviamente per la sovradiagonale di tale diagonale principale, i cui elementi valgono -1 :

$$a_{i,i+1} = -1 \text{ per } 1 \leq i \leq n-1$$

In realtà, e in modo del tutto coerente con la regola sopra fornita, si può asserire in maniera computazionalmente corretta che anche l'elemento «di posto zero» del vettore colonna di riferimento contiene il valore -1 . Tale elemento virtuale entra quindi dinamicamente a far parte della matrice solo quando si fa «scorrere» verticalmente la prima colonna per generare le successive. Un modo didatticamente valido per visualizzare dinamicamente la formazione di tale matrice è, infatti, il seguente. Si parte, come sottolineato, dal singolo vettore colonna formato secondo la regola (25):

$$\begin{array}{ll} (1) & 1 \\ (2) & 1 \\ (3) & 0 \\ (4) & 0 \\ (5) & -1 \\ (6) & 0 \\ \vdots & \vdots \end{array}$$

I numeri riportati tra parentesi sono chiaramente gli indici di riga, per facilitare la lettura. La seconda colonna si ricava quindi per scorrimento verticale, spostando in basso di una posizione l'intero vettore colonna iniziale, in modo tale che $a_{r,2} = a_{r-1,1}$ per $2 \leq r \leq n-1$ e inserendo (solamente in questo passaggio!) un valore -1 nella posizione iniziale, lasciata libera dallo scorrimento.

⁹L'articolo è molto sintetico, forse eccessivamente, anche dal punto di vista storico: ad esempio non cita esplicitamente il contributo storico del nostro Francesco Brioschi (1824-1897) o le relazioni con il metodo di Cramer.

¹⁰Una matrice di Toeplitz è semplicemente una matrice caratterizzata da diagonali discendenti da sinistra verso destra i cui elementi assumono ordinatamente valori costanti. Nel nostro caso prendiamo in considerazione una matrice asimmetrica quadrata di lato n , ossia di dimensione identica al numero naturale del quale stiamo calcolando la funzione di partizione $P(n)$.

¹¹Conventionalmente è detta diagonale principale quella situata tra l'angolo in alto a sinistra e quello in basso a destra.

		-1	
(1)	1		-1
		\searrow	
(2)	1		1
		\searrow	
(3)	0		1
		\searrow	
(4)	0		0
		\searrow	
(5)	-1		0
		\searrow	
(6)	0		-1
	:		:

I passaggi successivi sono poi tutti meccanicamente identici al seguente, fino al raggiungimento di n colonne:

(1)	1	-1	0
(2)	1	1	\searrow -1
(3)	0	1	\searrow 1
(4)	0	0	\searrow 1
(5)	-1	0	\searrow 0
(6)	0	-1	\searrow 0
	:	:	\searrow :

Dove l'elemento mancante di volta in volta inserito $a_{1,3} \dots a_{1,n}$ è costantemente pari a zero. Supponendo per comodità $n > 3$, aggiungo qui in modo esplicito anche la banale regola per la formazione della prima riga, già evidente dagli esempi fatti: nel nostro caso $a_{1,1} = 1$, $a_{1,2} = -1$, seguiti da $n - 2$ valori nulli: $a_{1,3} = a_{1,4} = \dots = a_{1,n} = 0$.

Naturalmente una spiegazione così prolissa e pedissequa, per la gioia di tutti gli studenti più giovani, non può che sottendere il classico

Esercizio. Si lascia al lettore interessato, come facile esercizio, la stesura di un programma in linguaggio C che - data in input la sola dimensione n , un intero non segnato maggiore di due - generi ordinatamente i vettori di riferimento (colonna 1, riga 1) e da essi l'intera matrice di Toeplitz desiderata.

Tutto ciò vale appunto, con ogni evidenza, solamente a livello didattico per la «costruzione» della matrice alla lavagna o con un linguaggio che non offre alcun supporto all'algebra matriciale. Negli applicativi di calcolo numerico e nelle librerie per l'algebra lineare, invece, è di norma necessario e sufficiente specificare due vettori (corrispondenti alla prima colonna e alla prima riga) per ottenere la generazione automatica di una generica matrice di Toeplitz.

Segue un semplice esempio di implementazioni in Python, ove si demanda alle comode funzioni matriciali di SciPy il lavoro «sporco».

```
from scipy import zeros
from scipy.linalg import det, toeplitz

#####
# Espressione lambda per il calcolo dell'n-esimo numero pentagonale.
#####
penta = lambda n : n*(3*n-1)/2

#####
# Restituisce l'n-esimo numero pentagonale generalizzato.
# 0, 1, 2, 5, 7, 12, ...
#####
def GPN(n):
```

```

if n < 0:
    return 0
if n%2 == 0:
    return penta(n/2+1)
else:
    return penta(-(n/2+1))

#####
# Funzione accessoria per il calcolo del segno ( $-1^m$ ) legato
# all'indice del GPN considerato.
#####
def GPN_sign(n):
    if n%4==2 or n%4==3:
        return -1
    else:
        return 1

#####
# Funzione accessoria per l'input utente di un numero intero.
# Consente di evitare il ricorso alla farraginosa input().
#####
def Input_integer(s):
    while True:
        try:
            x = int(raw_input(s))
            if x > 2:
                break
        except ValueError:
            print ("Valore non numerico: ripetere l'input...")
    return x

#####
## Funzione di partizione implementata con il metodo Malenfant:
## calcolo del determinante di una matrice di Toeplitz.
#####

def Partition_function(n):
    # Si preparano i due vettori di base per la matrice di
    # Toeplitz: prima colonna, prima riga.
    C = zeros(n)
    R = zeros(n)
    R[0]=1
    R[1]=-1
    i = 0

    # Si popola il vettore colonna con un metodo diretto,
    # immettendo gli opportuni valori solo nelle locazioni
    # indicizzate da numeri pentagonali generalizzati.
    while True:
        if GPN(i) > n:
            break
        else:
            C[GPN(i)-1] = GPN_sign(i)
            i = i +1

    # Si genera la matrice e se ne calcola il determinante
    # usando metodi generici di libreria (scipy.linalg).
    # Si noti che esistono algoritmi dedicati per questo genere
    # di matrice, con prestazioni superiori.
    return int(det(toeplitz(C,R)))

```

```
#####
# Routine di prova
#
#####
n = Input_integer("Immettere il valore di n (minimo 3): ")
print "P({0}) = {1}".format(n, Partition_function(n))
```

È ben noto che la classica versione basata sul calcolo numerico (es. decomposizione LU) di un determinante Toeplitz ha prestazioni tipiche dell'ordine di $O(n^2)$. Tuttavia esistono algoritmi dedicati con prestazioni migliori, come indicano le seguenti essenziali coordinate bibliografiche: [KB00, Ste03, CGS⁺08, Li11]. Taluni algoritmi accuratamente ottimizzati presenti in letteratura possono offrire prestazioni dell'ordine di $O(n \cdot \log^a(n))$, con a piccolo intero positivo (tipicamente $a = 3$), quindi - considerati a sé stanti - risulterebbero migliori rispetto all'algoritmo di Euler già analizzato a suo tempo. Non risulta trascurabile ai fini delle prestazioni, tuttavia, anche l'inizializzazione dinamica della matrice di Toeplitz a monte del calcolo del determinante.

Vale la pena di sottolineare, *en passant*, che questo è un tipico problema altamente idoneo per computer vettoriali. Naturalmente l'esemplificazione è qui basata su complesse librerie: non è certo questa la sede adatta ad una disamina anche superficiale dei metodi numerici dedicati più validi ed efficienti per il calcolo del determinante di una siffatta matrice di interi, l'analisi ed implementazione dei quali richiederebbero ben altri spazi. Ma chiaramente non è questo il nostro scopo: spero invece di avere fin qui fornito una idea chiara per il più vasto pubblico possibile - pensando soprattutto agli studenti più giovani ed ai practitioners meno familiari con la notazione usata - della soluzione proposta da Malenfant per esprimere la funzione di partizione come *somma finita* di termini, e in alternativa come determinante di una matrice di costruzione e risoluzione relativamente semplici da un punto di vista dell'efficienza computazionale.

5.3 La formula di Ono-Bruinier.

Il risultato ottenuto da Ken Ono (Emory University) e Jan Hendrik Bruinier in [FKO] si presenta come segue:

$$P(n) = \frac{1}{24n-1} \cdot \sum_{Q \in Q_n} \mathcal{P}(\alpha_Q) \quad (26)$$

La formula appare decisamente elegante e concisa. Tuttavia, prima di abbandonarci a facili entusiasmi matematici, occorre considerare gli aspetti computazionali. Se è vero che l'articolata dimostrazione garantisce che ogni singolo addendo nella (26) sia algebrico, va però chiarito che la funzione $\mathcal{P}()$ che compare nella sommatoria rappresenta una forma modulare non olomorfa di peso nullo, appartenente alla famiglia delle forme deboli di Maass. Non si tratta esattamente di un oggetto matematico «amichevole» dal punto di vista computazionale, né risulta di immediata comprensione. Dirò solo che simili funzioni sono da decenni in uso nella teoria algebrica dei numeri, la più pura e sofisticata branca della matematica pura, che in modo del tutto contorto studia gli «innocui» numeri interi attraverso gli strumenti e i metodi dell'analisi complessa.

La valutazione di tale somma, in breve, consiste di una parte preliminare di natura combinatoria, seguita dalla valutazione (numerica) di una serie di Fourier applicata alla forma modulare predetta. Per quanto ci riguarda, la prima parte potrebbe anche essere oggetto di interesse per sé stessa, in quanto utilizza metodi di generazione di ideali algebrici e/o di corrispondenti forme quadratiche binarie definite positive con determinante della forma $-24n+1$ e ulteriori vincoli sui coefficienti, applicando poi la corrispondenza di Gross-Kohnen-Zagier. Tuttavia, da quel punto in poi, temo che sarebbe un lavoro improbo proseguire sia nella esemplificazione con poche righe di codice in un HLL idoneo, sia nella spiegazione elementare del risultato, mantenendola comprensibile per la maggioranza dei lettori.

Al di là degli aspetti computazionali (sono tuttora in corso ulteriori studi anche dal punto di vista algoritmico), l'importanza matematica di questa formula è veramente notevole, perché conferma delle ipotesi «visionarie» risalenti a Ramanujan e svela un inatteso comportamento macroscopico *di tipo frattale* nella funzione di partizione $P(n)$. Invito tutti gli interessati a seguire la lezione introduttiva di Ken Ono che richiede un background matematico minimale. A causa delle considerazioni sopra riassunte, seppur a malincuore, ho dovuto pertanto optare per limitare drasticamente lo spazio dedicato al lavoro di Ono e soci, che invece meriterebbe ben altra attenzione per il suo portato filosofico ed epistemico.

6 Conclusioni.

Come già ricordato in apertura, il materiale qui presentato costituisce sostanzialmente una rielaborazione e riorganizzazione di contenuti già pubblicati sul *blog* dell'autore «Titolo provvisorio...». Il presente articolo

riepilogativo ha richiamato la definizione elementare di partizione di un numero naturale e alcuni concetti immediatamente ad essa correlati. Si è presentato, in forma volutamente leggera ed ironica, il principale problema aperto in tale settore: la mancanza di una formula chiusa per il calcolo diretto di $P(n)$. A margine si sono presentati semplici ma potenti sorgenti esemplificativi in quattro diversi linguaggi (C, Python, J, COMAL), accompagnati dalla descrizione in pseudocodice di due fondamentali algoritmi e numerosi richiami ai principali capisaldi della letteratura scientifica e applicativa in materia, sia pure senza alcuna pretesa di esaustività.

In chiusura si sono presentati alcuni risultati teorici molto recenti, corredati ove opportuno di esempi di calcolo e di un ulteriore sorgente in Python. Tali risultati, come sottolineato, in ultima analisi rivestono limitato interesse applicativo, ma comunque indicano ancora nel terzo millennio, dopo secoli di studi sulle partizioni, una inesauribile attività alla ricerca di una formula chiusa «soddisfacente» per $P(n)$ secondo i criteri condivisi dalla comunità scientifica nell’ambito della matematica discreta, che qui abbiamo richiamato con le magistrali parole del professor Stanley. L’Autore spera di avere almeno incuriosito il lettore, incoraggiando ulteriori approfondimenti e ricerche.

Riferimenti bibliografici

- [AE04] George E. Andrews and Kimmo Eriksson, *Integer partitions*, Cambridge University Press, 2004.
- [And84] George Andrews, *The theory of partitions*, Cambridge University Press, Cambridge, 1984.
- [Arn10] Jörg Arndt, *Matters computational*, Springer Berlin Heidelberg, 2010.
- [AW95] G. Almkvist and H. S. Wilf, *On the coefficients in the hardy-ramanujan-rademacher formula for $p(n)$* , Journal of Number Theory **50** (1995), no. 2, 329–334.
- [BJ66] Corrado Böhm and Giuseppe Jacopini, *Flow diagrams, turing machines and languages with only two formation rules*, Commun. ACM **9** (1966), no. 5, 366371.
- [BMSW54] Robert L. Bivins, N. Metropolis, Paul R. Stein, and Mark B. Wells, *Characters of the symmetric groups of degree 15 and 16*, Mathematical Tables and Other Aids to Computation **8** (1954), no. 48, 212–216.
- [Bos08] R. E. Boss, *Partitions of numbers: an efficient algorithm in j*, Vector **23** (2008), no. 4, 121–132.
- [Bow03] Frank Bowman, *Introduction to bessel functions*, Dover Publications Inc., 2003.
- [CGS⁺08] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, and J. Zhu, *A superfast algorithm for toeplitz systems of linear equations*, SIAM Journal on Matrix Analysis and Applications **29** (2008), no. 4, 1247–1266.
- [CM05] Alina Carmen Cojocaru and M. Ram Murty, *An introduction to sieve methods and their applications*, London Mathematical Society Student Texts, Cambridge University Press, 2005.
- [Com55] Stig Comét, *Notations for partitions*, Mathematical Tables and Other Aids to Computation **9** (1955), no. 52, 143–146.
- [dS04] Marcus du Sautoy, *The music of the primes*, HarperCollins Publishers, 2004.
- [FKO] A. Folsom, Z. A. Kent, and K. Ono, *L-adic properties of the partition function*, 1586–1609.
- [Ful96] William Fulton, *Young tableaux: With applications to representation theory and geometry*, London Mathematical Society Student Texts, Cambridge University Press, 1996.
- [GC65] Carl Friedrich Gauss and Arthur A. Clarke, *Disquisitiones arithmeticae*, Yale University Press, 1965.
- [GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete mathematics: A foundation for computer science*, 2nd ed., Addison-Wesley, Reading, MA, 1994.
- [HR18] G. H. Hardy and S. Ramanujan, *Asymptotic formulae in combinatory analysis*, Proceedings of the London Mathematical Society **s2-17** (1918), no. 1, 75–115.
- [Ive62] Kenneth Iverson, *A programming language*, John Wiley and Sons, Inc, New York, 1962.
- [KB00] Peter Kravanja and Marc Van Barel, Numerical Algorithms **24** (2000), no. 1/2, 99–116.
- [Knu05] Donald E. Knuth, *The art of computer programming, volume 4, fascicle 2: Generating all tuples and permutations*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2005.

- [KO09] Jerome Kelleher and Barry O’Sullivan, *Generating all partitions: A comparison of two encodings*, 2009.
- [KS98] Kreher and Stinson, *Combinatorial algorithms: Generation, enumeration, and search*, CRC PR INC, 1998.
- [Leb72] N. N. Lebedev, *Special functions and their applications*, Dover Publications Inc., 1972.
- [Li11] Hsuan-Chu Li, *On calculating the determinants of toeplitz matrices*, Journal of Applied Mathematics and Bioinformatics **1** (2011).
- [Mal11] Jerome Malenfant, *Finite, closed-form expressions for the partition function and for euler, bernoulli, and stirling numbers*, 2011.
- [Mar07] Stuart Martin, *Schur algebras and representation theory*, Cambridge University Press, 2007.
- [MZ97] Munarini and Salvi Zagaglia, *Matematica discreta*, Città Studi Edizioni, 1997.
- [Nij78] Albert Nijenhuis, *Combinatorial algorithms for computers and calculators*, Academic Press, New York, 1978.
- [Pap93] Theoni Pappas, *The joy of mathematics*, Wide World Publishing, U.S., 1993.
- [PW79] E. S. Page and L. B. Wilson, *An introduction to computational combinatorics*, Cambridge University Press, 1979.
- [Rad38] Hans Rademacher, *On the partition function $p(n)$* , Proceedings of the London Mathematical Society **s2-43** (1938), no. 1, 241–254.
- [Rei77] Edward Reingold, *Combinatorial algorithms : theory and practice*, Prentice-Hall, Englewood Cliffs, N.J, 1977.
- [Rus] Frank Ruskey, *Combinatorial generation*, Preliminary working draft. University of Victoria, Victoria, BC, Canada 11, 20.
- [Sch86] M. R. Schroeder, *Number theory in science and communication : with applications in cryptography, physics, digital information, computing, and self-similarity*, Springer-Verlag, Berlin New York, 1986.
- [Sil97] Joseph Silverman, *A friendly introduction to number theory*, Prentice Hall, Upper Saddle River, N.J, 1997.
- [Ski09] Steven S. Skiena, *The algorithm design manual*, Springer-Verlag GmbH, 2009.
- [Sta86a] Richard P. Stanley, *Enumerative combinatorics*, vol. 1, Wadsworth Publ. Co., Belmont, CA, 1986.
- [Sta86b] Dennis Stanton, *Constructive combinatorics*, Springer-Verlag, New York, 1986.
- [Ste03] Michael Stewart, *A superfast toeplitz solver with improved numerical stability*, SIAM Journal on Matrix Analysis and Applications **25** (2003), no. 3, 669–693.
- [Yan00] Winston C. Yang, *Derivatives are essentially integer partitions*, Discrete Mathematics **222** (2000), no. 1, 235–245.
- [ZS94] Antoine Zoghbi and Ivan Stojmenovic, *Fast algorithms for generating integer partitions*, Tech. report, International Journal of Computer Mathematics, 1994.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

La rivolta delle sette sette delle sette...

Sommario

Prendo in prestito volentieri dal geniale Achille Campanile il titolo di questo articolo, che riassume una delle più visitate pagine del mio antico blog. I numeri ciclici sono uno degli argomenti maggiormente gettonati nella divulgazione matematica ed attraggono attenzione anche al di fuori della ristretta cerchia degli specialisti. In effetti non c'è autore di grande diffusione che non se ne sia occupato: Andrews, Conway, Gardner, Hodges... per non dire delle centinaia di siti di ludomatematica e divulgazione che ne parlano. All'epoca dei miei studi si accennava a questa e ad altre «curiosità» numeriche normalmente già al biennio del liceo scientifico. Non mancano peraltro problemi aperti anche in questo settore: come quasi sempre avviene in teoria del numero, perfino le domande apparentemente più banali, del tipo «quanti sono i numeri ciclici» non trovano risposta in un teorema o in una formula chiusa.

«*Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.*»
(Leopold Kronecker, 1823-1891)

1 Introduzione.

Prima di entrare in *medias res*, ripassiamo brevemente alcuni concetti di scuola media, che spesso nella formazione tecnoscientifica rimangono sepolti nella memoria a causa della sovraesposizione nei gradi successivi ad ogni genere di «analisi matematica» a base di numeri reali, complessi e quant'altro.

Atteso che un *numero primo* p è un numero *naturale* (numero intero non negativo, e scriviamo $p \in \mathbb{N}$) che ammette come unici divisori quelli cosiddetti banali, ossia l'unità e il numero stesso, un numero razionale $r \in \mathbb{Q}$ è una frazione $r = \frac{n}{d}$ in cui $n, d \in \mathbb{N}, d \neq 0$ e i due elementi sono detti rispettivamente n numeratore e d denominatore¹.

Dato un numero razionale r , possono avversi solamente due casi:

1. $d = 2^a \cdot 5^b$, con $a, b \in \mathbb{N}$. In questo specifico caso, l'espansione decimale della frazione ha sempre un numero **finito** di cifre dopo la virgola².
2. In ogni altro caso, abbiamo a che fare con una sequenza di cifre decimali **infinita**, che prima o poi si ripete: un numero decimale periodico, come ad esempio $1/77 = 0,01298701298701298701298701\dots = 0,\overline{012987}$.

Nel caso 2, la sequenza di cifre che si ripete infinitamente è detta *periodo*, e la sua *lunghezza* q è data dal numero di cifre decimali che vi compaiono. Nel caso esemplificato, il periodo ha lunghezza $q = 6$.

Senza scendere ulteriormente nei dettagli, focalizziamo l'attenzione su una frazione razionale *propria*³ con alcuni particolari numeri primi al denominatore, ossia k/p (con $1 \leq k < p$ e p diverso da 2 e da 5). L'operazione consiste in realtà di una serie di divisioni intere, nelle quali si determina ogni singola cifra decimale e il relativo resto. Ma proprio i possibili resti, per definizione finiti e compresi tra 1 e $p - 1$, sono la chiave della periodicità, perché una volta esaurite le possibilità essi iniziano ciclicamente a ripetersi, esattamente come le cifre del quoziente. La differenza tra i vari numeri primi sta solo nella lunghezza del periodo: alcuni di essi, infatti, se posti al denominatore di una frazione sono caratterizzati dall'avere uno sviluppo decimale con periodo di lunghezza pari a $p - 1$, ossia la **massima lunghezza possibile**, e nella divisione compaiono ciclicamente tutti i possibili $p - 1$ valori di resto non nulli. Consideriamo ad esempio i risultati delle frazioni k/p quando $p = 7, 1 \leq k < p$:

¹A rigore, n e d possono essere interi relativi, ossia dotati di segno: $n, d \in \mathbb{Z}$ e il segno della frazione è determinato con la «regola dei segni». Qualora il segno sia complessivamente negativo, ossia quando n e d hanno segno discordi, si antepone alla frazione il segno $-$ che indica una moltiplicazione implicita per -1 . Il resto rimane totalmente invariato ed è completamente irrilevante ai fini del discorso qui affrontato.

²Talora può essere comodo considerare tale successione di cifre come una sequenza infinita definitivamente nulla dopo la i -esima cifra, per trattare omogeneamente i due casi, come d'altronde spesso accade a fini analitici anche in altri contesti (es. partizioni di interi). Tuttavia, dal punto di vista del calcolo numerico siamo semplicemente di fronte ad una espansione decimale finita.

³Si dice propria (o normalizzata, nella letteratura internazionale) una frazione razionale a/b tale che $1 \leq a < b$, la cui espansione decimale è quindi necessariamente minore di 1.

k	k/p
1	0,142857
2	0,285714
3	0,428571
4	0,571428
5	0,714285
6	0,857142

Come si può notare, il periodo ha sempre lunghezza 6 ed è sempre costituito dalle cifre {1, 2, 4, 5, 7, 8} che si presentano solo in diverso ordine. I numeri primi appena descritti sono appunto detti *generatori* dei numeri ciclici ai quali abbiamo accennato sopra, mentre in generale un qualsiasi numero primo arbitrariamente scelto non risponde necessariamente a queste definizioni. La sequenza di questi particolari primi è catalogata in OEIS col numero A001913 come «full repetend primes», quei numeri primi p per i quali 10 è una *radice primitiva* (in modulo p). Vedremo meglio tra breve il significato di tale espressione.

Se invece consideriamo un diverso primo, ad esempio $p = 13$, avremo la situazione seguente valutando la frazione propria k/p :

k	Classe 1	Classe 2
1	0,076923	
2		0,153846
3	0,230769	
4	0,307692	
5		0,384615
6		0,461538
7		0,538461
8		0,615384
9	0,692307	
10	0,769230	
11		0,846513
12	0,923076	

In questo caso, abbiamo due classi di periodi che si alternano [Eck83], e non a caso: $2 = 12/6 = (p - 1)/q$, formula della quale è piuttosto elementare dimostrare la generalizzazione. La lunghezza del periodo, infatti, è sempre un divisore di $p - 1$ e incidentalmente, $p - 1$ è sempre pari.

Purtroppo, però, non esiste un metodo universale per individuare i primi generatori, detti anche primi ciclici. Ad oggi non è neppure noto quanti numeri primi siano ciclici, né se questi ultimi siano effettivamente infiniti. Abbiamo solo una congettura sul rapporto tra primi ciclici e numeri primi, dovuta al grandissimo algebrista Emil Artin (1898 – 1962) [JWW61], legata alla valutazione del prodotto infinito convergente che segue:

$$C = \prod_{i=1}^{\infty} \left[1 - \frac{1}{p_i(1-p_i)} \right] \cong 0,3739558136\dots \quad (1)$$

Tale valore porta il nome di *costante di Artin*: nella formula p_i rappresenta intuitivamente l' i -esimo numero primo.

La definizione di *radice primitiva* a cui accennavamo sopra deriva dall'aritmetica modulare, in particolare dai resti della divisione di potenze successive crescenti del 10 per il primo considerato (ciò che come suggerito costituisce implicitamente il meccanismo della divisione quando calcoliamo il reciproco di tale numero primo). In aritmetica modulare, un numero naturale n è una radice primitiva modulo p (nel caso particolare di p primo) se ogni numero inferiore a p^4 ⁴ è congruente a una potenza di n in modulo p . Tornando all'esempio di $p = 7$, avremo infatti:

⁴Nel caso generale, si parla invece di ogni numero coprimo con p .

$$\begin{aligned}
10^1 &\equiv 3 \pmod{7} \\
10^2 &\equiv 2 \pmod{7} \\
10^3 &\equiv 6 \pmod{7} \\
10^4 &\equiv 4 \pmod{7} \\
10^5 &\equiv 5 \pmod{7} \\
10^6 &\equiv 1 \pmod{7} \\
10^7 &\equiv 3 \pmod{7} \\
&\dots
\end{aligned}$$

I possibili resti [1, 6] ossia tutti i naturali non nulli inferiori a p compaiono nelle prime sei posizioni, poi iniziano a ripetersi ciclicamente. Anche con questo approccio si ripete, intuitivamente, quanto avviene (con segni opposti degli esponenti) nel calcolo del reciproco. Per questo motivo si dice che i numeri primi ciclici, come il 7, hanno come radice primitiva il 10 poiché ne dividono le potenze successive con tutti i possibili resti, anche qui con un periodo pari a $p - 1$.

1.1 Numeri ciclici.

In breve, si tratta di numeri interi positivi che possiedono alcune interessanti proprietà:

- Sono *generati* da un numero primo ciclico, secondo la formula:

$$C_b(p) = \frac{b^{p-1} - 1}{p} \quad (2)$$

ove b è la base numerica prescelta e p è il numero primo dato, che non sia un divisore della base: $\text{MCD}(b, p) = 1$. Non tutti i numeri primi generano un numero ciclico, come spiegato appena sopra;

- Sono caratterizzati dall'essere sempre espressi con $p - 1$ cifre, il che spiega la loro strettissima correlazione con la rappresentazione e quindi la *base numerica* prescelta. Esiste peraltro un teorema che sancisce l'inesistenza di numeri ciclici in quelle basi che siano quadrati perfetti, $b = a^2$ per qualche naturale $a > 1$;

- Quando moltiplicati per il proprio primo generatore, danno come risultato una sequenza di cifre identiche, tutte pari a $b - 1$;

- Se moltiplicati per altri interi, presentano invece sempre esattamente le medesime cifre del valore di partenza, ma in un diverso ordine. In sostanza, generano delle permutazioni cicliche delle loro stesse cifre.

La sequenza dei numeri ciclici è classificata in OEIS con il codice A180340. Ecco i primi cinque, con indicati i relativi primi generatori:

$$\begin{aligned}
&142.857, p = 7 \\
&588.235.294.117.647, p = 17 \\
&52.631.578.947.368.421, p = 19 \\
&434.782.608.695.652.173.913, p = 23 \\
&344.827.586.206.896.551.724.137.931, p = 29
\end{aligned}$$

Questi pochissimi esempi sono sufficienti per osservare il ritmo sbalorditivo al quale le cifre di questi numeri (rispettivamente 6, 16, 18, 22, 28...) crescono, come indica la correlazione tra il loro logaritmo decimale e $p - 1$. Il più noto di questi numeri, in base dieci, è 142857, generato appunto dal numero sette (da cui, ovviamente, il titolo). Si tratta anche del più piccolo numero ciclico in tale base.

$$\begin{aligned}
142857 \cdot 1 &= 142857 \\
142857 \cdot 2 &= 285714 \\
142857 \cdot 3 &= 428571 \\
142857 \cdot 4 &= 571428 \\
142857 \cdot 5 &= 714285 \\
142857 \cdot 6 &= 857142 \\
142857 \cdot 7 &= 999999
\end{aligned}$$

Si osservi con attenzione lo schema delle permutazioni. Spezziamo in due gruppi di tre cifre ciascun risultato ed attribuiamo un nome simbolico a ciascuno di tali gruppi:

$$\begin{aligned} A &= 142, B = 857 \\ C &= 285, D = 714 \\ E &= 428, F = 571 \end{aligned}$$

Si ha quindi il seguente schema notevole, nel quale ciascuna coppia alla locazione i si inverte alla locazione $p - i$, per $1 \leq i \leq 3$:

$$\underbrace{142}_{A} \underbrace{857}_{B}, \quad \underbrace{285}_{C} \underbrace{714}_{D}, \quad \underbrace{428}_{E} \underbrace{571}_{F}, \quad \underbrace{571}_{F} \underbrace{428}_{E}, \quad \underbrace{714}_{D} \underbrace{285}_{C}, \quad \underbrace{857}_{B} \underbrace{142}_{A}$$

Risulta anche evidente che l'intera sequenza risulta palindroma. Per renderci meglio conto delle ulteriori conseguenze di questa peculiare proprietà, consideriamo il numero di partenza 142857 e scriviamo le sue due metà A e B adeguatamente incolonnate:

$$\begin{matrix} 1 & 4 & 2 \\ 8 & 5 & 7 \end{matrix}$$

Sommendo le cifre $a_1 a_2 \cdots a_{2k}$, $2k = p - 1$ in posizione corrispondente $a_i + a_{k+i}$ per $1 \leq i \leq k$, si ottiene sempre 9: nell'esempio, $1+8 = 4+5 = 2+7 = 9$. Ciò vale, come evidenzia l'esempio, anche per tutti i multipli, eccetto ovviamente quelli caratteristici dati dal primo generatore e suoi multipli. Si tratta di una conseguenza del teorema di Midy, un contributo ottocentesco alla teoria del numero (parzialmente dimenticato per qualche decennio) recentemente riscoperto, con delle interessanti generalizzazioni [Gin04, Lew06].

Infine, alla luce di quanto richiamato in introduzione, osserviamo senza ulteriori considerazioni cosa succede passando ai reciproci:

$$\frac{1}{7} = 0, \overline{142857} \quad \frac{1}{142857} = 0, \overline{000007}$$

2 Due semplici esempi applicativi.

Il nostro divertimento informatico, in questa occasione, consiste in un paio di esempi in Python e C (necessariamente accoppiato ad una famosa libreria per il calcolo intero a precisione arbitraria, MIRACL) per fare un po' di number crunching casalingo. L'idea è quella di generare un po' di numeri ciclici in base $b = 2$, per poi osservarli con tutta calma. Il cinquantesimo di codesti "numeretti" richiede 660 bit (ovvero 21 qword a 32 bit) per essere espresso. Più in generale, dal diciottesimo numero binario ciclico in poi la dimensione è proibitiva per un trattamento con i tipi di default e gli strumenti standard dei linguaggi tradizionali, anche sulla più potente delle attuali workstation a 64 bit.

```
#!/usr/local/bin/python

##
## Tabella dei numeri primi inferiori a 1000: viene utilizzata per verificare
## rapidamente se un primo dato risulta generatore di un numero ciclico.
##
## Lavorando in base due, l'unico numero primo pari e' stato
## opportunamente omesso.
##

Primi = [ 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
        49, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
        109, 113, 121, 127, 131, 137, 139, 143, 149, 151, 157, 163, 167, 169,
        173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241,
        251, 257, 263, 269, 271, 277, 281, 283, 289, 293, 307, 311, 313, 317,
        323, 331, 337, 347, 349, 353, 359, 361, 367, 373, 379, 383, 389, 397,
        401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479,
        487, 491, 499, 503, 509, 521, 523, 529, 541, 547, 557, 563, 569, 571,
        577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653,
        659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751,
        757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 841,
        853, 857, 859, 863, 877, 881, 883, 887, 899, 907, 911, 919, 929, 937,
        941, 947, 953, 961, 967, 971, 977, 983, 991, 997]
```

```

print("Abbiamo a disposizione", len(Primi), "primi" \
      " come potenziali generatori ciclici.")

## Si specifica la base numerica di lavoro
base = 2

## Totalizzatore dei numeri ciclici verificati
tot_cyclics = 0

for p in Primi:
    t = 0
    r = 1
    cyclic = 0

    while 1:
        t += 1
        x = r * base
        d, r = divmod(x, p)
        cyclic = cyclic * base + d
        ##print t, d, r, cyclic
        if r == 1:
            break

    if t == p - 1:
        tot_cyclics += 1
        print("*****")
        print("# Primo generatore = {0:3d} -> il corrispondente numero ciclico" \
              " ha {1} cifre in base {2}".format(p, t, base))
        fmt = "{" + "0:0{0}b".format(t) + "}"
        print(fmt.format(cyclic))

print("\n*****")
print("Sono stati elaborati i primi {0} numeri primi dispari consecutivi.".format(
    (len(Primi))))
print("per generare i primi {0} numeri ciclici in base {1}.".format(tot_cyclics,
    base))
## EOF: Cyclic.py ##

```

Si apprezzeranno la linearità e compattezza dell'esempio Python, nonostante le dimensioni dei numeri trattati. Per confronto, si analizzi il codice C necessario per utilizzare una libreria dedicata:

```

/*
**
** Codice di esempio per la ricerca di numeri ciclici.
** Fa uso della nota libreria MIRACL.
**
** Compilato con Borland C/C++ 5.5.1
**
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mem.h>
#include "miracl.h"

/*
** Chiediamo alla libreria di pregenerare tutti i numeri primi
** non superiori al valore qui definito
*/

```

```

#define PRIME_CEILING 50000

/*
** Ci accontentiamo di cercare i primi MAX_CYCLICS numeri ciclici
*/
#define MAX_CYCLICS 50

/*
** Dimensione, in bytes, riservata ad ogni singola variabile intera
** Inutile esagerare, per il momento
*/
#define MIRACL_SIZE 500

/*
** Stabiliamo un limite pratico al numero di cifre che intendiamo
** mostrare a video (o redirigere su file)
*/
#define MAX_DIGITS 80

/*
** Abilitando la seguente costante, si stampano TUTTI i multipli
** del numero ciclico appena trovato, per mostrarne le proprietà
** combinatorie
*/
//#define SHOW_MULTIPLES

void display_num(big n, int wid, miracl *mip) {
    cotstr(n, mip->IOBUFF);

    if (strlen(mip->IOBUFF) < (unsigned)wid - 1) {
        int ds;
        char *outs;

        ds = wid - strlen(mip->IOBUFF);
        outs = (char *)malloc(ds);

        if (NULL == outs) {
            fputs("## Errore di allocazione ! ##\n", stderr);
            exit(1);
        }

        —ds;
        memset(outs, '0', ds);
        outs[ds] = '\0';
        printf("%s%s\n", outs, mip->IOBUFF);
        free(outs);
    } else {
        printf("%s\n", mip->IOBUFF);
    }
}

int main(void) {
    /*
    ** Poniamo in questa variabile la base numerica nella quale
    ** vogliamo lavorare: la più "informatiche" possibile è
    ** ovviamente la base binaria.
    ** D'altro canto, in esadecimale non esistono numeri ciclici,
    ** per un noto teorema che esclude le basi che siano quadrati
    ** perfetti.
    */
    unsigned int base;
}

```

```

/*
** Totalizzatore per i numeri ciclici , avrà come limite la
** pseudocostante di preprocessore denominata MAX_CYCLICS
*/
unsigned int total;

/* Varie ed eventuali */
unsigned int i, r, t;

/* Ecco i numeri interi davvero grandi ! */
big mx, x1, x2;
miracl *mip;

total = 0;
i = 0;

/*
** Lavoriamo in base binaria. Per utilizzare altre
** basi, occorre "filtrare" diversamente
** il nostro array di numeri primi potenziali generatori ,
** escludendo i divisori della base scelta.
** D'altro canto, si ricordi che in esadecimale
** non esistono numeri ciclici , per un noto teorema
** che ne esclude la presenza in basi che siano quadrati perfetti .
*/
base = 2;

mip = mirsys(MIRACL_SIZE, 10);
mx = mirvar(0);
x1 = mirvar(0);
x2 = mirvar(0);

mip->IOBASE = base;
gprime(PRIME_CEILING);

while (total < MAX_CYCLICS) {
    i++;

    if (mip->PRIMES[i] == 0)
        break;

    t = 0;
    r = 1;

    zero(mx);
    zero(x2);

    /*
    ** Implementazione di un algoritmo pedestre , ma efficace
    */
    do {
        ++t;
        convert(r * base, x1);
        r = subdiv(x1, mip->PRIMES[i], x2);
        premult(mx, base, mx);
        add(x2, mx, mx);
    } while (r != 1);

    /*
    ** Se il test seguente è TRUE, abbiamo un numero ciclico

```

```

/*
if (t == (unsigned)mip->PRIMES[ i ] -1) {
    int j;
    fputs("*****\n", stdout);

    ++total;
    printf("# primo generatore = %d -> il corrispondente numero "
           "ciclico ha %d cifre in base %d\n",
           mip->PRIMES[ i ], mip->PRIMES[ i ] -1, base);

    if (MAX_DIGITS > mip->PRIMES[ i ]) {
        display_num(mx, mip->PRIMES[ i ], mip);
    } else {

        puts("Sorry, too big to print... ");
    }
}

#endif
}

printf("\n*****\n"
      "Sono stati testati i primi %d numeri primi dispari consecutivi\n"
      "per generare i primi %d numeri ciclici in base %d.\n\n",
      i, total, base);

return EXIT_SUCCESS;
}
/** EOF: cyclic.c */

```

Il codice è scritto nel modo più didattico e stimolante possibile, con ampiissimo spazio per modifiche e personalizzazioni. Quasi inutile rimarcare che esistono algoritmi maggiormente efficienti, e che questo genere di ricerche viene in realtà quasi sempre svolto tramite applicativi di calcolo numerico dedicati.

Attenzione alle costanti di preprocessore definite in testa al sorgente: sono interrelate in modo piuttosto sottile, dunque non cambiatele senza aver meditato (e letto la documentazione di MIRACL). Il codice d'esempio viene fornito "as is" e l'unica garanzia è che, una volta debitamente compilato, l'eseguibile occuperà un po' di spazio sui vostri supporti di massa.

Riferimenti bibliografici

- [Eck83] Michael W. Ecker, *The alluring lore of cyclic numbers*, The Two-Year College Mathematics Journal **14** (1983), no. 2, 105–109.
- [Gin04] Brian Ginsberg, *Midy's (nearly) secret theorem: An extension after 165 years*, The College Mathematics Journal **35** (2004), 26.
- [JWW61] Jr. John W. Wrench, *Evaluation of artin's constant and the twin-prime constant*, Mathematics of Computation **15** (1961), no. 76, 396–398.
- [Lew06] Joseph Lewittes, *Midy's theorem for periodic decimals*, 2006.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

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 (donda 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** $\mathbb{N} = \{0, 1, 2, 3, \dots\}$, 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 stabilità tra i vari enti matematici¹. 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 $f_S : A \rightarrow \{0, 1\}$ definita come segue per ogni $a \in A$:

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

¹ Per un arciroto 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 $A = \{a_0, a_1, \dots, a_{n-1}\}$, una volta fissata un’applicazione biunivoca arbitraria (di norma immediata) tale per cui $a_0 \leftrightarrow 0, a_1 \leftrightarrow 1, \dots$, esso è isomorfo all’insieme $A_N = \{0, 1, 2, \dots, n - 1\}$ 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):

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

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.

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 riduce² 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 ubiqi: 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 è 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*)

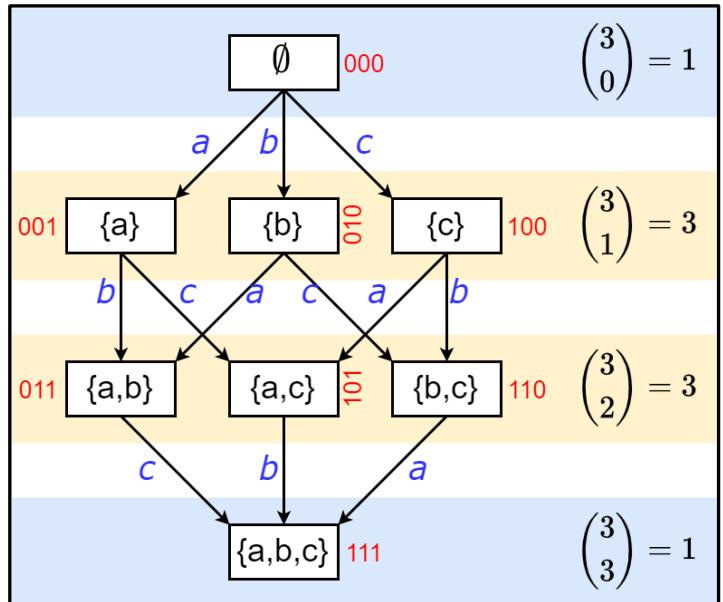


Figura 1: diagramma di Hasse arricchito dell'insieme potenza per $A=\{a,b,c\}$.

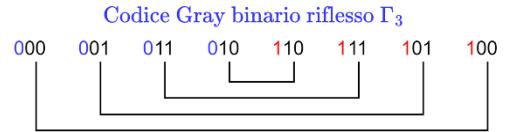
² A patto, ovviamente, che non vogliamo complicarci "inutilmente" la vita con ordinamenti particolari, restrizioni sulla dimensione dei subset, algoritmo del banchiere e altre amenità, comunque secondarie rispetto al concetto qui espresso.

$\wp(A)$ 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	\emptyset	{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:

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

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 è un semplice esempio tabulare per i primi valori di n chiariranno definitivamente tale modalità costruttiva.

$$\begin{aligned} \Gamma_0 &= \text{''''} \\ \Gamma_1 &= \textcolor{blue}{0}, \textcolor{red}{1} \\ \Gamma_2 &= \textcolor{blue}{00}, \textcolor{red}{01}, \textcolor{red}{11}, \textcolor{red}{10} \\ \Gamma_3 &= \textcolor{blue}{000}, \textcolor{red}{001}, \textcolor{red}{011}, \textcolor{blue}{010}, \textcolor{red}{110}, \textcolor{red}{111}, \textcolor{blue}{101}, \textcolor{blue}{100} \end{aligned}$$

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:

$$\Gamma[n] = n \oplus (n \gg 1) \quad (4)$$

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 $n > 0$:

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

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

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 $0 \rightarrow 1$, sovente indicata con il simbolo \uparrow in vari contesti applicativi;
3. **Il numero di bit** che si *invertono* complessivamente (indipendentemente dal verso della transizione: $0 \rightarrow 1$ o $1 \rightarrow 0$) 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 $1 \dots n$:

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

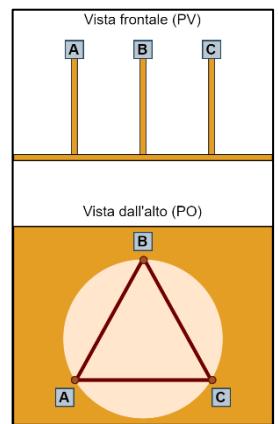
3 Le regole del gioco.

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 $\phi_1 < \phi_2 < \dots < \phi_n$, 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:

$$M(n) = 2^n - 1 \quad (7)$$

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.

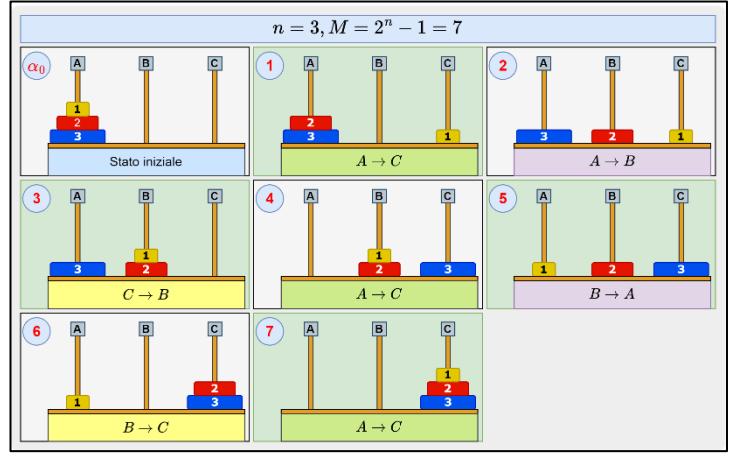


Figura 2: Sequenza risolutiva per $n=3$.

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),... 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.

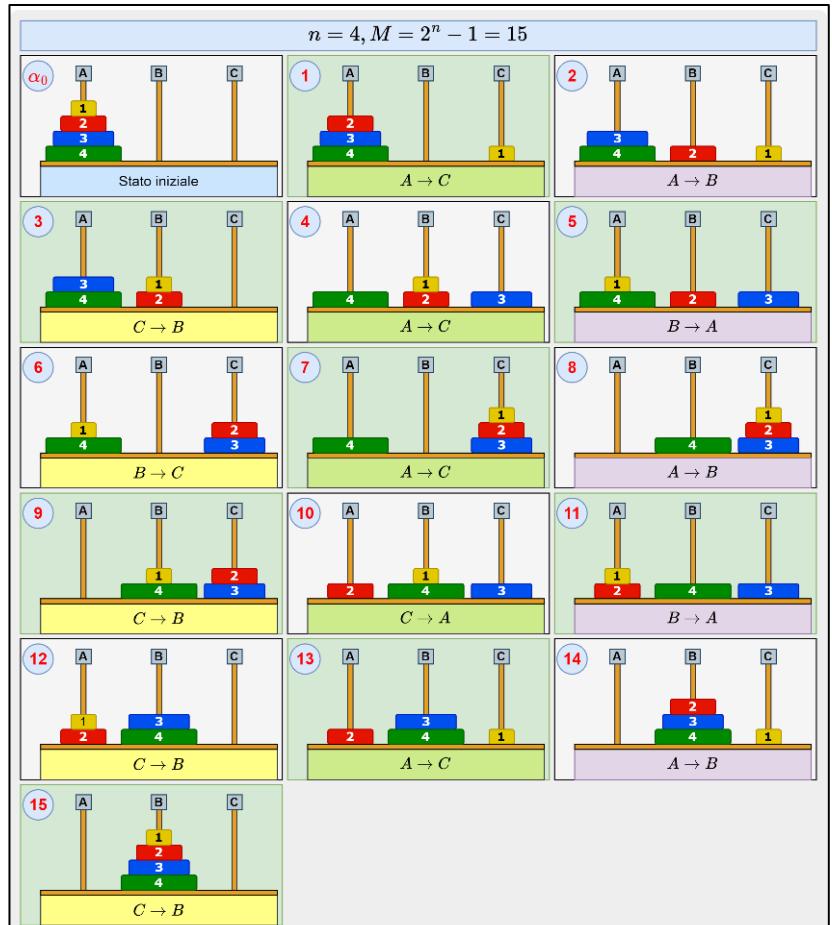


Figura 3: Sequenza risolutiva per $n=4$.

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, ... 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:

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

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:

$$Src = (m \wedge (m - 1))(mod\ 3) \quad (9)$$

$$Dest = (m \vee (m - 1) + 1)(mod\ 3) \quad (10)$$

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:

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 $D[i]$** 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

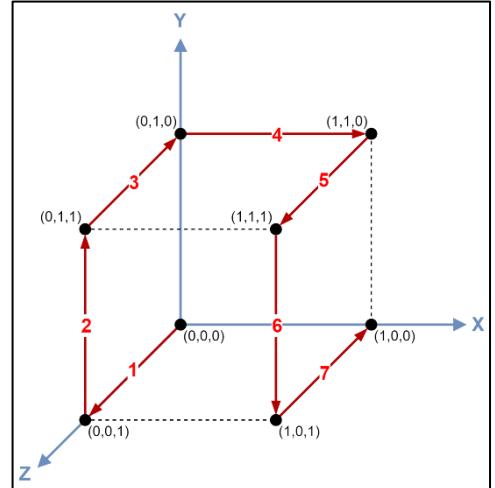


Figura 4: cammino hamiltoniano in un cubo booleano, si notino le coordinate binarie.

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, \dots, 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 algoritmo 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 controsenso 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"
38.           "*** Mosse.....: %zd\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"
73.             "## Mosse.....: %zd\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, Alessandro 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 prudentiale, 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 prudentiale 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, probabilmente 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 programmati 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.pdf>.
- 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.

© Copyright 1984-2024 by M.A.W. 1968



Quest'opera viene rilasciata con licenza Creative Commons **Attribuzione - Non commerciale - Condividi allo stesso modo** 4.2 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Il tifoso, l'arbitro e il calciatore

Sommario

Prendo a prestito il titolo di una nota commedia all'italiana (Pier Francesco Pingitore, 1982) perché in questo breve articolo si parla di come creare, in generale, calendari d'incontri e tornei. Strumenti utili in vari frangenti, dallo studio del comportamento sociale di volatili e mammiferi alla teoria elettorale, come pure nello sport, dalle boccette agli scacchi passando appunto per palle e palloni di varia foggia. Sappiamo però per lunga esperienza che, ogni volta che si sfiora l'argomento, il lettore medio pensa subito invariabilmente al pallone e al campionato all'italiana. Da qui, ovviamente, il titolo!

1 Introduzione.

Il materiale qui presentato costituisce una riorganizzazione di quanto già discusso sullo storico *blog* dell'autore «Titolo provvisorio...». Ne mantiene volutamente inalterati sia il tono colloquiale e didascalico che l'impostazione strettamente divulgativa.

Sebbene esista perfino una fondamentale monografia [And] completamente dedicata a tornei e affini, oltre ad interi capitoli specifici in testi di combinatorica e combinatorial designs dedicati alle matrici torneo e dintorni, tali raffinati metodi sono normalmente ignorati dai programmati, che sembrano invece molto propensi ad imbattersi in versioni modificate del metodo deterministico più elementare per la generazione di tali oggetti combinatori.

Tale metodo gode di insolita popolarità, anche se porta il nome dell'oscuro scacchista austriaco Johann Nepomuk Berger. Già l'attribuzione del nome è comunque filologicamente errata: fonti autorevolissime della storia degli scacchi (in primis la Bibbia della storia scacchistica, [Hoo]) riconducono l'idea originale al non meno sconosciuto Richard Schurig, il quale in definitiva ha semplicemente pubblicato una serie di tavole precalcolate per vari numeri di giocatori nel 1886 sul Deutsche Schachzeitung. L'altro illustre sconosciuto, appunto Berger, le avrebbe poi solo riproposte tali e quali nel suo Schachjahrbucher del 1892-93.

Purtroppo l'attribuzione del nome non è l'unico errore che accompagna questo algoritmo nelle varie fonti online e sui forum, ma di certo è il più veniale. Per questo, prima di passare alle costruzioni più esotiche o scavare nella teoria algebrica e combinatorica delle matrici torneo, vogliamo proporre qui una descrizione corretta ed una implementazione di riferimento che funga da punto di partenza per attrarre interesse sul tema e stimolare ulteriori approfondimenti.

2 L'algoritmo di Berger (anzi, di Schurig).

Il cosiddetto "algoritmo di Berger" si trova così erroneamente referenziato un po' ovunque in letteratura, sebbene codesto signore come già anticipato non l'abbia in effetti inventato, e soprattutto non l'abbia esposto in forma di algoritmo ma solo come semplici tavole precompilate.

Le idee sottese sono antichissime, in particolare il concetto di *cerniera*. Il tipo di tornei che qui ci interessano è detto "round robin", esattamente come l'algoritmo di scheduling più elementare noto dalla teoria dei sistemi operativi multitasking, ben conosciuto da chiunque abbia affrontato un minimo di studi istituzionali in informatica¹.

La storia di questa idea, di per sé, sarebbe talmente densa ed interessante da riempire un intero tomo. Qui ci limitiamo all'ovvio e all'arcinoto, citando per l'ennesima volta l'apoftegma del piccolo Gauss alle prese con la somma dei primi cento numeri naturali. Una buona vulgata vuole che egli li scrivesse nella seguente guisa, traendo poi le opportune conclusioni dalle somme di ciascuna coppia verticale:

1	2	3	...	49	50
100	99	98	...	52	51
101	101	101	...	101	101

¹Se nell'uno o nell'altro caso qualche lettore fosse stato punto da vaghezza di sapere chi diavolo fosse mai questo signor Robin, si consoli: l'espressione non ha alcuna attinenza con un nome proprio e deriva semplicemente, a dispetto di molte fonti male informate, dalla corruzione anglofona del francese *ruban* che significa "nastro". Dunque si parla di un **nastro circolare**, il che è la perfetta descrizione concettuale di quel che accade: un ordine lineare, finito e limitato (la lista ordinata dei partecipanti) si trasforma - con una operazione tra le più tipiche e paradigmatiche del pensiero discreto - in un ordine circolare, periodico, ovvero finito ma non limitato, semplicemente unendo gli estremi.

Da qui l'arcinota formula, che modernamente scriviamo:

$$\sum_{j=1}^n j = \frac{n(n+1)}{2} \quad (1)$$

Tale versione dell'aneddoto risulta forse meno accurata storicamente, ma massimamente "informatichese" in quanto realizza una pratica economia di risorse: in questa versione, infatti, ciascun numero viene scritto una e una sola volta. Quel che conta qui è comunque il concetto: nella lista così "riplegata" ciascun numero "incontra" il suo omologo, come i denti delle due metà di una cerniera lampo si incastrano perfettamente gli uni negli altri quando accostati. Ciò spiega la denominazione, sovente adottata in letteratura, di algoritmo a "cerniera" per tutti i tipi di approccio derivati da questa idea. Interessante notare che la somma di ciascuna coppia di numeri in verticale è costante, ma le differenze indicano che tali coppie formano un completo **sistema di differenze**, che è il reale motivo algebrico per cui tale costruzione funziona.

L'algoritmo in pratica è tutto qui: si fissa arbitrariamente un pivot, e per ogni giornata si fanno scorrere in un verso stabilito tutti gli altri giocatori, in genere di una posizione (in realtà, si dimostra banalmente che qualsiasi valore di scorrimento relativamente primo al numero di partecipanti diminuito di una unità genera un calendario valido). In questo modo, per costruzione, ciascun giocatore incontra il pivot una e una sola volta per ogni serie completa di scorrimenti (che riporta la lista esattamente nella posizione iniziale), e meccanicamente ne consegue che ogni squadra incontra una e una sola volta tutte le altre, per l'ovvia corrispondenza posizionale stabilita dalla "cerniera". Supponendo ad esempio di avere 8 squadre, numerate da 1 a 8, avremo per la prima giornata i seguenti accoppiamenti, da leggersi in verticale, per colonne:

$$\begin{array}{cccc} \color{red}1 & 2 & 3 & 4 \\ 8 & 7 & 6 & 5 \end{array} \text{ ovvero } 1 - 8, 2 - 7, 3 - 6, 4 - 5$$

Procedendo al primo scorrimento in senso orario e mantenendo la prima squadra come pivot, avremo per la seconda giornata:

$$\begin{array}{cccc} \color{red}1 & 8 & 2 & 3 \\ 7 & 6 & 5 & 4 \end{array} \text{ ovvero } 1 - 7, 8 - 6, 2 - 5, 3 - 4$$

e così via per le giornate successive, fino a ripetere la configurazione iniziale dopo 8 rotazioni.

Per aggiungere un po' di spezie, è poi possibile inserire qualche banale euristica preliminare, come rimescolare la lista dei partecipanti o scegliere in modo pseudocasuale verso e ampiezza delle rotazioni (da una lista di possibili coprimi), al solo fine di ottenere calendari leggermente variati ad ogni esecuzione, mitigando la rigidità strutturale dell'algoritmo originale.

Resta essenziale ribadire il concetto portante: è proprio in tal modo che **fisicamente** si svolgono molti tipi di torneo. Ad esempio, in quelli di dama e scacchi, i giocatori (tranne uno) al termine di una partita scalano fisicamente di posto attorno ad un lungo tavolo, o attorno ad una serie di singoli tavolini allineati, il che è quanto avviene peraltro anche nel ping pong ed in molti altri sport analoghi.

3 Un semplice esempio in C.

Il sorgente che segue, ampiamente commentato, dimostra tutti i concetti fondamentali e le immediate varianti del banale algoritmo in questione, la cui assoluta superiorità prestazionale e concettuale rispetto ad altre bislacche soluzioni (purtroppo molto diffuse) è assolutamente palese.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "cal_berger.h"

#define MAX_TEAMS 32U

typedef unsigned int uint;
typedef enum {FALSE, TRUE} Boole_t;

uint Squadre[MAX_TEAMS];
uint Calendario[MAX_TEAMS][MAX_TEAMS];

/****************/
/****************/
```

```

void shuffle( uint Array[], const size_t Size) {
    size_t i;

    for (i = Size; i > 1; —i) {
        register unsigned int t;
        size_t p = rand() % i;

        /* SWAP(Array[k-1], Array[rand(k)]) */
        t = Array[p];
        Array[p] = Array[i - 1];
        Array[i - 1] = t;
    }
}

int main() {
    size_t NumSquadre = 10;
    size_t pivot, Giornate;
    size_t Shift;
    size_t i, first, temp;

    randomize();

    /*
     ** Primo passo: si genera una permutazione pseudocasuale delle
     ** squadre, per introdurre un po' di nondeterminismo rispetto
     ** all'ordine naturale.
    */
    /* shuffle(Squadre, NumSquadre); */
    for (i = 0; i <= NumSquadre; ++i) {
        Squadre[i] = i;
    }

    shuffle(Squadre, NumSquadre);

    /*
     ** Secondo passo: si sceglie in modo pseudocasuale anche la
     ** squadra pivot, ossia il punto fisso richiesto dall'algoritmo
     ** di Berger.
    */
    pivot = rand() % NumSquadre;

    /*
     ** La squadra pivot viene posizionata al termine all'array.
     ** Se il numero di squadre viene immesso dall'utente, in caso di
     ** valore dispari si aggiungerà invece in tale posizione una squadra
     ** fittizia, codificata in modo univoco (es. 99), che rappresenta il
     ** "Riposo". Chi "incontra" tale squadra pivot, evidentemente, salta
     ** il turno.
    */
    /*
     ** In questo modo le squadre saranno sempre e comunque in numero pari.
    */
    temp = Squadre[pivot];
    Squadre[pivot] = Squadre[NumSquadre - 1];
    Squadre[NumSquadre - 1] = temp;
}

```

```

** Si noti che qui il valore viene diminuito di una unità, per
** semplificare i riferimenti in tutto il resto del programma.
*/
NumSquadre;
Giornate = NumSquadre;
first = 0;

/*
** Terzo passo: le rotazioni richieste dall'algoritmo non sono
** limitate ad un passo unitario. Si veda anche il file coprimes.h.
** Si sceglie quindi un valore per la rotazione: sono ammessi, oltre
** all'unità, tutti i numeri coprimi con NumSquadre -1.
** Attenzione al peculiare indirizzamento dell'array dei coprimi.
*/
Shift = coprimes[(NumSquadre -2) >> 1][rand() %
    coprimes[(NumSquadre -2) >> 1][NUM_COPRIMES]];

if (rand() & 1) {
    /*
    ** Come ulteriore elemento di dinamizzazione, si sceglie anche il
    ** verso di rotazione in modo pseudocasuale. La sostanziale
    ** (e non banale) simmetria dei coprimi inferiori ad un valore
    ** dato garantisce che, se il naturale  $1 < k < n$  è coprimo a  $n$ ,
    ** allora lo sarà anche  $n - k$ .
    */
    Shift = NumSquadre - Shift;
}

printf("Squadre ..... %d\nPivot ..... %d\n"
    "Shift ..... %d\n",
    NumSquadre +1, Squadre[NumSquadre], Shift);

/*
** Implementazione dell'algoritmo "a cerniera".
** La squadra pivot incontra, a rotazione, tutte le altre.
** Questo vincolo viene esteso meccanicamente alle altre coppie.
*/
for (i = 0; i < Giornate; ++i) {
    size_t id1;
    int ofs;

    printf(">> Giornata %2d *****\n", i +1, Squadre[NumSquadre], Squadre[first]);

    Calendario[Squadre[NumSquadre]][Squadre[first]] = i +1;
    Calendario[Squadre[first]][Squadre[NumSquadre]] = i +1;

    for (id1 = (first +1) % NumSquadre, ofs = NumSquadre -2;
        ofs > 0; ofs -= 2) {
        size_t id2 = (id1 + ofs) % NumSquadre;

        printf(", %d-%d", Squadre[id1], Squadre[id2]);

        Calendario[Squadre[id1]][Squadre[id2]] = i +1;
        Calendario[Squadre[id2]][Squadre[id1]] = i +1;
        id1 = (++id1) % NumSquadre;
    }

    puts("");
    first = (first + Shift) % NumSquadre;
}

```

```

    puts("*****\n");
    /*
    ** Fine lavoro. Stampa della matrice torneo.
    */
    for (i = 0; i <= NumSquadre; ++i) {
        size_t j;

        for (j = 0; j <= NumSquadre; ++j) {
            printf("%2d ", Calendario[i][j]);
        }
        puts("");
    }

    return EXIT_SUCCESS;
}
/* EOF: cal_berger.c */

```

Il sorgente proposto fa riferimento ad uno header, contenente tra l'altro una utilissima tabella precalcolata che elenca (in questo caso e per nostra comodità limitatamente ai soli naturali dispari della forma $n = 2m+1$, $m \in \mathbb{N}$) tutti i naturali non nulli inferiori ad n e relativamente primi ad esso: $\{a \mid 1 \leq a \leq n \wedge MCD(a, n) = 1\}$ ², nonché il loro numero ovvero la funzione totiente di Euler $\varphi(n)$ (si veda anche la OEIS A000010).

```

#ifndef __COPRIMES_H__
#define __COPRIMES_H__

#ifndef uchar_t
    typedef unsigned char uchar_t;
#endif

#ifndef uint_t
    typedef unsigned int uint_t;
#endif

/*
** Limite arbitrario (ma del tutto ragionevole)
** per i valori trattabili.
*/
#define MAX_VAL 32

/*
** Le rotazioni valide per la creazione di un calendario di incontri
** non sono limitate a quella unitaria, ma includono tutti e soli i valori
** coprimi a  $n-1$ , dove  $n$  è il numero delle squadre. I vincoli della
** implementazione garantiscono per costruzione che  $n-1$  sia sempre dispari,
** dunque occorre e basta limitarsi a siffatti valori nella costruzione della
** tabella dei coprimi.
**
** L'ultima locazione di ogni riga riporta il numero di coprimi inferiori ad  $n$ :
** abbiamo qui tabulato la funzione totiente di Euler  $\varphi(n)$  per piccoli valori
** di  $n$ , tutti dispari.
**
** La rappresentazione scelta obbedisce evidentemente a criteri di massima
** efficienza nella selezione pseudorandom a runtime, piuttosto che a istanze

```

²Da un punto di vista logico, nelle definizioni l'inclusione dell'estremo superiore dell'intervallo $[1, n]$ è a rigore del tutto inutile tranne nel caso singolare $n = 1$. La condizione logica $MCD(n, n) = 1$ è infatti definitivamente falsa per ogni $n \geq 2$. In letteratura tale notazione ridondante viene adottata principalmente per brevità e omogeneità di trattamento, come in molti altri casi analoghi, invece di definire esplicitamente il valore della funzione nel caso limite $\varphi(1)$ a margine. Interessante invece rilevare che nella progettazione di ambienti e applicazioni di calcolo risulta quasi sempre necessario definire esplicitamente anche $\varphi(0)$, tipicamente come $\varphi(0) = 0$.


```

    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 0, 0},
/* 30 */ { 8, 1, 7, 11, 13, 17, 19, 23, 29, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/* 31 */ {30, 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},
/* 32 */ {16, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0} };

#endif
/* EOF: coprimes.h */

```

Tutto qui: semplice, elegante, efficace e soprattutto efficiente.

Nota conclusiva: il calendario così generato è ovviamente dimostrativo e generico, ergo non tiene conto delle peculiarità di questo o quello sport (ad esempio l'alternanza tra partite "in casa" e non, che comunque è poco più che una questione di stampa), perché qui non svolgiamo i compiti per casa di nessuno, ma è perfetto per illustrare i concetti dell'algoritmo e le sue più immediate varianti, per fare in modo che vi siano sempre piccole differenze nei calendari generati di volta in volta.

Ecco qui l'output di una semplice sequenza di esecuzioni consecutive, in cui si può apprezzare la continua variazione dei calendari generati.

```

Squadre ..... 10
Pivot ..... 1
Shift ..... 8
>> Giornata 1 *****
2-3, 6-10, 1-8, 5-4, 9-7
>> Giornata 2 *****
2-10, 3-8, 6-4, 1-7, 5-9
>> Giornata 3 *****
2-8, 10-4, 3-7, 6-9, 1-5
>> Giornata 4 *****
2-4, 8-7, 10-9, 3-5, 6-1
>> Giornata 5 *****
2-7, 4-9, 8-5, 10-1, 3-6
>> Giornata 6 *****
2-9, 7-5, 4-1, 8-6, 10-3
>> Giornata 7 *****
2-5, 9-1, 7-6, 4-3, 8-10
>> Giornata 8 *****
2-1, 5-6, 9-3, 7-10, 4-8
>> Giornata 9 *****
2-6, 1-3, 5-10, 9-8, 7-4
*****
```

0	8	9	6	3	4	2	1	7	5
8	0	1	4	7	9	5	3	6	2
9	1	0	7	4	5	3	2	8	6
6	4	7	0	1	2	9	8	5	3
3	7	4	1	0	8	6	5	2	9
4	9	5	2	8	0	7	6	3	1
2	5	3	9	6	7	0	4	1	8
1	3	2	8	5	6	4	0	9	7
7	6	8	5	2	3	1	9	0	4
5	2	6	3	9	1	8	7	4	0

```

Squadre ..... 10
Pivot ..... 1
Shift ..... 1
>> Giornata 1 *****
2-6, 8-7, 10-5, 3-4, 9-1
>> Giornata 2 *****
2-8, 10-6, 3-7, 9-5, 1-4

```

```

>> Giornata 3 ****
2-10, 3-8, 9-6, 1-7, 4-5
>> Giornata 4 ****
2-3, 9-10, 1-8, 4-6, 5-7
>> Giornata 5 ****
2-9, 1-3, 4-10, 5-8, 7-6
>> Giornata 6 ****
2-1, 4-9, 5-3, 7-10, 6-8
>> Giornata 7 ****
2-4, 5-1, 7-9, 6-3, 8-10
>> Giornata 8 ****
2-5, 7-4, 6-1, 8-9, 10-3
>> Giornata 9 ****
2-7, 6-5, 8-4, 10-1, 3-9
*****

```

0	6	5	2	7	8	3	4	1	9
6	0	4	7	8	1	9	2	5	3
5	4	0	1	6	7	2	3	9	8
2	7	1	0	3	4	8	9	6	5
7	8	6	3	0	9	4	5	2	1
8	1	7	4	9	0	5	6	3	2
3	9	2	8	4	5	0	1	7	6
4	2	3	9	5	6	1	0	8	7
1	5	9	6	2	3	7	8	0	4
9	3	8	5	1	2	6	7	4	0

```

Squadre ..... 10
Pivot ..... 0
Shift ..... 4
>> Giornata 1 ****
1-6, 3-5, 8-10, 9-7, 2-4
>> Giornata 2 ****
1-2, 4-9, 7-8, 10-3, 5-6
>> Giornata 3 ****
1-5, 6-10, 3-7, 8-4, 9-2
>> Giornata 4 ****
1-9, 2-8, 4-3, 7-6, 10-5
>> Giornata 5 ****
1-10, 5-7, 6-4, 3-2, 8-9
>> Giornata 6 ****
1-8, 9-3, 2-6, 4-5, 7-10
>> Giornata 7 ****
1-7, 10-4, 5-2, 6-9, 3-8
>> Giornata 8 ****
1-3, 8-6, 9-5, 2-10, 4-7
>> Giornata 9 ****
1-4, 7-2, 10-9, 5-8, 6-3
*****

```

0	2	8	9	3	1	7	6	4	5
2	0	5	1	7	6	9	4	3	8
8	5	0	4	1	9	3	7	6	2
9	1	4	0	6	5	8	3	2	7
3	7	1	6	0	2	5	9	8	4
1	6	9	5	2	0	4	8	7	3
7	9	3	8	5	4	0	2	1	6
6	4	7	3	9	8	2	0	5	1
4	3	6	2	8	7	1	5	0	9
5	8	2	7	4	3	6	1	9	0

```

Squadre ..... 10
Pivot ..... 7
Shift ..... 8
>> Giornata 1 *****
8-10, 6-3, 9-5, 4-2, 7-1
>> Giornata 2 *****
8-3, 10-5, 6-2, 9-1, 4-7
>> Giornata 3 *****
8-5, 3-2, 10-1, 6-7, 9-4
>> Giornata 4 *****
8-2, 5-1, 3-7, 10-4, 6-9
>> Giornata 5 *****
8-1, 2-7, 5-4, 3-9, 10-6
>> Giornata 6 *****
8-7, 1-4, 2-9, 5-6, 3-10
>> Giornata 7 *****
8-4, 7-9, 1-6, 2-10, 5-3
>> Giornata 8 *****
8-9, 4-6, 7-10, 1-3, 2-5
>> Giornata 9 *****
8-6, 9-10, 4-3, 7-5, 1-2
*****

```

0	9	8	6	4	7	1	5	2	3
9	0	3	1	8	2	5	4	6	7
8	3	0	9	7	1	4	2	5	6
6	1	9	0	5	8	2	7	3	4
4	8	7	5	0	6	9	3	1	2
7	2	1	8	6	0	3	9	4	5
1	5	4	2	9	3	0	6	7	8
5	4	2	7	3	9	6	0	8	1
2	6	5	3	1	4	7	8	0	9
3	7	6	4	2	5	8	1	9	0

```

Squadre ..... 10
Pivot ..... 3
Shift ..... 1
>> Giornata 1 *****
4-8, 9-2, 3-1, 10-5, 7-6
>> Giornata 2 *****
4-9, 3-8, 10-2, 7-1, 6-5
>> Giornata 3 *****
4-3, 10-9, 7-8, 6-2, 5-1
>> Giornata 4 *****
4-10, 7-3, 6-9, 5-8, 1-2
>> Giornata 5 *****
4-7, 6-10, 5-3, 1-9, 2-8
>> Giornata 6 *****
4-6, 5-7, 1-10, 2-3, 8-9
>> Giornata 7 *****
4-5, 1-6, 2-7, 8-10, 9-3
>> Giornata 8 *****
4-1, 2-5, 8-6, 9-7, 3-10
>> Giornata 9 *****
4-2, 8-1, 9-5, 3-6, 10-7
*****

```

0	4	1	8	3	7	2	9	5	6
4	0	6	9	8	3	7	5	1	2
1	6	0	3	5	9	4	2	7	8
8	9	3	0	7	6	5	1	2	4

3	8	5	7	0	2	6	4	9	1
7	3	9	6	2	0	1	8	4	5
2	7	4	5	6	1	0	3	8	9
9	5	2	1	4	8	3	0	6	7
5	1	7	2	9	4	8	6	0	3
6	2	8	4	1	5	9	7	3	0
Squadre 10									
Pivot 3									
Shift 1									
>> Giornata 1 ****									
4-8, 9-2, 3-1, 10-5, 7-6									
>> Giornata 2 ****									
4-9, 3-8, 10-2, 7-1, 6-5									
>> Giornata 3 ****									
4-3, 10-9, 7-8, 6-2, 5-1									
>> Giornata 4 ****									
4-10, 7-3, 6-9, 5-8, 1-2									
>> Giornata 5 ****									
4-7, 6-10, 5-3, 1-9, 2-8									
>> Giornata 6 ****									
4-6, 5-7, 1-10, 2-3, 8-9									
>> Giornata 7 ****									
4-5, 1-6, 2-7, 8-10, 9-3									
>> Giornata 8 ****									
4-1, 2-5, 8-6, 9-7, 3-10									
>> Giornata 9 ****									
4-2, 8-1, 9-5, 3-6, 10-7									

0	4	1	8	3	7	2	9	5	6
4	0	6	9	8	3	7	5	1	2
1	6	0	3	5	9	4	2	7	8
8	9	3	0	7	6	5	1	2	4
3	8	5	7	0	2	6	4	9	1
7	3	9	6	2	0	1	8	4	5
2	7	4	5	6	1	0	3	8	9
9	5	2	1	4	8	3	0	6	7
5	1	7	2	9	4	8	6	0	3
6	2	8	4	1	5	9	7	3	0

La matrice torneo A viene popolata e stampata in modo che se la squadra a incontra la b nella giornata $g - esima$, avremo $A_{a,b} = A_{b,a} = g$. In chiusura, si invita il lettore ad osservare la peculiare simmetria della matrice degli incontri rispetto alla diagonale principale: quel che si ottiene, infatti, è esattamente un **quadrato latino simmetrico**, ossia una matrice $n \times n$ nella quale ciascun naturale $0 \dots n - 1$ compare esattamente una volta per ciascuna riga e colonna, con la peculiarità aggiuntiva che ogni riga i -esima è identica alla colonna i -esima.

Riferimenti bibliografici

- [And] Ian Anderson, *Combinatorial designs and tournaments*, Clarendon Press.
- [Hoo] David Hooper, *The oxford companion to chess*, Oxford University Press.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

Metodi Formali

Nella mia lunga esperienza parallela di formatore ho potuto ripetutamente sperimentare in prima persona l'enorme fatica dell'iniziare a parlare di metodi e linguaggi formali a dei giovani laureati in discipline attinenti l'informatica e l'elettronica.

Nel migliore dei casi qualcuno cita UML o i diagrammi di flusso, altri hanno inteso parlare qualche volta delle triple di Hoare, gli elettronici hanno forse visto in laboratorio qualche applicativo avanzato che sfrutta i BDD per la specifica e verifica di progetti Verilog e VHDL, ma in generale la confusione regna sovrana.

Soprattutto, fatto gravissimo, la maggioranza di questi giovanotti e neoprofessionisti - che abbiano seguito o meno un corso di metodi formali o software engineering - sembra ignorare bellamente l'enorme **potenza dei metodi formali** e la loro capacità di condurre alla produzione di software letteralmente **a prova d'errore**. Per tacere del fatto, troppo spesso ignorato, che tali metodi non sono mera teoria concepita in qualche eburnea turris accademica tanto per pubblicare qualcosa di nuovo, ma sono usati quotidianamente in società di engineering e softwarehouses di ogni dimensione, accomunate da livelli qualitativi estremamente elevati!

Nel migliore dei casi, i messaggi che passano all'epoca degli studi sono delle strane mezze verità del tipo "I metodi formali esistono da decenni, ma solo di recente hanno attirato attenzione" (falso, o comunque limitato a taluni mercati: sono invece ampiamente in uso nelle applicazioni più critiche, e previsti dalle relative normative, fin dai primi anni Novanta, temporalmente a ridosso di alcuni incidenti realmente gravi come quello del Therac-25) o "Alcune parti dei sistemi particolarmente critici vengono specificate tramite linguaggi formali" (falso: questo è solo uno dei possibili profili di applicazione, il più basilare e meno costoso. Ma la stragrande maggioranza delle normative prevede invece che i relativi sistemi siano specificati e verificati **per intero**).

Senza pretese di alcun genere, vorrei con questo breve articolo fare un minimo di chiarezza iniziale, recuperando in buona parte un vecchio post: tanto per cominciare a capire di cosa si parla.

Personalmente ho iniziato a familiarizzare con i metodi formali a fine anni Ottanta, quando ad esempio le logiche temporali erano una novità nel mondo della ricerca per questo tipo di applicazione. Ho avuto la fortuna di fare dei *formal methods* una delle mie specializzazioni nel momento più fecondo della loro storia e di poter seguire in prima persona sul campo - passo dopo passo, progetto dopo progetto, successo dopo successo - l'intera evoluzione teorica e applicativa di molti di tali metodi. Questa lunga esperienza applicativa ha confermato in modo reiterato e sempre più consapevole la mia felice intuizione iniziale riguardo la potenza di siffatti metodi nei frangenti più critici e complessi non solo della progettazione hardware, ma anche e soprattutto per il software.

Cosa è un linguaggio formale (Formal Language)?

Si tratta di linguaggi simbolici, di natura strettamente logico-matematica, dotati di un vocabolario, di una sintassi e di una semantica non ambigua: il tutto specificato in modo formale e rigoroso. Esistono diverse classi di linguaggi formali sviluppati negli ultimi 30 anni di studio sul software engineering e sui metodi informazionali: linguaggi algebrici come LARCH, OBJ, LOTOS, CASL; linguaggi di modellazione come Z, VDM, CSP, ma anche Reti di Petri ed altri automi a stati finiti; linguaggi più recenti, come OCL e Trio, basati sostanzialmente sui linguaggi di modellazione ibridati con altri simbolismi semiformali (es. UML), e poi logiche temporali lineari, circolari, branching (CTL*, da cui CTL e LTL...). Esistono ulteriori approcci complementari, sostanzialmente basati sulle semantiche formali dei linguaggi di programmazione, e qui giganteggia l'interpretazione astratta: un sapere esoterico encapsulato in applicativi complessi e costosissimi, come Polyspace o l'assai meno pretenziosa suite di applicativi proposti da AbsInt.

I linguaggi menzionati, e affini, permettono in generale la specifica e la verifica di proprietà statiche ma anche dinamiche del software, secondo il tipo di linguaggio/framework: in molte occasioni la soluzione migliore è, infatti, la combinazione integrata di più metodi formali.

Naturalmente detti linguaggi non sono neppure lontanamente parenti dei comuni linguaggi di programmazione. L'uso di un linguaggio formale richiede di operare **una duplice astrazione di tipo matematico, a livello sia procedurale che rappresentativo**: questo è il loro punto di estrema forza ed universalità, ma è anche causa di una serie di difficoltà operative nell'utilizzo, che richiede predisposizione, costanza e pratica assidua.

A cosa serve un FL?

Sostanzialmente un linguaggio formale nasce per assolvere a due scopi:

- 1) Dare una **descrizione matematicamente corretta** e non ambigua delle specifiche (**specificità formale del software** o livello 0); questo serve a comprendere perfettamente la specifica, aggirando i limiti del linguaggio naturale, evidenziando anche eventuali incongruenze della specifica stessa.
- 2) Verificare in modo estremamente rigoroso, con una vera e propria **dimostrazione logica** - esattamente come un qualsiasi teorema - che il software scritto rispetti la specifica (**verifica formale del software**, livello 1).

Non è poco. Nella pratica invalsa sono riconosciuti **tre** livelli principali di uso e applicazione dei FL, con costi ovviamente diversi: si va dalla semplice specifica formale, allo sviluppo e verifica formali, fino alla completa dimostrazione di correttezza ("livello 2" in letteratura), il livello più esaustivo e dai costi più elevati, giustificato unicamente nelle applicazioni per le quali sarebbe maggiore il costo di eventuali errori, tanto da renderne inaccettabile il guasto o malfunzionamento.

Perché nessuno o quasi parla di linguaggi e metodi formali?

Nonostante la loro impressionante potenza logica, che consente in molti campi di far sparire alla radice i concetti stessi di "errore software", "patch", "vulnerabilità" ed altri orrori (su cui peraltro alcuni individui ed aziende hanno costruito intere carriere e lucrosi commerci), i metodi formali ed i relativi linguaggi non vanno molto di moda nel mass market e dintorni. Tuttavia, la prova più evidente del loro assoluto successo è manifestata da una **assenza**: l'assenza di gravi notizie negative che riguardano i sistemi critici che ogni giorno regolano il traffico aereo e navale, i sistemi militari, i satelliti, le centrali energetiche, gli impianti petrolchimici... come al solito, un singolo albero che cade fa un gran fragore, ma una intera foresta che cresce non si nota neppure.

I motivi della negligenza nel mass market sono svariati ed investono non solo le caratteristiche intrinseche dei FL, ma anche fattori umani ed aziendali distribuiti tra tutti gli attori del processo di sviluppo software. Qui si possono forse accennare superficialmente almeno gli aspetti principali:

- Un linguaggio formale è eccellente per descrivere task critici, sistemi di controllo, famiglie coordinate e concorrenti di macchine Mealy o Moore arbitrariamente complesse, sistemi a parallelismo massivo, kernel realtime, file systems a prova d'errore, database in tempo reale, macchine virtuali, compilatori di compilatori ed altre straordinarie vette di complessità concettuale informatica (e non solo, basti pensare alle logiche sequenziali ed ai VLSI). Purtroppo però i linguaggi formali "classici" servono letteralmente a nulla nella descrizione di una interfaccia utente interattiva o di un wordprocessor, di un sito web o altro. Sebbene LTL/CTL da un lato e OCL dall'altro siano nati anche per estendere le classi di applicabilità, questa limitazione ha comunque ancor oggi un suo peso.
- I linguaggi formali sono **automatizzabili**, nel senso che sono manipolabili (a livello simbolico) e verificabili da un calcolatore, anche se non esiste un metodo "automatico" per sviluppare del software partendo dalle specifiche. Per una sfortunata contingenza, tuttavia, al di là di qualche simpatico balocco accademico e di alcuni inguardabili accrocchi low-cost, un software in grado di integrare decentemente l'uso di uno o più FL in un coerente processo di sviluppo industriale ha un costo di licenza in €/\$ che parte da **cifre a quattro o anche cinque zeri**. Non essendo oggetti commercialmente appetibili, chi li ha sviluppati inhouse (con notevoli costi) se li tiene ben stretti.
- Ingegneri e sviluppatori, seppure con una distribuzione altalenante secondo l'epoca ed il luogo degli studi, hanno in genere scarsa familiarità con la **matematica discreta, le logiche temporali ed in generale le logiche formali superiori**. A pochi *practitioners* è realmente chiaro come calare questi linguaggi nella pratica, poiché l'uso dei FL deve partire dall'analisi e guidare, senza compromessi o pasticci, l'intero processo di sviluppo. Il successo commerciale dei RAD ed il ricorso continuo alla prototipazione ingenera **abitudini sbagliate** (trial and error, iperfattazione di prototipi, eccesso di concentrazione sull'implementazione) e sostanzialmente incompatibili col rigore dei FL: abitudini che nella pratica risultano assai difficili da sradicare e fortificate dietro un'enorme muraglia di *hype* e pregiudizi fuorvianti. In questo la formazione superiore, sia per le nuove leve che per i professionisti, dovrebbe avere un ruolo più incisivo, a partire dalla divulgazione di case studies non banali il cui successo è dovuto all'impiego rigoroso e costante dei metodi formali.

- Sempre restando nelle softwarehouses, il management non ha la benché minima idea del risparmio di tempi e costi nascosti (nonché di seccature) che si può ottenere spendendo un po' di più all'inizio, nella formazione e nell'applicazione sistematica dei FL all'intero processo di sviluppo. Vista la forma mentis di coloro che prendono decisioni economiche con risvolti tecnici, appare estremamente difficile riuscire a dimostrare in che modo un elevato costo iniziale (non ricorrente, NRE costs) e del personale potrà, sul medio periodo, comportare un'enorme economia di gestione tagliando nettamente i costi di manutenzione dovuti ad errori, con ampie ricadute positive sul valore del prodotto e del brand.
- Dal lato dei committenti mainstream, infine, la situazione è perfino peggiore. Il cliente medio non riesce a mettere assieme e poi validare una specifica decente, per problemi di linguaggio: s'immagini cosa può succedere quando si mettano in mano ad una qualsiasi PMI migliaia di pagine fitte di simboli incomprensibili non solo ai profani, ma perfino ad una moltitudine di ingegneri e informatici. In breve, fuori dai mercati critici pochissimi clienti hanno la capacità e la volontà di finanziare un processo di specifica formale che non sono neppure in grado di seguire e comprendere.

I linguaggi formali sono invece una norma quotidiana in tutti i settori nei quali l'elevata affidabilità del sistema, definita quantitativamente dalle varie normative, è fondamentale, senza compromessi. Sono anche estremamente importanti in ambito IEEE, ACM, ISO ed in generale nella definizione degli standard. Quasi tutte le **normative** (altro mondo sconosciuto per i più) inerenti i sistemi critici fanno diretto riferimento all'uso obbligatorio di metodi formali, ad esempio:

- **DO-178/B**, la più rigorosa attualmente in vigore, per applicazioni avioniche ed equivalenti;
- **EN 60880** per il nucleare e l'energetica;
- **MIL-STD-498**, ovviamente di natura militare;
- **ESA PSS05** per l'aerospaziale;
- **EN 50128** per il mondo ferroviario;
- **EN 61508** sicurezza funzionale industriale e generale, con le sue numerose derivate (es. **EN 61511** per l'industria di processo, **EN 60601** per l'elettromedicale...).

Conclusione: se si ha una buona predisposizione logico-matematica (ma non occorre certo essere "geni" per capire ed usare un linguaggio formale: ben altro è **idearne** uno veramente valido, efficace, usabile!), ed una naturale propensione per l'ordine, la qualità ed i lavori fatti una volta per tutte in modo impeccabile, conviene sempre e comunque studiare bene i principi del software engineering, apprendere almeno un paio di linguaggi formali ed esercitarsi vita natural durante nel loro utilizzo.

Allo stesso scopo, un buon corso avanzato di logica matematica non dovrebbe esimersi dal trattare ampiamente le logiche temporali (almeno quelle lineari) e possibilmente anche la teoria categoriale. In questo modo ne guadagnerà, per apertura mentale ed abitudine al rigore logico, anche la produzione del software meno critico, cioè lo sbocco lavorativo a cui è orientata la stragrande maggioranza degli studenti.

Un minimo assaggio di bibliografia, privilegiando le edizioni più recenti e i manuali di base:

- **"Simulation engineering"**, Ledin, CMP books, 1-57820-080-6
- **"Computability and Logic"**, Boolos & Burgess, Cambridge University Press, 0-521-00758-5
- **"The Way of Z: Practical Programming with Formal Methods"**, J. Jacky, Cambridge Press, 0521559768
- **"Z: An Introduction to Formal Methods"**, Diller, Wiley, 0471939730
- **"Introduction to Formal Specification And Z"**, Potter & Sinclair, Prentice-Hall, 0132422077
- **"Understanding Z: A Specification Language and its Formal Semantics"**, Spivey, Cambridge University Press, 0521054141
- **"Le Reti di Petri: Teoria e Pratica. 1. Teoria e Analisi"**, G. W. Brams, Editions Masson
- **"Le Reti di Petri: Teoria e Pratica. 2. Modellazione e Applicazioni"**, G. W. Brams, Editions Masson

- "**Teoria dei sistemi ad eventi discreti**", Carlucci & Menga, UTET
- "**Automazione Industriale: Controllo Logico con Reti di Petri**", Luca Ferrarini, Pitagora Editrice
- "**Model checking**", Clarke & Grumberg, MIT Press, 0-262-03270-8
- "**Software blueprints**", Robertson & Agustì, AWL, 0-201-39819-2
- "**Introduction to Formal Languages**", Révész, Dover, 0-486-66697-2
- "**Logic in computer science**", Huth + Ryan, Cambridge, 0-521-65602-8
- "**The temporal logic of reactive and concurrent systems**", Manna & Pnueli, Springer-Verlag, 0-387-97664-7
- "**Computer-aided verification - CAV 90, LNCS 531**", Clarke & Kurshan, Springer-Verlag, 0-387-54477-1
- "**Computer-aided verification - CAV 97, LNCS 1254**", Grumberg, Springer-Verlag, 3-540-63166-6
- "**Computer-aided verification - PSCS**", Kurshan, Princeton press, 0-691-03436-2

Quando si parte il gioco de la zara...

*...colui che perde si riman dolente,
repetendo le volte, e tristo impara.
(Purgatorio, VI, 1-3)*

Nulla di meglio degl'immortali versi del nostro toscanissimo Vate per aprire questa entry dedicata ad un generatore pseudocasuale.

Il giuoco della "zara" (vocabolo strettissimamente imparentato con termini di uso quotidiano come *azzardo*, ma anche *zero*, *cifra* e *zefiro*) non è che un antico gioco di dadi: e proprio i dadi sono uno dei simboli *par excellance* della casualità e dei capricci della Fortuna, la dea bendata.

Ma a cosa serve la casualità, o una qualche sua approssimazione, nel mondo del calcolo? Nella prassi applicativa sono decisamente molte le situazioni nelle quali è necessario generare una sequenza (pseudo)casuale di numeri non correlati tra loro: ad esempio, per **simulare un esperimento fisico** (esiste una vastissima mole di letteratura inerente i metodi cosiddetti "Monte Carlo", definizione che risale direttamente a [John von Neumann](#)), o per popolare un database effettuando **prove di carico**, per eseguire **analisi statistiche**, per il **collaudo di algoritmi** numerici o di **protocolli di comunicazione**, e anche per **esigenze crittografiche** - esempio classico: per generare una valida chiave per OTP (one-time pad) di [Gilbert Vernam](#), algoritmo notevole, unico per il quale esiste una **dimostrazione di inviolabilità**. Inventato poco meno di un secolo fa, non pare molto noto nel mainstream (fors'anche a causa del suo utilizzo poco pratico nella quotidianità): si tratta infatti di un algoritmo che richiede una chiave segreta *genuinamente casuale* con lunghezza *non inferiore a quella dei dati da cifrare*.

Sfortunatamente però ad oggi non disponiamo di una definizione matematica di "casualità" **sufficientemente rigorosa e soddisfacente**, e in pratica coesistono diverse batterie di test empirici statistici per valutare la "bontà" di un generatore casuale, come ad esempio il famoso test "chi-square". Tuttavia, è noto che solamente i **generatori hardware** (normalmente basati su fenomeni di rumore elettrico nei semiconduttori e/o su effetti quantistici ben noti) sono in grado di soddisfare le più stringenti specifiche, mentre gli algoritmi per la generazione di sequenze pseudo-casuali (a rigore, è questa la definizione) soffrono di vari difetti che li rendono adatti ed efficaci solo in alcune tipologie di applicazioni.

Non esiste, in sostanza, un RNG (Random Numbers Generator) puramente software che risulti perfettamente **adatto ad ogni tipo di utilizzo**. In particolare, pare che non si ripeta mai a sufficienza che i generatori "crittograficamente sicuri" non sono in assoluto "migliori" di tutti gli altri. Sono anzi normalmente penalizzati dal punto di vista prestazionale e risultano inadatti nella quasi totalità delle altre applicazioni: in special modo nelle simulazioni più complesse, che spesso richiedono **centinaia o migliaia di sorgenti pseudorandom contemporaneamente attive**.

Il RNG del quale ci occupiamo oggi, denominato [Mersenne Twister](#), rientra nella categoria dei generatori lineari congruenziali, ed è quindi **per definizione** inadatto per applicazioni crittografiche esattamente come tutti gli altri della sua categoria, a causa della intrinseca riproducibilità dello stato interno e, a lungo andare, della intera sequenza da parte di un ipotetico attaccante - questo fatto, ripetuto *usque ad nauseam* dagli stessi autori e da decine di terze parti, è talmente ovvio che non deve stupire alcun lettore algoritmicamente alfabetizzato.

Con ciò, il MT si è negli anni imposto come l'algoritmo preferenziale nella quasi totalità delle applicazioni standard per un RNG, con particolare riguardo a grandi simulazioni e ottimizzazioni combinatorie, si è ritagliato - *rara avis* - un posto d'onore in moltissime applicazioni embedded su microcontroller di taglia media, ed è stato implementato come algoritmo di default per la funzione Random() o equivalente in un lungo elenco di librerie di runtime per linguaggi d'alto livello, inclusi gli ottimi Python e Ruby.

Le caratteristiche fondamentali di questo algoritmo, infatti, sono tali da renderlo ottimale per la stragrande maggioranza delle applicazioni. Desidero qui almeno elencarle succintamente.

1) Il periodo di un RNG è il numero di ripetizioni dopo il quale la sequenza pseudocasuale inizia a ripetersi. Nel caso di MT, tale **periodo** è semplicemente **mostroso**: $2^{19937}-1$. Tale valore dà anche il nome all'algoritmo, in quanto trattasi appunto di un [primo di Mersenne](#).

2) Ha un **ordine di equidistribuzione dimensionale estremamente elevato**, pari a 623 dimensioni. Questo ulteriore risultato record (i più comuni RNG arrivano a malapena a fattori dell'ordine di 5 o 6 dimensioni) sposta molto lontano dalle normali applicazioni i confini del problema di implicita correlazione dell'output, comune a quasi tutti i RNG di tipo LCG. Tale problema si evidenzia quando si considerano valori pseudocasuali generati in stretta successione, usandoli come elementi di un vettore che identifica un punto in uno spazio multidimensionale. Ciò è stato mostrato fin dal 1968 da George Marsaglia nel suo famosissimo articolo "[Random numbers fall mainly in the planes](#)" nel quale viene per la prima volta illustrata e quantificata l'esistenza di iperpiani preferenziali sui quali si aggregano i punti correlati a codesti valori pseudocasuali, invece di occupare uniformemente l'iperspazio dei valori possibili. Tali iperpiani portano da allora il nome di Marsaglia.

3) È uno tra i più **veloci**, se non il più veloce in assoluto, tra i generatori pseudorandom noti: tanto da poter essere impiegato anche in sistemi embedded e/o con ridotta capacità di elaborazione, i.e. mobile, consumer e affini.

4) La sequenza, nelle più recenti implementazioni dell'algoritmo, può dirsi **genuinamente casuale** in tutti i singoli bit di output, senza eccezioni o criticità, nell'accezione di "casualità" richiamata sopra: ovvero il tipo e il numero di test statistici superati, che include tutti i test più selettivi ad oggi noti.

Queste caratteristiche, in buona parte annunciate e comprovate dagli autori fin dalla prima versione dell'algoritmo, sono poi state asseverate in pratica dalle varie implementazioni, facendo di MT e delle sue varianti più immediate un algoritmo decisamente ottimale e versatile per quasi ogni tipo di normale applicazione degli RNG.

Pur senza scendere nei dettagli algebrici, che comunque sono decisamente interessanti, vorrei qui almeno accennare al fatto che l'algoritmo deve la sua elevatissima efficienza alla scelta di lavorare su un campo finito di ordine 2, il che consente di rappresentare efficientemente ciascun polinomio su una semplice stringa di bit di dimensione data e di trasformare somme e moltiplicazioni in operazioni tra le più efficienti per una normale CPU (xor e shift). L'uso di raffinati metodi di reticolo consente di ottimizzare ulteriormente il risultato.

Vale la pena di rimarcare che algoritmi di questo genere sono possibili **esclusivamente** in tale campo finito, e questo rappresenta la vera e propria rivoluzione concettuale che contraddistingue MT rispetto ai più comuni algoritmi storici, basati esplicitamente sul modulo per grandi valori.

Link utili:

- [Mersenne Twister: home page ufficiale](#)
- [Implementazione originale degli autori Takuji Nishimura e Makoto Matsumoto](#)
- [Descrizione dell'algoritmo](#)
- [Implementazione alternativa di Michael Brundage](#)
- [Home page pLab](#): sito accademico dedicato esclusivamente ai RNG

Per curiosità, riportiamo qui integralmente il codice C dell'implementazione *public domain* proposta da [Michael Brundage](#):

```
#define MT_LEN 624

int mt_index;
unsigned long mt_buffer[MT_LEN];

void mt_init() {
    int i;
    for (i = 0; i < MT_LEN; i++)
        mt_buffer[i] = rng.random_integer();
    mt_index = 0;
}

#define MT_IA 397
#define MT_IB (MT_LEN - MT_IA)
#define UPPER_MASK 0x80000000
#define LOWER_MASK 0x7FFFFFFF
#define MATRIX_A 0x9908B0DF
#define TWIST(b, i, j) ((b)[i] & UPPER_MASK) | ((b)[j] & LOWER_MASK)
#define MAGIC(s) (((s)&1)*MATRIX_A)

unsigned long mt_random() {
    unsigned long * b = mt_buffer;
    int idx = mt_index;
    unsigned long s;
    int i;

    if (idx == MT_LEN*sizeof(unsigned long))
    {
        idx = 0;
        i = 0;
        for (; i < MT_IB; i++) {
            s = TWIST(b, i, i+1);
            b[i] = b[i + MT_IA] ^ (s >> 1) ^ MAGIC(s);
        }
        for (; i < MT_LEN-1; i++) {
            s = TWIST(b, i, i+1);
            b[i] = b[i - MT_IB] ^ (s >> 1) ^ MAGIC(s);
        }
        s = TWIST(b, MT_LEN-1, 0);
        b[MT_LEN-1] = b[MT_IA-1] ^ (s >> 1) ^ MAGIC(s);
    }
    mt_index = idx + sizeof(unsigned long);
    return *(unsigned long *)((unsigned char *)b + idx);
/*
Matsumoto and Nishimura additionally confound the bits returned to the caller
but this doesn't increase the randomness, and slows down the generator by
as much as 25%. So I omit these operations here.

r ^= (r >> 11);
r ^= (r << 7) & 0x9D2C5680;
r ^= (r << 15) & 0xEFC60000;
r ^= (r >> 18);
*/
}
}
```


Il genio del Transputer

Transputer è la sincresi di **Transactional Computer**, come era riportato in molti seri articoli dell'epoca. Altre proposte interpretative vertono su "Transistor Computer", per evidenziare l'analogia macroscopica tra un singolo transistor entro un circuito complesso, ed un singolo nodo di elaborazione entro una architettura parallela.

In pieni anni Ottanta, mentre alla Intel stavano ancora tentennando e rimuginando se potesse essere davvero utile sviluppare processori CISC per le masse a 32 bit (mentre i Vax 11 li montavano già di serie fin dagli esordi, inizialmente realizzati a discreti su schede in formato lenzuolo e poi integrati nelle serie MicroVAX), la piccola società anglosassone INMOS iniziò a sviluppare un processore pensato esplicitamente per il parallelismo massivo, quindi in sostanza concepito come nodo entro una rete di numerosi processori interconnessi.

Questi processori, sui quali ho avuto l'onore e il piacere di lavorare per molti anni, sono davvero emblematici per moltissimi aspetti. Nel 1985, in nettissimo anticipo sui tempi, i Transputer furono i primi processori VLSI - ovviamente RISC! - ad incorporare una notevole quantità di memoria RAM e (almeno) quattro porte di I/O *high speed*.

Il loro binomio ideale con l'architettura Microchannel (MCA), un'altra geniale idea visionaria che "il mercato" stupidamente non ha saputo apprezzare, li rende in assoluto uno dei migliori e più flessibili prodotti dell'ingegno umano nell'intera storia del calcolo automatico.

I Transputer sono anche un esempio paradigmatico di processore **studiato in modo esplicito per ottimizzare l'uso di un linguaggio di altissimo livello** ([Occam](#), basato sul noto [CSP](#) ideato dal grande Hoare), dimostrando anche nella pratica che questa è la Via Regia.

Occam 2 (la versione precedente era poco più che uno studio di fattibilità) e i Transputer sono stati sviluppati praticamente in parallelo da INMOS, seguendo la pratica ingegneristica del **codesign** ovvero il design contemporaneo ed interdipendente di hardware e software, che ancor oggi impera nella migliore letteratura e nella prassi applicativa.

Occam è un linguaggio robusto, potente, espressivo, progettato in modo serissimo, basato su una rigorosa algebra di processo come CSP e quindi in grado di garantire formalmente una serie di attributi fondamentali di ogni programma.

Oggi assistiamo regolarmente al fiorire di decine di core entry e midrange specificamente ottimizzati per linguaggio C, ad esempio i più recenti Atmel ATMega e PIC18 con "extended ISA", per non dire dei DSP VLIW: in sostanza la "normalità" – ormai anche nei design baseline – consiste nel seguire la strada segnata anche dai geniali innovatori della INMOS, non a caso ora acquisita da SGS-Thomson.

Inoltre, la direzione del parallelismo massivo è da sempre la scelta privilegiata per il supercomputing: buoni ultimi anche Intel e AMD si sono accodati, con il multicore ed altre recenti "innovazioni", ovviamente in buona parte ideate vent'anni fa da altri.

Ma non basta: il valore didattico e applicativo di questi processori è inestimabile. Singolarmente sono piuttosto semplici, e strutturalmente offrono una flessibilità e una potenza senza pari. Esiste anche una [**fiorente scuola**](#) di programmazione dei Transputer utilizzando dialetti C appositamente modificati per il parallelismo.

Senza mezzi termini, queste macchine sono e restano avanti anni luce rispetto all'era in cui sono nate, in anni in cui il mainstream stava ancora faticosissimamente arrancando tra 8086, 286, 386sx, MC68020 e 030 e giù di lì – *excussez du peu*.

Dal punto di vista architettonale, abbiamo a che vedere con un tripudio di idee una più geniale e innovativa dell'altra. I Transputer, con la loro **rete punto a punto** strettamente orientata alle **transazioni message-passing** (da cui il nome!) facevano già uso di switch e altri sofisticati accorgimenti di arbitraggio hardware e firmware per l'interconnessione: la rete era strutturata in modo **gerarchico**, con **chip dedicati** all'I/O, controller/arbitri ed i veri e propri processori, di maggiore potenza elaborativa, che presiedevano all'elaborazione parallela.

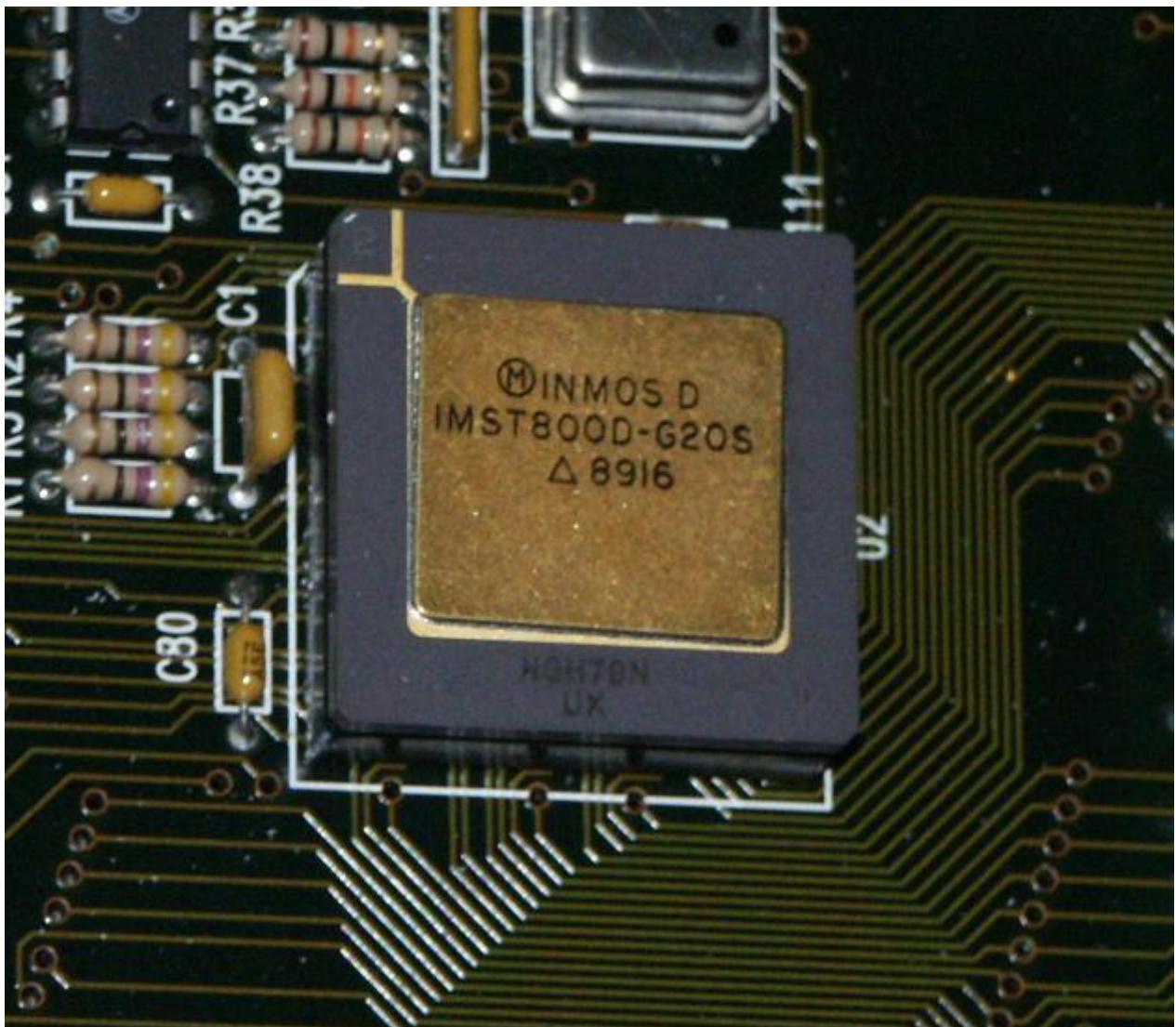
Parlare di questi chip in modo minimamente approfondito richiederebbe almeno un volume d'encyclopedia, anche perché sono varie le generazioni di core che si sono succedute. Il solo linguaggio Occam (la cui licenza,

nei primi anni Novanta, costava poco più di venti milioni di lire: ma l'hardware aveva costi decisamente limitati, in proporzione) ha poi una sua vasta letteratura specifica.

Ad oggi esistono numerosi gruppi di supporto e documentazione storica, il più famoso dei quali è sicuramente il [Transputer Archive](#) presso il sito oxoniano sui metodi formali (Occam è un tipico **linguaggio funzionale**, oltre ad essere intrinsecamente parallelo...).

Esiste inoltre un sito di riferimento e archivio storico della [Inmos](#), ora parte dell'SGS. I siti dedicati al Transputer e ad Occam sono comunque tuttora [numerosi](#), e ovviamente non mancano veri e propri [simulatori](#), alcuni anche molto sofisticati.

Ricordo inoltre che una delle mie macchine preferite di ogni tempo, l'Atari ATW800, montava schede a Transputer per realizzare un vero e proprio "mostro" di potenza elaborativa.



Lascia... o raddoppia?

Sommario

Raddoppio ricorsivo, recursion doubling, o metodo fan-in: con queste denominazioni si indica un approccio peculiare dei sistemi (massivamente) paralleli e dei processori vettoriali. Il presente articolo ne introduce le idee fondamentali tramite un semplice esempio, di cui si delinea anche un basilare profilo prestazionale.

«È più facile quadrare un circolo che arrotondare un Matematico»
(Augustus de Morgan, 1806-1871)

1 Introduzione.

Il materiale qui presentato costituisce una riorganizzazione ed un ampliamento di quanto già discusso sul *blog* dell'autore «Titolo provvisorio...» e ne mantiene volutamente inalterati sia il tono colloquiale e didascalico che l'impostazione strettamente divulgativa.

I **vector processor**, assieme ai Transputer, sono una delle poche vere idee geniali che costellano la storia dell'informatica applicativa, e (naturalmente) non hanno avuto il successo meritato a livello di personal computing. In compenso i soliti raccattatutto del mainstream hanno "ereditato" numerose idee del vector processing, facendone la base di "novità" tecnologiche come MMX e XMM. Eppure non occorrono particolari doti immaginative per comprendere le enormi potenzialità dei VP in accoppiata con i linguaggi **vettoriali** come APL, Q o J. I vantaggi in termini architettonici sono forse meno intuitivi, ma egualmente enormi: ad esempio, l'accesso vettoriale alla memoria (totalmente implementata on-chip in talune architetture) implica la totale inutilità di ineleganti soluzioni come la cache¹! Il ciclo di potenza, determinato dagli stadi che devono essere alimentati durante le singole fasi elaborative è altresì molto vantaggioso e ne fa un processore a bassissimo consumo energetico specifico. Non a caso, da anni i VP vengono applicati in sistemi embedded CPU-intensive, oltre al magico reame del supercomputing che da sempre costituisce la loro più naturale vocazione.

Come insito nell'onomastica, un processore vettoriale è un processore MIMD² o più spesso SIMD³ progettato per essere in grado di operare su interi vettori, non già su singoli valori (scalar). Per costruzione tali processori ammettono nella ISA operazioni aritmetiche solo tra elementi nella medesima posizione: quindi l'i-esimo elemento di un vettore operando dovrà di norma essere elaborato unicamente insieme all'i-esimo elemento di un secondo vettore, e così via, in stretta corrispondenza. Questo vincolo semplifica in maniera formidabile la progettazione del processore a parallelismo massivo, configurandolo esattamente come una serie di "corsie" a fronte di un casello autostradale: ad ogni corsia corrisponde una posizione nel vettore-operando, e in definitiva il numero di lanes esprime la massima dimensione di un vettore trattabile parallelamente in una singola operazione. Resta inteso che vettori di dimensione maggiore rispetto alla dotazione di lanes vengono comunque trattati segmentandoli.

Un buon punto di partenza per maggiori informazioni è l'appendice G dell'Hennessy-Patterson [HP]. Il secondo autore, in particolare, ha contribuito con numerosi articoli scientifici al progresso del settore del vector processing.

¹L'architettura vettoriale è la più tipica soluzione per il progetto e la realizzazione di supercalcolatori, a partire dai Cray. Dunque possiamo attenderci qualche "sorpresa" hardware rispetto all'architettura "povera" dei PC e server mainstream.

Due concetti risultano di cruciale importanza in un vector processor:

1) l'operabilità strettamente vincolata a posizioni corrispondenti nei vettori-operando, che ovviamente implica la garanzia di nessuna interazione o dipendenza tra i dati di un medesimo vettore;

2) il rapporto tra lanes e lunghezza del vettore-operando, che sovente comporta l'esecuzione iterativa di una medesima operazione su più dati memorizzati in locazioni strettamente sequenziali.

Il tipico approccio è quello di architetture come lo Stardent Titan, con i suoi banchi di registri multipli (es. 8 registri da 64 lanes ciascuno) e la memoria interleaved tipica dei supercalcolatori.

In base a tali considerazioni, anche se effettivamente esistono schemi di caching per talune architetture di VP (queste sì davvero esotiche), l'approccio generale - condiviso anche da taluni processori ibridi scalari/vettoriali, o comunque capaci di emulare modalità vettoriali - è quello di usare per costruzione il bypass della cache, grazie a banchi di registri dedicati - una soluzione ampiamente in uso anche all'altro estremo del computing applicativo, ossia nei sistemi embedded.

²Acronimo di Multiple Instruction, Multiple Data.

³Acronimo di Single Instruction, Multiple Data.

2 Un elementare esempio algoritmico.

Come per quasi ogni altra piattaforma, anche nel caso dei vector processors i migliori risultati si ottengono impiegando algoritmi appositamente studiati o modificati per adattarsi alle caratteristiche di questi interessanti core. Una delle tecniche più importanti per la ristrutturazione e il riadattamento ai vector processors di algoritmi classici consiste nel **raddoppio ricorsivo**. Il recursion doubling (o metodo fan-in, dall'elettronica) è una tecnica di tipo *divide et impera* per scomporre algebricamente un problema, rendendolo più trattabile vettorialmente.

Il più basilare degli esempi riportati sui testi è l'arciclassico *prodotto interno di due vettori in \mathbb{R}* , universalmente definito come segue - data la dimensione dei vettori stessi espressa dall'intero positivo arbitrario $n > 1$:

$$p = \sum_{i=0}^n x_i \cdot y_i \quad (1)$$

Supposto che n sia pari, si pone $m = n/2$ ed è immediata la seguente riscrittura, con ovvio significato dei simboli e in particolare degli indici:

$$p = \sum_{i=1}^m x_i \cdot y_i + \sum_{j=m+1}^n x_j \cdot y_j \quad (2)$$

Si tratta, in questo caso, di calcolare separatamente due somme (parziali) di prodotti, unendole poi con una somma finale. Si nota immediatamente che tali somme sono del tutto *indipendenti* l'una dall'altra, e possono quindi essere calcolate in modo disgiunto e nel medesimo momento. Questa condizione di intrinseco parallelismo matematico è la chiave per la comprensione dell'intera strategia. Generalizzando, si tratta di iterare (caso per caso) un tale procedimento, secondo il numero di processori target, chiamiamolo P , e la dimensione dell'input, che chiamiamo n come sempre si usa in complessità computazionale. Si deve notare che codesta scomposizione ha anche l'eccellente effetto collaterale di minimizzare gli errori di accumulo tipici del floating point, a causa della sostanziale indipendenza dei risultati che vengono poi ricombinati linearmente, con l'effetto globale di aumentare notevolmente la stabilità dei relativi algoritmi numerici.

3 Stima prestazionale del metodo.

Non è difficile valutare quantitativamente il guadagno prestazionale di un simile approccio. Assumiamo, per semplificare, di avere già pronti i prodotti membro a membro: tale tempo in realtà non è direttamente influenzato dal cambio di strategia algoritmica, dunque lo consideriamo costante, omettendolo dal calcolo dei rapporti.

Chiamiamo t_S il tempo necessario per una operazione elementare di somma tra due addendi, a valle del calcolo dei prodotti: avendo n addendi, avremo $n - 1$ addizioni e quindi, nella forma manualistica elementare, il calcolo del prodotto interno avrà ovviamente costo in tempo pari a:

$$(n - 1) \cdot t_S \quad (3)$$

Spezzando il problema in due semisomme parallele, il tempo necessario va praticamente a dimezzarsi. Avremo infatti:

1. il tempo necessario ad effettuare circa la metà delle somme (meno una: i segni sono sempre uno in meno degli addendi...), e
2. il tempo necessario a movimentare i risultati parziali, che chiameremo t_C , e sommarli.

Scriviamo nel modo più pedissequo la formula, per meglio memorizzarne la composizione, prima di semplificarla algebricamente.

$$\left(\frac{n}{2} - 1\right) \cdot t_S + t_C + t_S = \frac{n}{2} \cdot t_S + t_C \quad (4)$$

Il rapporto tra questi due costi, espressi dalla 3 e dalla 4 rispettivamente, ci fornisce il valore dell'ottimizzazione:

$$G = \frac{(n - 1) \cdot t_S}{\frac{n}{2} \cdot t_S + t_C} \quad (5)$$

Tale rapporto, in pratica, è molto vicino a 2. Per generalizzare, supponendo di avere un numero di processori P pari alla dimensione n del problema (con n potenza intera del 2):

$$G = \frac{(n-1) \cdot t_S}{k \cdot (t_S + t_C)} \quad \text{per } n = 2^k, k > 1 \quad (6)$$

A noi conviene, per semplificare ulteriormente la formula, esprimere il tempo t_C in funzione del tempo di somma. Su talune architetture, uno dei due risulta per costruzione trascurabile rispetto all'altro. Poniamo dunque $\tau = \frac{t_C}{t_S}$ e dividiamo numeratore e denominatore per la quantità t_S - sicuramente non nulla... Possiamo così scrivere la nostra formula di speedup in funzione della dimensione del problema:

$$G = \frac{n-1}{(\tau+1) \cdot \log_2 n} \quad (7)$$

Allo stesso modo, possiamo esprimerla in funzione del numero di processori:

$$G = \frac{P-1}{(\tau+1) \cdot \log_2 P} \quad (8)$$

Asintoticamente, annullando i tempi di comunicazione e le latenze, è interessante notare che rimane:

$$\bar{G} = \frac{P-1}{\log_2 P} \ll P \quad (9)$$

A questo punto, è solo opportuno rimarcare che il fattore di costo t_C assume una notevole importanza, in quanto riassume globalmente tutti i costi introdotti con l'uso del raddoppio ricorsivo: salvataggi, riletture e altre inefficienze intrinseche. Tale fattore quantifica il concetto che in questo genere di transazioni "si perde comunque qualcosa", di norma secondo la legge macroscopica e asintotica $\frac{N}{\log_2 N}$ e ciò implica, in definitiva, che non serve aumentare a dismisura il numero di processori (o lanes) in parallelo, in quanto esiste un limite superato il quale i costi aggiuntivi (legati principalmente ai trasferimenti di memoria) superano i vantaggi. Ciò è generalmente noto come argomento o legge di Amdahl sui limiti del parallelismo, che oltre un certo valore introduce più inefficienze di quante possa eliminarne.

La bottom line è che, per i vector processors, esiste un limite pratico per i vantaggi ottenibili aumentando il numero di lanes, il che conduce a poter calcolare in modo molto pratico un break-even point che consente di progettare core (e multicore, come in alcune recenti proposte) VP dimensionati in modo ottimale per la generalità delle applicazioni e la dimensione dei problemi normalmente trattati.

4 Conclusioni.

Come già ricordato in apertura, il materiale qui presentato costituisce sostanzialmente una rielaborazione e riorganizzazione di contenuti già pubblicati sul *blog* dell'autore «Titolo provvisorio...». Si è sinteticamente introdotto il concetto di raddoppio ricorsivo tramite il più classico degli esempi, estrapolando poi in modo molto didascalico una formula generale per stimare le prestazioni del metodo in funzione del tempo di clock richiesto da una data operazione atomica, parametro certamente noto per ogni data architettura. L'Autore spera di avere almeno incuriosito il lettore, incoraggiando ulteriori approfondimenti e ricerche.

Riferimenti bibliografici

[HP] Hennessy and Patterson, *Computer architecture a quantitative approach, 6th edition*, ELSEVIER.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

Happy Birthday, Lady Ada!

Il 10 dicembre 1815 nasceva a Londra Augusta Ada Byron, figlia legittima (l'unica) del poeta George Gordon, meglio noto come Lord Byron, e di Anna Isabelle (Annabella) Milbanke, baronessa di Wentworth. Dopo il matrimonio con il barone Lord William King, poi nominato Conte di Lovelace, la nostra Ada diverrà Lady Ada Augusta, Contessa di Lovelace.

In occasione del secondo centenario della nascita di Ada, è doveroso ricordare brevemente ma nel modo più corretto la figura di colei che è a buon diritto considerata la prima programmatrice della storia, vera e propria madrina dell'informatica, principalmente a causa di **questo** esempio prototipale di procedura per il calcolo dei numeri di Bernoulli sulla Macchina Analitica di Babbage.

Tale "macchina" avrebbe dovuto essere un pesantissimo, complicatissimo, cervellotico mostro meccanico composto da molte più parti di una moderna autovettura e azionato a vapore per il calcolo "error-free" di tavole numeriche e la creazione diretta dei relativi cliché di stampa. La Macchina Analitica, tuttavia, non venne mai realizzata. Essa venne illustrata da Babbage a Torino durante il secondo congresso degli scienziati italiani e succintamente descritta in un articolo in francese di Luigi Federico Menabrea, noto scienziato italiano e allievo di una vera gloria italica ottocentesca, l'ingegnere e matematico Giorgio Bidone, studioso di idrostatica e fluidodinamica di fama internazionale.

Ebbene, la nostra Ada è passata alla storia per la sua **traduzione** di tale articolo, al quale ha aggiunto note (incluso il diagramma sopra referenziato) che ne triplicavano la lunghezza, dopo un lungo ed intenso carteggio con lo stesso Babbage, intercalato da numerosi incontri col medesimo.

Fino a qui - dettaglio più, pettegolezzo meno - la parte arcinota della storia, la vulgata che tutti gli informatici e affini hanno sentito raccontare a lezione o letto almeno una mezza dozzina di volte nella vita, specialmente in quest'anno di celebrazioni lovelaciane che ha visto un notevole fiorire di articoli-fotocopia in merito, da pulpiti variamente autorevoli.

Più di qualcuno dei miei lettori sarà però molto sorpreso nell'apprendere che esistono ben **sei** diverse biografie dedicate ai soli sette lustri di non facile esistenza di questa nobildonna: ancora più sorprendente sarà sapere che tali biografie espongono e sostengono punti di vista spesso contrastanti in merito alle sue reali capacità tecniche e matematiche.

In realtà, è fin troppo facile intravedere dietro certe pretestuose polemiche giocate sui classici elementi di disputa storica (carteggi e quaderni andati perduti, dei quali si può solo indirettamente supporre l'esistenza) la spessa ombra dell'interesse fazioso di certa storiografia funzionale a ben precise quanto trite ideologie. In sostanza, nulla più del solito coacervo di tristissime e prevedibili banalità di parte.

La tristezza poi aumenta esponenzialmente nel constatare l'assurda pretesa di taluni autori d'andare a scindere a livello subatomico i presunti singoli contributi di Babbage e di Ada, quasi sempre col fine di circoscrivere e sminuire le capacità di quest'ultima: pessimo vizio che avrebbe fatto fremere di sdegno e indignazione il grande, rigorosissimo storico della scienza **Alexandre Koyré**, ma che purtroppo caratterizza in generale la moderna tendenza a burocratizzare e quantificare il libero pensiero, mortificando lo spirito cooperativo (si pensi alle potenzialità e ai risultati del progetto **Polymath** o del GIMPS, tra i tanti) e la creatività della ricerca in modo assurdo ed inaccettabile, come per anni ha sostenuto con forza anche il compianto professor Giorgio Israel, del quale si sente vieppiù la mancanza.

Fortunatamente, un recente **articolo** dello storico Silvio Hénin (autore tra l'altro anche di **questo bel lavoro**) pubblicato su "Mondo digitale", rivista ufficiale dell'AICA, riassume in modo assai

equilibrato la tormentata vicenda umana della figlia di Lord Byron e il suo intrecciarsi con l'altrettanto complessa vicenda di Charles Babbage.

Rimando senz'altro i lettori all'interessante e ben documentato lavoro di Hénin (con relativa bibliografia) per i dettagli, che purtroppo in questa sede non troverebbero posto per le consuete limitazioni di spazio.

Qui voglio invece segnalare l'opera colossale dei ricercatori dello Science Museum di Londra, che nel 1991 (completando poi la parte di stampa nel 2000) hanno presentato il loro accuratissimo lavoro di costruzione del **Difference Engine N° 2** secondo i progetti originali di Babbage.

In merito alla figura di quest'ultimo, troppo spesso descritto in modo eccessivamente colorito e prono a generare malintesi, consiglio di partire dallo studio dei suoi contributi matematici, ben analizzati tecnicamente in un recente lavoro di J. M. Dubbey "The mathematical work of Charles Babbage", Cambridge University Press, prima di affrontare le varie biografie del personaggio.

In chiusura, una breve nota a margine. Tutti sanno che ad Ada negli anni Settanta è stato dedicato un linguaggio di programmazione ad elevatissima affidabilità, adottato come standard dal DoD, il Dipartimento della Difesa statunitense. Molti meno sono quelli che conoscono il dettaglio dello standard MIL-STD-1815, promulgato il 10 dicembre 1980: si tratta del reference ufficiale del linguaggio Ada, promanato nel giorno del suo compleanno e archiviato con un numero che è in realtà l'anno di nascita della nostra eroina.

Al di là di questa curiosità, non mancano i soliti confusi e male informati(zzati) che - pur ignorando fatti assai più importanti e pregnanti - con irresponsabile leggerezza si sentono in grado di elargire i più sgangherati pareri da operetta in merito al linguaggio Ada e alla sua collocazione nel panorama della produzione di software: un cumulo di raccapriccianti idiozie, in senso greco, che purtroppo siamo talora costretti a leggere in giro per la Rete. Decisamente deprimenti e fastidiose per un professionista che ha iniziato circa trent'anni fa ad utilizzare ed apprezzare Ada per le applicazioni più critiche in assoluto!

Per mettere a tacere definitivamente le troppe voci dell'ignoranza, vale la pena di riassumere qui alcune verità fattuali banalmente verificabili, peraltro arcinote a chiunque non sia un totale profano del mondo dei sistemi embedded critici.

1) Il più recente **standard** del linguaggio è stato promulgato appena tre anni fa, nel 2012. Dunque è più recente perfino del C11!

2) Il linguaggio Ada è tuttora **obbligatorio** per normativa o abitudine contrattuale invalsa in numerosi settori istituzionali nel mondo anglofono (e oltre), a partire dal settore militare, avionico e aerospaziale. Il Dipartimento della Difesa USA lo rese contrattualmente obbligatorio, con pochissime eccezioni, a partire dal 1991. Nel 1997 è stata pubblicata una revisione della modulistica contrattuale DoD che aumenta il numero di eccezioni inizialmente ammesse, ma non sono mai stati cancellati completamente gli obblighi già sanciti, anche perché i tempi di realizzazione e manutenzione in tale ambito sono molto ampi e lo *status quo* delle commesse non può certo essere modificato in corso d'opera. Nel frattempo, sui quattro continenti, una nutrita maggioranza di altre istituzioni militari con grande potere d'acquisto (sia dentro che fuori dalla NATO) avevano prontamente imitato il DoD USA: su scala globale gli obblighi contrattuali per l'uso di Ada sono rimasti in media piuttosto diffusi, allargandosi peraltro rapidamente anche ad altri settori pubblici e privati rigidamente normati. Ada è e rimane in pratica, assieme ad Eiffel, **l'unico linguaggio** considerato accettabile e più facilmente certificabile per architetture di fascia elaborativa media e medio-alta nei tipi di impiego massimamente critici: dal militare all'aerospaziale, dall'avionica al ferroviario all'energetica, ai sistemi di controllo del traffico aereo e navale, nonché in alcuni sistemi bancari e di supercalcolo. Il suo accoglimento e la sua diffusione presso la qualificata comunità internazionale dello sviluppo embedded altamente critico supera ampiamente i dialetti dedicati del linguaggio C, ad esempio, pur

con tutto il relativo supporto di MISRA, SP-Lint, interpretazione astratta e quant'altro. **Nulla**, assolutamente, induce gli analisti di mercato delle multinazionali a supporre che tale radicatissima percezione possa subire variazioni in senso negativo in un futuro prossimo o sul medio periodo.

3) Nell'anno di grazia 2015 Ada è attivamente utilizzato in **decine di nuove commesse istituzionali** in campo militare, navale, ferroviario, avionico, aerospaziale. Tale stato di fatto non è certo destinato a mutare in tempi brevi: in primis perché non ve n'è alcun reale motivo, e visti anche i già richiamati obblighi contrattuali in ordine ai periodi di garanzia di sistemi e impianti (due o tre decenni, secondo i mercati, fino al mezzo secolo di vita utile di una portaerei).

4) Esiste da decenni un fiorentissimo mercato popolato da non meno di tre dozzine tra compilatori, interpreti, runtime blindati, RTOS partizionati ARINC-653 con runtime Ada: mercato che non conosce crisi, sostenuto dai soliti noti colossi multinazionali tuttofare e da alcuni dei più importanti player del mercato embedded critico. Al di là di un paio di noti opensource accademici come GNAT o l'assai più solido ecosistema **SPARK**, quasi tutti i prodotti brand si collocano nella fascia alta del mercato e sono certificati o certification-ready per applicazioni della massima criticità (dalla basilare EN 61508 alla RTCA DO-178, la più rigorosa normativa in vigore). Per un elenco (molto) parziale, **si veda ad esempio qui**.

APL e J: linguaggi vettoriali

1 Introduzione.

Per noi tutti è comune l'esperienza di utilizzare una calcolatrice: personalmente ricordo ancora con nostalgia una delle primissime "portatili" Texas Instruments anni Settanta entrata in casa mia, spessa (e pesante!) come una Garzantina, con una sfilza di display rossi a sette segmenti in miniatura, affamata di energia e sensibilmente **lenta** nel produrre i risultati (per la cronaca, la calcolatrice in questione era una TI SR-10: in assoluto il secondo modello di calcolatrice tascabile messo in commercio).

Poi nella nostra vita da studenti si sono avvicendate le prime calcolatrici scientifiche Sharp o Casio, i PET, gli home computer, i PC con gli spreadsheet come VisiCalc - Lotus 1-2-3 - Quattro - Multiplan, le prime pionieristiche applicazioni statistiche come Systat e SPSS, magari un esame o due di calcolo numerico sostenuto anche con l'ausilio di una delle tante HP basate sulla "cervellotica" RPN¹, l'incontro con "i linguaggi del calcolo", da FORTRAN a C++ arricchito con le più varie librerie numeriche.

Oggi esistono ambienti spaventosamente potenti e articolati per il calcolo numerico e perfino simbolico, in grado non solo di aiutare chi fa i conti, ma addirittura di coadiuvare e consentire lo sviluppo di nuova matematica, vista la capacità simbolica e dimostrativa. Come i miei abituali lettori ben sanno aderisco integralmente alla filosofia, partorita da grandissime menti come Wolfram e Chaitin, secondo la quale il futuro della computazione e della matematica costruttiva sarà sempre più strettamente intersecato, fino a fondersi in nuove discipline, nelle quali è il calcolatore stesso che suggerisce e crea nuova matematica interagendo col matematico computazionale che vi siede davanti, aiutando nella comprensione di problemi e modelli non ancora perfettamente focalizzati ma anche producendo vere e proprie dimostrazioni e inferenze.

Tuttavia, nonostante la presenza di ambienti di calcolo così potenti, lo spazio per i due mondi paralleli delle calcolatrici (programmabili, grafiche, RPN, con USB... ma anche meno pretenziose) e dei linguaggi standalone dedicati interamente al calcolo rimane ancora notevole. Non pensiamo solo alle applicazioni legacy in FORTRAN, un vero highlander come il COBOL, oppure alla diffusione di Python con il suo sterminato ecosistema di librerie e framework: esistono anche linguaggi più esotici che sono usatissimi in taluni ambienti.

In questa occasione voglio appunto parlarvi di una coppia di linguaggi vettoriali ai quali sono particolarmente affezionato, da moltissimi anni: APL e J, concepiti dal geniale e compianto Kenneth Iverson. Si tratta di due tra i migliori linguaggi mai ideati, esplicitamente inventati per il calcolo, con una notazione sintetica, potentissima e flessibile.

2 APL e la genesi di J.

APL nasce negli anni Sessanta ed il suo principale problema, paradossalmente, consiste nell'essere eccessivamente in anticipo sui tempi: i suoi operatori sono infatti simboli non convenzionali tratti direttamente dal formalismo matematico, quindi normalmente non presenti sulle tastiere e sui device di output. Si veda anche questa bella pagina storica, predisposta con l'usuale cura dal buon John Savard, che mostra anche alcune interessanti macchine IBM dotate di tastiera estesa appositamente progettata per l'uso di APL. Oggi tale linguaggio potrebbe facilmente rifiorire, sotto forma di applicazione grafica per Windows e X-window, in qualche modo somigliante ad una calcolatrice programmabile con duecentocinquanta "tasti".

Personalmente, dopo le prime esperienze di programmazione APL con degli IBM 5100 (che a metà anni Ottanta erano considerati dei residuati, ma risultavano ancora discretamente diffusi in ambito SOHO) e gli storici terminali IBM 3270 con apposita tastiera, ho avuto la fortuna di utilizzare a fine anni Ottanta non solo

¹RPN è l'acronimo di Reverse Polish Notation, una notazione postfissa per le formule che ha mantenuto una certa utilità in ambito informatico, in quanto semplifica la stesura di parser, oltre ad essere molto amata dagli entusiasti delle calcolatrici tascabili programmabili di HP. Tuttavia, il logico polacco Jan Lukasiewicz (1878 – 1956) al quale viene ascritta la paternità di tale notazione ha invece in effetti ideato la cosiddetta notazione polacca (PN), strettamente simile alla precedente ma che è *prefissa*, per eliminare la necessità delle parentesi e semplificare così la notazione nel calcolo proposizionale: un'idea ad oggi largamente desueta nell'ambito della logica formale.

Ritroviamo invece la RPN nella implementazione dell'ALU (Arithmetical-Logical Unit) di molte CPU, grazie alla sua inerente semplicità implementativa a livello di circuiti logici sequenziali e di RTL (Register-Transfer Logic). Il suggerimento di adottare tale sequenza postfissa nella dinamica funzionale delle CPU si deve al noto e poliedrico matematico John Von Neumann (1903 – 1957).

una tavoletta grafica con maschera APL su cartoncino per una workstation HP Apollo, ma anche una tastiera elettromeccanica di nuova concezione dedicata ad APL su una macchina Bull di elevata potenza elaborativa.

Tuttavia, proprio per superare questi inconvenienti come la necessità di un hardware non standard piuttosto costoso (non era ancora il momento dei game cockpit prodotti nel Far East per pochi spiccioli) e le notevoli difficoltà di visualizzazione, all'inizio degli anni Novanta il gruppo di Iverson e Hui ha concepito J, una evoluzione di APL nella quale gli operatori sono costituiti da sequenze di due o più caratteri convenzionali, ovvero una sorta di digrafi e trigrafi. Nella prima metà dei Novanta APL è stato quindi gradualmente abbandonato fuori dall'ambito mainframe, e non sono mancati costosi esperimenti per l'uso di J su vector processor, un connubio in realtà molto naturale, ma confinato ad una nicchia applicativa.

Quanto sia leggibile il linguaggio J, a prima vista, potete giudicarlo da soli:

```
** Esempio 1 **
comb=: [:; (., &.><@:>:@; \.) ^ : ( i .@>:@-~`[ `(`(1:<@i .@{.~<:@-, 2:))

** Esempio 2 **
Choleski=: 3 : 0
n=.#A=.y
if. 1>:n do.
    assert. (A=|A)>0=A NB. check for positive definite
    %:A
else.
    p=.>.n%2 [ q=.<.n%2
    X=(p,p){.A [ Y=(p,-q){.A [ Z=(-q,q){.A
    L0=. Choleski X
    L1=. Choleski Z-(T=(h Y) mp %:X) mp Y
    L0,(T mp L0),.L1
end.
)
** Esempio 3 **
bob =: ":, ' bottle' , (1 = ]) }. 's of beer'_
bobw=: bob , ' on the wall'_
beer=: bobw , ', ', bob , ';' take one down and pass it around , , bobw@<:
```

La reazione del programmatore quadratico medio è del tipo "Ma quella roba è peggio del PERL e del LISP messi insieme...", quindi non allarmatevi. I tre esempi sono liberamente tratti dal sito di riferimento, ricchissimo di materiale di studio. Nell'ordine, il primo genera tutte le combinazioni di ordine m da un insieme di cardinalità $n > m$; il secondo applica la ben nota decomposizione di Cholesky ad una matrice (specialità della casa per APL e J); il terzo è quella simpatica goliardata di "99 bottles of beer".

Dal nostro punto di vista, J è un linguaggio per *operazioni matriciali* di tipo MIMD (Multiple Instruction, Multiple Data). Non si tratta per natura di un *linguaggio funzionale*, al contrario di Haskell o Miranda: tuttavia supporta alcuni costrutti di programmazione basati sulle funzioni di ordine superiore di Backus (il padre della specifica sintattica BNF, tra l'altro), ovvero funzioni in grado di accettare altre funzioni come parametri, e/o di restituirle come risultati. Il linguaggio J viene per questo definito linguaggio di programmazione "a livello funzionale", praticamente unico nella sua categoria, assieme all'ancor più esotico FP ideato proprio da John Backus ed a linguaggi specializzati come ERLANG.

Una curiosità importante: si tratta di uno dei pochissimi linguaggi che non aderiscono al modello di macchina Von Neumann, e peraltro vi sono almeno un paio di interessantissime (sebbene difficilmente per l'informatico quadratico medio) pubblicazioni su possibili modelli di hardware che implementino APL nel modo più efficiente. Il tutto si collega, in maniera forse poco palese ma intuitiva, a modelli semiconosciuti di vector computing fioriti a partire dagli anni Novanta.

Tornando agli aspetti più generali, la sintassi e il modello di parsing di J già da soli sono meritevoli del massimo interesse, e fonte di ispirazione per chiunque abbia motivo di scrivere anche un ristretto subset di funzioni per un parser-giocattolo ad uso "scripting numerico".

L'idea fondamentale dietro la creazione di APL e poi di J è la strettissima somiglianza con la normale notazione matematica, che intende semplificare l'approccio a chi possiede buone basi applicative: lo statistico, il matematico attuariale, il tecnico assicurativo o bancario, e in generale tutte le categorie di applicativi che fanno calcolo numerico e statistico su base matriciale, dall'ingegnere al biologo al fisico.

Dal punto di vista della licenza, come ben spiegato sul sito, il linguaggio è oggi di libero utilizzo per tutti gli utenti e tutte le destinazioni d'uso. Tradizionalmente i sorgenti dell'implementazione standard non vengono forniti (J Software è una normale softwarehouse che vende i propri prodotti), ma uno degli aspetti fondamentali

e molto ben curati è la facilità di integrazione di J con prodotti di terze parti, dalle librerie accademiche per il calcolo numerico al framework dotnet.

Similmente al simpatico (UN)ICON, anche J propone un completo ambiente per Windows con grafica avanzata, supporto UNICODE, interfacce verso database - framework - applicazioni - DTP, il tutto con la qualità e l'affidabilità di un prodotto commerciale utilizzato attraverso i decenni da grandi istituzioni del settore finanziario e scientifico.

Non perdete l'occasione di dare un'occhiata alla mole enorme di documentazione ed esempi, e di installare l'ambiente J sul vostro SO preferito. Il tempo speso ad apprendere J darà sicuramente i suoi frutti, migliorando nettamente la qualità del vostro modo di pensare non convenzionalmente alle soluzioni software, ma anche il modo di fare quei conti applicativi che spesso servono, con un ambiente del tutto gratuito e di grande funzionalità.

Consiglio di stampare e studiare tutta la documentazione, senza dimenticare "J for C programmers" e di seguire passo dopo passo "Concrete Maths Companion" assieme al testo originale: nel giro di qualche settimana avrete appreso bene non solo una importantissima base di matematica discreta, ma anche i fondamenti del linguaggio J. Agli studenti in particolare suggerisco di alternare lo studio dei soliti Matlab e affini con J, usandolo magari per preparare qualche tesina o lavoro di supporto nell'ambito dei vari corsi di analisi numerica, algebra lineare, geometria computazionale.

3 Un esempio applicativo.

Nulla è più efficace di un esempio concreto di uso del linguaggio spiegato passo dopo passo. In questa occasione, useremo un classico problema-balocco e vedremo come popolare una generica matrice $m \times n$ (poniamo, giusto per fissare le idee, $3 \leq m \leq n$ con $m, n \in \mathbb{N}$) secondo lo schema indicativo «a zig-zag» qui riportato:

$$\begin{array}{cccc} 3 & 8 & 9 & 11 \\ 2 & 4 & 7 & 10 \\ 0 & 1 & 5 & 6 \end{array} \quad (1)$$

Convenendo di far partire gli indici da zero e di anteporre l'indice di riga, siano $a_{i,j}$ gli elementi della matrice. In riferimento allo schema 3×4 appena presentato, appare immediatamente chiaro che un ordinamento come quello richiesto corrisponde ad un "percorso" attraverso l'array che tocca, nell'ordine, le seguenti celle:

$$a_{2,0}, a_{2,1}, a_{1,0}, a_{0,0}, a_{1,1}, a_{2,2}, a_{2,3}, a_{1,2}, a_{0,1}, a_{0,2}, a_{1,3}, a_{0,3} \quad (2)$$

Sarebbe parimenti facile riportare, per ogni "mossa", la relativa regola applicata sotto forma di freccia direzionale e/o trasformazione da applicare alla coppia di indici correnti per ottenere la successiva, ottenendo così una rappresentazione operazionale adatta ad una soluzione per FSM o per trasformazioni successive. Tuttavia, tra le numerose soluzioni disponibili (reticolari e modulari, algebriche, a stati finiti...) ritengo in questa sede di privilegiare quella algebrica, basata sulla seguente osservazione. Per comodità, riportiamo intanto in ogni cella i relativi indici di riga e colonna, secondo le nostre comode convenzioni *informatiche*:

$$\begin{array}{cccc} 0,0 & 0,1 & 0,2 & 0,3 \\ 1,0 & 1,1 & 1,2 & 1,3 \\ 2,0 & 2,1 & 2,2 & 2,3 \end{array} \quad (3)$$

Provando a sommare gli indici contenuti di ciascuna cella, si nota qualcosa di decisamente banale, ma forse non ovvio, che aiuta a costruire un approccio destinato a risolvere con facilità tutti gli esercizi di questa categoria:

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{array} \quad (4)$$

Dunque *la somma degli indici è costante per ciascuna antidiagonale*, ed identifica in modo immediato l'antidiagonale stessa. Una volta notato che ciascuna antidiagonale è univocamente identificata dalla somma degli indici delle corrispondenti coordinate, il relativo programmino in C o in Python è in pratica già scritto. Occorre solo mettere a fuoco un paio di dettagli oggettivamente meno interessanti, come il punto di partenza di ciascuna antidiagonale e la relativa lunghezza. Si osservi che si ha la seguente ovvia distribuzione delle origini delle antidiagonali:

$$\begin{array}{cccc} 0 & \nearrow & \nearrow & \nearrow \\ 1 & \nearrow & \nearrow & \nearrow \\ 2 & 3 & 4 & 5 \end{array}$$

3.1 Esempio di implementazione in C89.

Proviamo ora ad applicare le proprietà appena evidenziate in un linguaggio universalmente noto e semplice da seguire. Nel seguente frammento di codice C, siano R e C rispettivamente i limiti superiori per gli indici di riga e colonna della matrice. Abbiamo esplicitato in modo assolutamente ridondante tutte le variabili necessarie alla generazione esaustiva degli indici per una copertura della matrice per antidiagonali, indicando anche con chiarezza la dimensione di ciascuna di quest'ultime - tra parentesi quadre nella printf() di controllo. Il codice fornisce in output una lista ordinata, human-readable, delle coordinate di ciascuna cella appartenente ad ogni antidiagonale, dalla 0 alla $R + C$ inclusive, ossia - in ultima analisi - l'esatta sequenza in cui vengono "visitate" le singole celle della matrice.

```
int ad, col;
int diags = R + C + 1;
for (ad = 0; ad < diags; ++ad)
{
    int first = (ad <= R) ? 0 : ad - R;
    int last = 1 + min(ad, C);
    printf("%2d [%d]: ", ad, last - first);
    for (col = first; col < last; ++col)
    {
        printf("(%d,%d)", ad - col, col);
    }
    puts("");
}
```

Esempio di output per $R = 4$, $C = 8$:

```
0 [1]: (0,0)
1 [2]: (1,0)(0,1)
2 [3]: (2,0)(1,1)(0,2)
3 [4]: (3,0)(2,1)(1,2)(0,3)
4 [5]: (4,0)(3,1)(2,2)(1,3)(0,4)
5 [5]: (4,1)(3,2)(2,3)(1,4)(0,5)
6 [5]: (4,2)(3,3)(2,4)(1,5)(0,6)
7 [5]: (4,3)(3,4)(2,5)(1,6)(0,7)
8 [5]: (4,4)(3,5)(2,6)(1,7)(0,8)
9 [4]: (4,5)(3,6)(2,7)(1,8)
10 [3]: (4,6)(3,7)(2,8)
11 [2]: (4,7)(3,8)
12 [1]: (4,8)
```

Non dovrebbe essere necessario rimarcare che tutte le varianti del problemino sono ottenibili con banali rotazioni e trasposizioni della matrice iniziale, effettuate operando unicamente sugli indici. A questo punto, abbiamo praticamente già scritto una possibile soluzione in linguaggio C, sulla quale non mi soffermerei più del necessario.

```
/*
 **** Soluzione in C al quesito "creazione di una matrice a zig-zag".
 ** Sfrutta la relazione tra gli indici di riga e colonna delle
 ** antidiagonali di una matrice bidimensionale arbitraria.
 */

#include <stdio.h>
#include <stdlib.h>

#define MAX_ROW 5
#define MAX_COL 9
```

```

int Array[MAX_ROW][MAX_COL];

void Fill_Array(const int R, const int C)
{
    /*
    ** Variabili di induzione principali:
    ** AntiDiagonale corrente e COLonna.
    */
    int ad, col;
    /* Totale delle (anti)diagonali nella matrice.      */
    int diags = R + C + 1;
    /* Indicatore della direzione corrente: +1 o -1.  */
    int dir = 1;
    /* Contatore 0..R*C-1 */
    int cnt = 0;
    /* Progressivo per il contatore */
    int progr = 0;

    for (ad = 0; ad < diags; ++ad)
    {
        /*
        ** Ai fini della massima chiarezza, si rendono esplicite le
        ** variabili intermedie dipendenti dai valori di induzione.
        ** Rendere il codice meno leggibile e' sempre possibile.
        */
        int first = (ad < R) ? 0 : ad -R;
        int last = 1 + min(ad, C);
        /*
        ** Tra le varie possibili soluzioni per invertire ciclicamente
        ** il verso di percorrenza, si sceglie la piu' elegante:
        ** la generazione degli indici avviene unidirezionalmente,
        ** quindi il valore di cella viene opportunamente
        ** incrementato o decrementato con una variabile ausiliaria.
        */
        cnt = progr + ((dir > 0) ? 0 : (last - first -1));
        /*
        ** Si aggiorna ora il progressivo, che sara' il valore iniziale di cnt
        ** per il prossimo ciclo.
        */
        progr += last - first;
        for (col = first; col < last; ++col)
        {
            /*
            ** L'assegnazione tiene conto della specificita' dell'esercizio,
            ** che richiede uno zigzag con partenza dall'angolo R,0.
            ** L'indicizzazione sfrutta la relazione tra gli indici per le
            ** celle appartenenti all'antidiagonale (somma i+j costante).
            */
            Array[R-ad+col][col] = cnt;
            cnt += dir;
        }
        dir = -dir;
    }
}

int main(void)
{
    int i, j;
    int Rows, Cols;
    Rows = MAX_ROW;
    Cols = MAX_COL;
}

```

```

printf("Creazione di una matrice a zig-zag: %d x %d\n\n", Rows, Cols);
Fill_Array(Rows -1, Cols -1);
for (i = 0; i < Rows; ++i)
{
    for (j = 0; j < Cols; ++j)
    {
        printf("%2d ", Array[i][j]);
    }
    puts(" ");
}

Cols = Rows;
printf("\nCreazione di una matrice a zig-zag: %d x %d\n\n", Rows, Cols);
Fill_Array(Rows -1, Cols -1);
for (i = 0; i < Rows; ++i)
{
    for (j = 0; j < Cols; ++j)
    {
        printf("%2d ", Array[i][j]);
    }
    puts(" ");
}

return (0);
}
/* EOF: zigzag.c */

```

Ecco uno snapshot dell'output, dopo avere compilato con BCC 5.5.1:

Creazione di una matrice a zig-zag: 5 x 9

```

10 19 20 29 30 38 39 43 44
 9 11 18 21 28 31 37 40 42
 3  8 12 17 22 27 32 36 41
 2  4  7 13 16 23 26 33 35
 0  1  5  6 14 15 24 25 34

```

Creazione di una matrice a zig-zag: 5 x 5

```

10 18 19 23 24
 9 11 17 20 22
 3  8 12 16 21
 2  4  7 13 15
 0  1  5  6 14

```

3.2 Esempio in linguaggio J.

Passiamo alla parte maggiormente interessante. Naturalmente in J non abbiamo alcuna necessità di sporcarci le mani con tutti questi dettagli di bassa cucina: i banali vincoli algebrici e combinatori sopra elencati vengono gestiti in modo trasparente ed efficiente dai potentissimi operatori e dalle ricche primitive del linguaggio.

L'espressione finale necessaria ad ottenere il risultato desiderato è la seguente, e consta di "ben" trentadue caratteri, inclusi gli spazi (opzionali) e le parentesi (idem):

```

($ [: /:@; [: <@|. `</. [: |. i.) 3 4
3 8 9 11
2 4 7 10
0 1 5 6

```

Come di consueto in letteratura, riporto qui le istruzioni J e il relativo output, riproducibile nell'ambiente interattivo J 6.02 (la versione 7 è leggermente più ostica per i neofiti).

La superiorità rispetto agli oltre duemilaseicento caratteri - "fuori tutto", come si dice in Marina - del programma C appena presentato è semplicemente schiacciante. Appare pressoché impossibile ottenere una concisione maggiore, eccetto forse con una macro entro qualche esotico ambiente di calcolo matriciale. Qui diviene tangibile l'enorme potenza di un linguaggio come J, nato appositamente per l'algebra lineare e la matematica discreta. Il rovescio della medaglia appare altrettanto evidente. Mettete "quella roba" davanti al naso di un practitioner quadratico medio: per quanto "bravo" ed "esperto" nel 999,9 per mille dei casi vi dirà che "...non ha alcun senso, hai lasciato passeggiare il micio sulla tastiera...".

E invece J è anche relativamente facile da leggere (molto più del PERL, tanto per dire), con un minimo di allenamento. Qui, usando come scusa il puerile problemino dello zig-zag, cercherò di trasmettervi il principio che lo scoglio principale, più che i digrafi e trigrafi, sono i concetti del linguaggio, decisamente controintuitivi per la maggioranza degli informatici.

Per questo vorrei spiegare più da vicino, spazio permettendo, "come funziona" l'espressione prodotta. Dal punto di vista logico, l'espressione compie le seguenti operazioni:

1. Generazione di una matrice opportunamente popolata dai primi $m \times n$ interi;
2. Lettura per antidiagonali, in direzioni alternate;
3. Ordinamento dei valori per righe e colonne;
4. Presentazione del risultato.

Vediamole ora con maggiore dettaglio.

3.2.1 Generazione della matrice.

Occorre subito rimarcare che le espressioni in J vengono valutate **da destra a sinistra**, e come tali è opportuno leggerle, a meno ovviamente delle parentesi. Pertanto dovremo iniziare la nostra analisi da | . i., che peraltro è la parte più semplice: si tratta di un *generatore* di numeri interi non negativi, in grado con identica semplicità di generare sequenze e matrici multidimensionali.

```
i . 3
0 1 2

i . 5
0 1 2 3 4

i . 3 4
0 1 2 3
4 5 6 7
8 9 10 11
```

Semplice, vero? Allo stesso modo, con l'operatore | . possiamo invertire l'ordine di generazione.

```
| . i . 3
2 1 0

| . i . 3 4
8 9 10 11
4 5 6 7
0 1 2 3
```

3.2.2 Lettura per antidiagonali.

A noi occorre ora "passeggiare" sulle antidiagonali della matrice così popolata. Il problema è ovviamente molto comune nel calcolo matriciale, che è il vero e proprio terreno di coltura di J: dunque non deve stupire il fatto che esista un apposito operatore, /.

```

< / . | . i . 3 4
+-----+
| 8 | 9 4 | 10 5 0 | 11 6 1 | 7 2 | 3 |
+-----+

```

Il costrutto `/.` applica la funzione che lo precede (in questo caso il *boxing* `<`) alle antidiagonali di una matrice, per ottenere il bel risultato visibile sopra: l'input viene suddiviso, appunto, per antidiagonale.

Prima di procedere oltre, è però il caso di spiegare un altro paio di concetti fondamentali in J. In tale linguaggio *non esiste priorità degli operatori*, neppure di un limitato sottoinsieme (es. operazioni algebriche e logiche), per il semplice motivo che J riconosce circa **duecentocinquanta operatori** e un gran numero di loro combinazioni: una complessità che supera di gran lunga la capacità ragionevolmente gestibile da un interprete o compilatore - per non dire delle ancor più limitate capacità del bipede che sta seduto davanti alla tastiera.

Ciò implica che, ad esempio, i risultati di una sequenza di operazioni algebriche potrebbero sorprendervi, se non siete più che accorti nell'uso delle parentesi. Quanto appena asserito ha rilevanza particolare in un caso che ci riguarda da vicino: la composizione di funzioni tramite il cosiddetto "gerundio". In effetti, in J la sintassi è specificata con una nomenclatura strettamente modellata attorno a quella del linguaggio naturale, e in particolare all'analisi grammaticale e quindi alle parti del discorso. In realtà il gerundio, al di là dei casi degeneri (è applicabile anche ad una singola funzione, usando una sintassi particolare), può e deve interpretarsi come in linguaggio C si considera *una lista di puntatori a funzione*. In effetti J ha la (sorprendente?) capacità di iterare sui medesimi dati due o più funzioni, alternandole ed espandendole adeguatamente: tutto ciò grazie al costrutto detto, appunto, gerundio. Vediamo un semplice esempio - e, di nuovo: attenzione alla mancanza di priorità degli operatori!

```

1) 2+i . 4
2 3 4 5
2) +/2+i . 4
14
3) */2+i . 4
120
4) +`*/2+i . 4
29
5) 2+3*4+5
29
6) ((5+4)*3)+2
29
7) *`+/2+i . 4
46
8) 2*3+4*5
46
9) ((5*4)+3)*2
46

```

Gli esempi dovrebbero essere autoesplicativi, una volta ricordato che la composizione `n/` interpone l'operatore `n` (nel nostro caso, `+` o `*`) tra gli elementi di una sequenza, e che la valutazione delle espressioni avviene, come già ricordato, **da destra a sinistra**. Ho aggiunto, per comodità referenziale, dei numeri di riga. In particolare, alla 9) e alla 6) ricostruiamo l'esatta sequenza di valutazione dell'espressione gerundiale espansa, che come vedete chiaramente alterna i due verbi (nel nostro caso, `+` e `*` ovvero somma e moltiplicazione multiple) applicandoli opportunamente all'input fornito.

Naturalmente i lettori più smaliziati avranno già capito che questo excursus sul gerundio serve appunto ad introdurre l'operazione di *scansione alternata delle antidiagonali*, che nella nostra espressione è realizzata appunto tramite un bel gerundio:

```

< / . | . i . 3 4
+-----+
| 8 | 9 4 | 10 5 0 | 11 6 1 | 7 2 | 3 |
+-----+
<@|. ` < / . | . i . 3 4
+-----+
| 8 | 9 4 | 0 5 10 | 11 6 1 | 2 7 | 3 |
+-----+

```

```

+-----+
<`(<@|.) / . |. i. 3 4
+-----+
| 8| 4 9|10 5 0|1 6 11|7 2|3|
+-----+

```

Si nota benissimo come, nel secondo caso, l'ordinamento dei valori nei box risulta alternatamente invertito, come desiderato. Il terzo caso, riportato per riferimento, illustra come modificare il medesimo gerundio (massima attenzione alla sintassi!) per invertire la direzione dello zig-zag.

3.2.3 Ordinamento dei valori.

Siamo già molto prossimi al risultato desiderato, com'è evidente analizzando l'output dell'esempio precedente. A questo punto, dobbiamo solo riordinare per righe e colonne gli elementi attualmente raggruppati per anti-diagonali. Il concetto di sorting in J, com'è lecito attendersi, offre qualche sorpresa rispetto ad altri linguaggi. Come spesso accennato in passato, l'ordinamento di un array può essere validamente rappresentato come una applicazione biettiva che mappa gli indici dell'array di input su nuove posizioni. Ad esempio, dato l'array C Array[] = "ASORTINGEXAMPLE", ordinarlo alfabeticamente in senso ascendente significa, di fatto, effettuare le seguenti assegnazioni:

```

ASorted [0] = Array [0]; /* 'A' */
ASorted [1] = Array [10]; /* 'A' */
ASorted [2] = Array [8]; /* 'E' */
ASorted [3] = Array [14]; /* 'E' */
ASorted [4] = Array [7]; /* 'G' */
...

```

In sostanza, per descrivere l'array ordinato è sufficiente fornire la *sequenza degli indici* che compaiono nelle espressioni a destra del segno di uguaglianza, la cui corrispondenza con i primi n naturali è implicita (posizionale). Tale sequenza, in ultima analisi, è una *permutazione degli indici* e rappresenta in forma vettoriale (tabulare) l'applicazione che mappa l'array di partenza su un nuovo array, ordinato secondo un criterio specifico (in questo caso, lessicografico ascendente). Spero sia sufficientemente chiaro che la permutazione in oggetto rappresenta, di fatto, il criterio di ordinamento usato: se volessi un ordine antilessicografico, ad esempio, non avrei che da invertire la permutazione.

Il linguaggio J fornisce, con un uso monadico o diadico (ciò che in altri contesti chiamiamo *unario* e *binario*: in soldoni, l'arietà o molteplicità degli operandi ai quali si applica un operatore) del medesimo gruppo di operatori, sia il metodo per *classificare* un input (i.e. generare una permutazione degli indici), sia la funzione di ordinamento vera e propria, che prende in input una lista e un criterio di ordinamento - al limite, la lista di input medesima, come nel nostro esempio: il che produce il criterio di default, in questo caso lessicografico ascendente.

```

A =: 'ASORTINGEXAMPLE'
/: A
0 10 8 14 7 5 13 11 6 2 12 3 1 4 9
/:~A
AAEEGILMNOPRSTX
(/: A){A
AAEEGILMNOPRSTX

```

Tornando al nostro problema, occorre e basta unire l'azione dell'operatore "grade up" $/:$ con il banale operatore di listing $;$; sull'output finora prodotto per ottenere il risultato desiderato:

```

<@|. `< / .|. i. 3 4
+-----+
| 8| 9 4|0 5 10|11 6 1|2 7|3|
+-----+
; <@|. `< / .|. i. 3 4

```

```

8 9 4 0 5 10 11 6 1 2 7 3
  /:@; <@|. ` < / . | . i . 3 4
3 8 9 11 2 4 7 10 0 1 5 6

  3 4 $ /:@; <@|. ` < / . | . i . 3 4
3 8 9 11
2 4 7 10
0 1 5 6

```

L'ultimo statement illustra come il nostro lavoro, a questo punto, sia praticamente terminato.

3.2.4 Presentazione del risultato.

Giunti a questo punto, è sufficiente ridare alla lista (ottenuta tramite l'operatore ;, a sua volta necessario per elaborare la lista multipla prodotta dall'output precedente) la forma originale di matrice $m \times n$. Data la peculiarità dell'operatore \$ a ciò preposto, usiamo semplicemente le regole di composizione funzionale proprie di J, e l'apposito operatore cap [:.

```

zigzag =: ($ [: /:@; [: <@|. ` < / . [: | . i .)

zigzag 3 4
3 8 9 11
2 4 7 10
0 1 5 6

```

4 Conclusioni.

Come già ricordato in apertura, il materiale qui presentato costituisce sostanzialmente una rielaborazione e riorganizzazione di contenuti già pubblicati sul *blog* dell'autore «Titolo Provvisorio». L'Autore spera di avere almeno incuriosito il lettore, incoraggiando ulteriori approfondimenti e ricerche.

Nota storica: quando il presente articolo è stato pubblicato sul blog dell'Autore, la pagina relativa all'implementazione in J dello zigzag su Rosetta Code non esisteva (in realtà non esisteva neppure il sito stesso). La questione era invece stata discussa molti anni prima nella mailing list del linguaggio J.



Quest'opera viene rilasciata con licenza **Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo** 4.0 Italia. Per leggere una copia della licenza visita il sito web Creative Commons Italia BY-NC-SA 4.0 o spedisci una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Stante la tipica volatilità di Internet, l'Autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante l'originale stesura del presente articolo.

