Retroprogramming: Commodore 64

di M.A.W. 1968

Indice

Capitolo 1

Bibliografia essenziale per Commodore 64.

1.1 Introduzione.

Con oltre 17 milioni di unità prodotte, il Commodore 64 è indiscutibilmente l'Home Computer più venduto della storia e il relativo mercato bibliografico è conseguentemente stato uno dei più sviluppati in assoluto, con centinaia di titoli stampati da decine di editori di ogni dimensione, firmati da autori spesso di fama internazionale nel campo dei microcomputer e relativi processori 8 bit. Qui si vuole proporre un percorso di studio che comprende i testi più essenziali, rivelatisi alla prova del tempo e nella didattica superiori agli altri per qualità dei contenuti, impostazione, accuratezza, autorevolezza.

Per la massima efficacia e qualità dei risultati, il percorso di studio individuale deve necessariamente seguire la consueta progressione: prima l'architettura, poi il linguaggio macchina in sé, poi i vari Assembler con le loro idiosincrasie sintattiche e sistemi di macro profondamente incompatibili gli uni con gli altri (inclusi i moderni ambienti di *cross-development*, ormai sempre più necessari), poi la programmazione dei singoli chip periferici specializzati e infine le applicazioni, studiando e ristudiando il codice applicativo che ormai si trova in giro, tra disassemblati e rilasci al pubblico dominio, in quantità esorbitanti.

L'assioma fondamentale per la programmazione low level su C64 (ma vale anche per qualsiasi altra piattaforma) è che si impara leggendo **molto** codice Assembly avanzato scritto da programmatori professionisti e ben preparati: lo studio della mediocrità, dice il grandissimo Harold Bloom, non può che generare altra mediocrità. A sua volta, comprendere nei dettagli tale codice richiede lo studio di svariate tipologie di testi, senza limitarsi al solo Assembly che è il punto di arrivo.

I testi proposti sono in lingua inglese per una precisa scelta didattica ma anche per l'enormemente maggiore reperibilità rispetto ai testi in lingua italiana, talora anche molto validi ma estremamente ardui a reperirsi oggigiorno.

1.2 Breve bibliografia ragionata.

I primissimi due testi che occorre studiare *from cover to cover* sono, fin troppo ovviamente, i manuali originali Commodore: la *Guida per l'Utente del Commodore 64* e la *Guida di Riferimento per il Programmatore*. Nonostante i numerosi errori e refusi, talora presenti anche nel codice BASIC d'esempio e nei riferimenti, rimangono un viatico assolutamente imprescindibile per accostarsi alla macchina e iniziare a programmare in modo consapevole.

Occorrono poi, in parallelo, testi architetturali sul C64 e in particolare mappe di memoria accurate e commentate come quelle sponsorizzate dalla celeberrima rivista *Compute!*: [Lee87], [Hee85], [Hee84] magari affiancate anche da [PK84]. Certo non farà male anche la giustamente famosa guida alla riparazione del C64 di Art Margolis [Mar85], magari affiancata da [Bre85]: anche se non avete dimestichezza col saldatore, il testo di Margolis rimane uno dei riferimenti architetturali più autorevoli e presenta schemi elettrici affidabili, pressoché completi, per le principali revisioni del C64.

Tra i testi migliori per l'Assembly in quanto tale ci sono sicuramente il capolavoro di Zaks [Zak83] (come minimo sindacale, ma in realtà non mi farei mancare nessuno dei testi di tale autore) e quello di Leventhal [Lev86], come pure il testo di Andrews [And85] e quello, unico nel suo genere, di Lawrence e England [LE83]. Aggiungerei inoltre volentieri [Jon84] e [Smi85] e, per chi ha bisogno di partire veramente da zero, il Sinclair (proprio lui!) [Sin84] affiancato dal classico [Dav84] e da un testo di introduzione alla logica della programmazione come il notissimo Sprankle [SH11].

Ad un livello più avanzato ci sono sicuramente [Sut85], [Eng85] e una serie di testi più language-agnostic ma specifici sul tema ludico tra cui [Hei83], [Ple84], [Fab84], [Kru86].

Dopo lo studio dei testi consigliati, affiancato dalla lettura dei numerosi sorgenti pubblicamente disponibili nelle aree applicative più disparate (dalla *demoscene* al gaming, dai database al controllo di hardware esterno, alle telecomunicazioni...), il lettore avrà raggiunto un soddisfacente grado di familiarità con gli strumenti dello sviluppo low level per C64 e potrà proseguire per suo conto nello studio e nella ricerca sul campo.

Capitolo 2

Mastercode BASIC Extender.

2.1 Introduzione.

Il testo di cui discutiamo [LE83] è probabilmente noto ai più nell'edizione italiana Jackson Libri del 1985 «Il linguaggio macchina del Commodore 64» [LE85]. La «strana coppia» di autori è costituita da David Lawrence, programmatore BASIC professionista ed autore di numerosi testi di informatica, e Mark England, ingegnere elettronico esperto in Assembly.

L'idea di un libro dall'approccio così innovativo, che ancora oggi rimane unico nello sterminato panorama della letteratura tecnica applicativa per C64, è venuta a Mark England: evidentemente stufo (come molti di noi) di studiare quei testi della prima ondata sull'Assembly dei PET che si limitavano a duplicare le informazioni su istruzioni, opcode e operandi già presenti nel datasheet della CPU 6510 proponendo poi, nel migliore dei casi, una fantasiosa silloge di brevi routine assembly di scarso o nullo interesse pratico.

Altro punto critico della maggior parte dei testi sull'Assembly della prima metà anni Ottanta: presupponevano che il lettore avesse facile e immediato accesso ad un ambiente assemblatore, cosa in realtà nient'affatto scontata all'epoca, visti anche i costi di licenza dei principali ambienti di sviluppo disponibili in quel periodo.

Nasce così l'idea di sviluppare un completo assemblatore scritto interamente in BASIC, che il lettore potrà digitare gradualmente come salutare esercizio, aiutato anche da un verificatore di checksum: e questa è solo la prima parte del divertimento, perché gli «esempi applicativi», invece di essere il solito inconcludente florilegio di scarne routine aritmetiche e di I/O, sono in realtà una completa applicazione che guida il lettore nei meandri dell'interprete BASIC e dell'interazione al più basso livello col Kernal, aggiungendo ben 15 nuovi comandi e funzioni al CBM BASIC V2.

2.2 Mastercode.

La prima parte del testo è interamente occupata dai listati dell'Assembler Mastercode, il quale oggigiorno riveste unicamente interesse storiografico e didattico: digitarne il codice per intero, possibilmente su una macchina reale, costituirebbe un ottimo esercizio per molti neofiti. Si tratta di un ambiente modulare, basato su menu, fondamentalmente completo nell'impostazione ma molto limitato sia nella sintassi Assembly accettata, sia nelle prestazioni. Il suo pregio fondamentale è stato quello di conferire una totale autonomia e indipendenza al testo rispetto ai costosi ambienti commerciali all'epoca esistenti. Non occorre però neppure rimarcare che l'editor di memoria, il disassembler e il monitor risultano realmente utilizzabili solo in un ristretto novero di casi, dal momento che risiedono nella memoria dedicata al codice BASIC, e risentono di limitazioni drastiche sia rispetto ai monitor software dedicati, sia a maggior ragione rispetto a quelli su cartuccia, spesso perfino rilocabili e con caratteristiche decisamente avanzate. L'immagine 2.2.1 mostra l'aspetto originale del menu principale di Mastercode.

Figura 2.2.1: Il menù principale di Mastercode

2.3 BASIC Extender.

La programmazione di un BASIC Extender differisce profondamente dalla normale programmazione applicativa e richiede una solida consapevolezza di numerosi dettagli inerenti il codice in ROM (Kernal e BASIC). Una piena comprensione del codice richiederà, oltre allo studio del testo in questione, anche la consultazione di mappe di memoria e disassembly commentati come [Hee84], [Hee85], [Lee87] e una conoscenza dei meccanismi fondamentali dell'interprete residente.

I comandi e le funzioni del CBM BASIC V2 constano di parole chiave o *keyword*, quelle che normalmente digitiamo nei programmi e al prompt interattivo, come **LIST** o **PRINT**. Codeste stringhe però non vengono memorizzate letteralmente nel programma, per ovvi motivi di ottimizzazione in spazio: ciascuna di esse è invece sostituita da un codice univoco, rappresentato da un piccolo intero (un singolo byte) detto *token*. Tale valore numerico è caratterizzato dall'avere il bit più significativo posto a 1, per evitare ambiguità con altri elementi sintattici previsti dal linguaggio, come nomi di variabile o operatori. In fase di esecuzione, il *token* viene utilizzato per indicizzare correttamente la *jump table* o tabella dei vettori che contiene gli *entry point* delle routine assembly che implementano i singoli comandi BASIC; durante il listing il valore del *token* viene invece usato per un lookup inverso sulla tabella delle keyword, in modo che l'utente veda esattamente il comando che ha digitato in fase di immissione, rendendo del tutto trasparente il meccanismo di tokenizzazione o *crunching*.

Per gli usuali motivi di ottimizzazione in spazio (8kib di ROM erano pochi anche all'epoca), la tabella delle keyword è memorizzata usando uno dei tanti accorgimenti intelligenti dell'era Commodore: invece di sprecare un byte ponendolo a zero per ogni entry, rendendola così null-terminated o ASCII-zero che dir si voglia, il bit più significativo MSB dell'ultimo carattere è semplicemente posto a 1. Ciò, ovviamente, limita drasticamente ad ASCII-7 (in realtà PETSCII-7) il range dei caratteri effettivamente rappresentabili, il che tuttavia non costituisce affatto un problema per tale tipo di applicazione. Si tratta di una prassi talmente radicata che molti assemblatori offrono una apposita direttiva a tale scopo, come la *.shift* del Turbo Assembler.

Tenendo in mente quanto appena succintamente ricordato, i principali problemi che ci si trova ad affrontare nel progettare una espansione BASIC che rispetti il più possibile l'esistente sono i seguenti:

1) Il CBM BASIC risiede su una memoria di sola lettura (ROM o EPROM), tuttavia per rendere possibile l'interoperabilità con l'Extender occorre modificarne il codice in vari punti.

2) La tabella delle keyword standard ha dimensione limitata a 256 bytes, in quanto viene scandita entro l'interprete originale in modalità di indirizzamento indicizzato tramite il registro Y, ovviamente a 8 bit. Sfortunatamente, tale tabella risulta inutilizzabile ai fini di una espansione, in quanto già quasi totalmente occupata.

3) Occorre scrivere il codice Assembly per ciascuno dei nuovi comandi aggiunti all'Extender, in modo ovviamente compatibile con il firmware residente sulla macchina.

Il punto 1) (che su quasi ogni altro PET sarebbe l'epitaffio di ogni tentato progetto del genere) è pressoché insignificante nel nostro caso, dal momento che il C64 possiede infatti 64kib effettivi di memoria RAM contigua ed indiscriminatamente accessibile, alcuni segmenti della quale sono rimappati su ROM o port di I/O. Ma è possibile (e molto facile) copiare integralmente il contenuto della ROM nella RAM "sottostante", abilitando poi definitivamente quest'ultima e rendendo modificabile ogni byte copiato, ovviamente solo entro la sessione corrente.

La sola copia in RAM, tuttavia, non risolve il problema della tabella delle keyword e quello parallelo della corrispondente tabella dei vettori (come già ricordato, i puntatori alle singole routine corrispondenti a ciascuna keyword), la quale è parimenti quasi piena. Vi è poi un ulteriore problema: i *token* già occupati dalle keyword standard vanno dal 128 al 202, più 255 che rappresenta il pigreco. In considerazione di tutti questi vincoli, è strettamente necessario predisporre altre indipendenti tabelle, ed alterare parzialmente (seppure in modo minimo) il codice dell'interprete esistente, come già anticipato. Fortunatamente, l'intelligenza e la lungimiranza dei progettisti Commodore hanno fatto sì che buona parte del codice coinvolto sia revettorizzabile con estrema semplicità, in quanto puntato da variabili collocate entro i primi 2 kb di RAM, nell'area di sistema. In particolare, il vettore a 16 bit alla locazione 306h (ossia agli indirizzi 306 e 307 esadecimali) indirizza la routine PRTTOK, e risulta facilmente reindirizzabile per i nostri scopi. I *token* per le nuove keyword assumeranno valori da 203 in su, il che li rende immediatamente distinguibili da quelli standard.

Per le tabelle aggiuntive, la soluzione adottata dalla maggior parte dei BASIC Extender (incluso quello di cui discutiamo) consiste nell'uso di tabelle alternative indirizzate dinamicamente. Si deve partire dal presupposto che la routine di tokenizzazione («crunching» nella letteratura Commodore) del BASIC V2 non è particolarmente sofisticata, né tantomeno ottimizzata per le prestazioni: invece di ricorrere a strutture dati dedicate (come un albero binario) e algoritmi efficienti, si limita ad effettuare una banale ricerca lineare sulla tabella delle keyword usando come chiave la potenziale keyword attualmente in elaborazione. Un semplice contatore, inizializzato a monte del loop di scansione, viene incrementato ad ogni mancata corrispondenza, ed è proprio tale contatore che al termine della routine (salvo errori di sintassi) conterrà il valore del *token* corrispondente alla keyword esaminata nella linea BASIC corrente.

Oltre a questa struttura di base, decisamente elementare, vi sono poi nel codice dell'interprete residente varie euristiche per gestire le eccezioni e le singolarità del linguaggio: tra questi, il trattamento del separatore ':' che non è un *token* vero e proprio, ma viene gestito separatamente nel codice del cruncher/tokenizer.

La tabella aggiuntiva delle keyword viene indirizzata dinamicamente da una apposita routine CRUNCH nel codice di Lawrence & England, quando la keyword correntemente analizzata non ha trovato riscontro nella tabella originale. Di fatto, la scansione di ogni singola keyword e l'associazione con il relativo codice numerico (tokenizzazione) in una linea di codice seguono uno schema logico estremamente semplice e lineare:

1. Il tokenizer confronta la stringa della potenziale keyword presente nella linea di codice correntemente analizzata con le keyword della tabella originale del BASIC V2 (con inizio alla locazione $A09E): END, FOR, NEXT, DATA, ...

2. Se il confronto va a buon fine, si esce dal loop principale del tokenizer. A quel punto il numero contenuto in $0B (con l'aggiunta di un offset pari a $80, che per convenzione caratterizza i token BASIC) è il valore del *token* corrispondente al comando trovato nella linea di codice BASIC appena analizzata e viene quindi memorizzato assieme al resto della riga di codice.

3. Ad ogni confronto fallito il *token counter* (contenuto nella locazione di pagina zero $0B) viene incrementato e si passa alla keyword successiva in tabella.

4. Se al termine del loop la stringa non ha trovato corrispondenza, la scansione della tabella delle keyword è giunta a fine e il *token counter* ha raggiunto il suo valore massimo per il BASIC standard, pari nel nostro caso a 202 ($CA). Normalmente a questo punto si avrebbe un errore sintattico, ma in questo caso potrebbe ancora trattarsi di un nuovo comando. Una piccola modifica al codice ROM originale impone quindi di richiamare la routine CRUNCH sopra menzionata, la quale sostituisce dinamicamente nel tokenizer l'indirizzo di partenza della tabella delle keyword con il valore della label KEYWRD nel nostro sorgente (di norma $C000) e lo costringe ad eseguire un nuovo loop di scansione, ricominciando dal punto 1 ma con una differente tabella delle keyword e **senza azzerare il token counter**. In questo modo, per costruzione, è garantita l'assegnazione di token nell'intervallo riservato all'Extender, dal 203 in poi.

5. Se uno dei confronti all'interno della tabella delle keyword estese va a buon fine, siamo in presenza di una keyword dell'Extender, che viene regolarmente tokenizzata. In caso contrario: **SYNTAX ERROR**.

In fase di esecuzione viene implementato un meccanismo analogo per la scelta della tabella dei vettori, in questo caso semplificato dalla immediata discriminazione tra gli intervalli di valori dei *token* standard ed estesi.

La soluzione originariamente prevista da Lawrence e England prevedeva un totale di 15 nuove keyword, di cui tre funzioni, oltre a una coppia di istruzioni puramente rappresentative (FAST e SLOW) che fin dall'inizio l'autore del presente articolo non ha mai ritenuto opportuno implementare. La tabella seguente riassume sinteticamente le nuove funzionalità disponibili: poiché alcuni comandi hanno un numero elevato di parametri, per esigenze tipografiche si è scelto di ometterli quando potevano compromettere la leggibilità.

UNDEAD	Ripristina un programma BASIC dopo un (accidentale) comando NEW.
SUBEX	Elimina l'ultimo indirizzo di ritorno dallo stack.
RKILL	Compressione sorgente BASIC, elimina spazi e REM.
DOKE <addr>,<val>	Versione a 16 bit della POKE.
PLOT <row>,<col>	Posiziona il cursore alle coordinate specificate.
DELETE <inizio>,<fine>	Cancella intervalli di linee contigue da un sorgente BASIC.
BSAVE <...>	Salva un blocco di memoria su file.
BLOAD, BVERIFY <...>	Carica (verifica) un blocco di memoria da file.
MOVE <to>,<from>,<len>	Copia un blocco di memoria ad un nuovo indirizzo.
FILL <start>,<len>,<val>	Riempie un blocco di memoria con il valore dato.
RESTORE <line>	Versione estesa della RESTORE originale.
VARPTR(<var>)	Restituisce un puntatore alla variabile referenziata.
YPOS()	Complementare alla POS() del BASIC V2, restituisce la riga del cursore.
DEEK(<addr>)	Complementare a DOKE, è una PEEK() a 16 bit.

Nella implementazione originale era previsto, oltre al file binario prodotto da Mastercode e contenente il codice della vera e propria estensione BASIC, un loader scritto in BASIC V2 che si occupava del caricamento da disco (o nastro), della copia da ROM a RAM del BASIC V2 originale, della rilocazione dell'estensione a partire dall'indirizzo $C000 e della revettorizzazione.

2.4 Porting ed estensione del progetto.

Nel 1985, dopo avere pedissequamente digitato tutto il codice originale proposto dal testo, si è provveduto al porting dei sorgenti dell'Extender su Turbo Assembler, in un unico file. Tale lavoro mirava a superare le inerenti limitazioni sintattiche di Mastercode, rendendo più facile la manutenzione e l'espansione del progetto. Tutte le operazioni di caricamento e predisposizione dell'ambiente possono infatti essere centralizzate ed eseguite direttamente in Assembly senza il minimo problema, con un singolo eseguibile «mascherato» da programma BASIC ovvero caricato interamente da disco a partire dalla locazione di default $0801 e successivamente eseguito con un semplice comando RUN. Il codice di startup elaborato all'epoca è qui riportato in una moderna versione per CBM Prg Studio.

```
;************************************
; Codice "universale" per startup BASIC
;************************************
*=$0801
;------------------------------------
        WORD BASEND                     ; Indirizzo della prossima linea
        WORD 1985                       ; Numero di linea -> anno di stesura...
        BYTE $9E                        ; Token per "SYS"
        TEXT "2102:"                    ; Indirizzo di avvio: $0836
        BYTE $8F                        ; Token per "REM"
        BYTE $22                        ; Token per '"'
; Rendo illeggibile il listing con una serie di DEL ($14)
        BYTE 20,20,20,20
        BYTE 20,20,20,20
        BYTE 20,20,20,20
; Solo la stringa seguente apparira' nel listato:
        TEXT "(c) lawrence & england ***"
        BYTE 0                          ; Terminatore di linea
BASEND  WORD 0                          ; Terminatore di programma
;------------------------------------
;************************************
;** Inizio codice applicazione      **
;************************************
```

Esaminando il codice il lettore più smaliziato noterà come, già in quel lontano 1985, l'allora giovane programmatore univa costantemente il divertimento all'apprendimento. Andando oltre la banale funzionalità di avvio da BASIC del codice Assembly, il listato con pochi bytes aggiuntivi risulta anche «protetto» da sguardi indiscreti (un programmatore minimamente esperto userà comunque un monitor con disassembler per esaminare a piacimento il codice dopo il caricamento) con una tecnica all'epoca universalmente diffusa per i software da edicola e commerciali. Se si esegue un LIST subito dopo il caricamento del codice, si ottiene unicamente quanto illustrato in figura 2.4.1.

Figura 2.4.1: Il listato dell'extender, abilmente mascherato.

La sezione di codice successiva si occupa delle operazioni preliminari necessarie alla modifica del BASIC residente su ROM: copia il BASIC nella RAM sottostante e in seguito modifica alcuni vettori, reindirizzando alle nuove routine dell'espansione che ne permettono la coesistenza col BASIC originale.

```
;************************************
; Espansione BASIC / RELEASE 2.0
;************************************
```

```
;-------------------------------------------------
; Rilocazione codice in $C000
;-------------------------------------------------
START   LDA #<LDR_END+1
        STA MVSTART
        LDA #>LDR_END+1
        STA MVSTART+1
        LDY #$00
        STY MVDEST
        LDA #$C0
        STA MVDEST+1
        LDX #LEN

MOVE1   LDA (MVSTART),Y
        STA (MVDEST),Y
        INY
        BNE MOVE1
        INC MVSTART+1
        INC MVDEST+1
        DEX
        BNE MOVE1
;-------------------------------------------------
; Modifiche all'interprete BASIC C64:
; copia BASIC ROM in RAM e revectoring
;-------------------------------------------------
        LDA $01         ; Seleziona la ROM BASIC
        ORA #$01
        STA $01
        LDY #$00

; Non occorre modificare il registro $01 ad ogni lettura:
; l'hardware del C64 e' progettato per scrivere
; comunque su RAM, anche se viene selezionata la ROM.
BSTART  LDA BASIC_START,Y
BDEST   STA BASIC_START,Y
        INY
        BNE BSTART
        INC BSTART+2
        INC BDEST+2
        LDX BDEST+2
        CPX #$C0
        BNE BSTART

MODIFY  LDA $01         ; Seleziona la RAM in $A000-$BFFF
        AND #$FE
        STA $01
        LDX #_JMP
        STX $A7E1
        LDA #<EXECUT
        STA $A7E2
        LDA #>EXECUT
        STA $A7E3
        STX $AFAD
        LDA #<FUNEVL
        STA $AFAE
        LDA #>FUNEVL
        STA $AFAF
        STX $A604
        LDA #<CRUNCH
        STA $A605
        LDA #>CRUNCH
```

```
          STA $A606
          LDA #<PRTTOK
          STA $0306
          LDA #>PRTTOK
          STA $0307
          LDA #<RESTORE-1
          STA $A024
          LDA #>RESTORE
          STA $A025
           ...
LDR_END   BYTE 0
```

L'estensione BASIC, nella nuova versione per Turbo Assembler, è poi stata arricchita con ulteriori nuovi comandi, di ispirazione didattica o mutuati da altre estensioni BASIC. Vale la pena di sottolineare nuovamente che questo progetto ha un valore precipuamente didattico e non presenta la benché minima velleità di porsi al livello di un Simon's BASIC o di qualsiasi altro Extender commerciale.

CLS	Cancella lo schermo.
COLOR <sfondo>,<bordo>,<testo>	Imposta i colori usando i codici CBM.
COMMANDS	Elenca a video tutti i nuovi comandi e funzioni dell'estensione.
FCHR <...>	Riempie con un carattere un'area rettangolare a video.
FCOL <...>	Riempie con il colore dato un'area rettangolare a video.
LOWER	Imposta il set di caratteri minuscolo.
NORM	Comoda scorciatoia per impostare colori e case di default.
REP	Imposta ciclicamente l'autorepeat della tastiera.
SCRINV <...>	Inverte sfondo e testo in un'area rettangolare a video.
SFILL <...>	Versione specializzata di FILL per la memoria video.
UPPER	Complementare a LOWER, imposta il set maiuscolo.

Si è inoltre introdotto un comando OLD, semplice e più sobrio sinonimo per UNDEAD. L'immagine seguente mostra il risultato dell'esecuzione di COMMANDS dopo il caricamento dell'estensione.

2.5 Un curioso bug.

All'epoca l'aggiunta di nuovi comandi all'Extender rivelò presto un problema non previsto dagli autori, in quanto il bug apparentemente non era riproducibile nel contesto delle 14+3 keyword originali. L'immagine 2.5.1 mostra in modo immediato il problema.

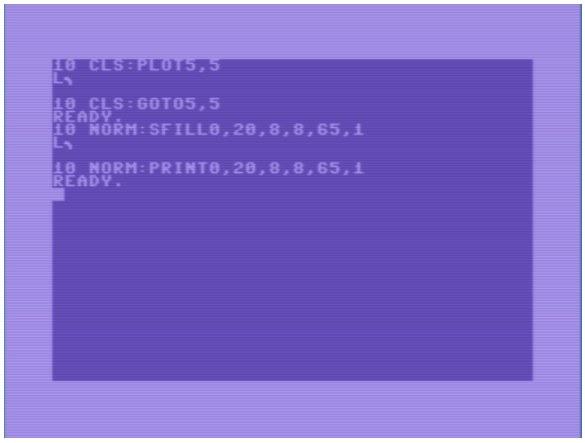


Figura 2.5.1: Un bug decisamente imprevisto dagli autori.

Come si vede, esisteva un problema di errata tokenizzazione: il comando PLOT viene tokenizzato e quindi listato come GOTO, mentre la PRINT prende il posto di SFILL, e così via. Questo nel caso in cui sulla riga di programma siano presenti due o più nuove keyword.

In base ai meccanismi accennati al paragrafo §3, si possono evidentemente creare situazioni di concatenazione di nuovi comandi su una singola linea a causa di cui l'associazione tra *token counter* e indirizzo della tabella delle keyword non è più sincronizzata: ciò avviene perché esistono (rari) percorsi di esecuzione nel codice del tokenizer ROM che non transitano dal punto di iniezione in cui viene richiamato il codice aggiuntivo CRUNCH previsto dagli autori. Ci si trova quindi in una situazione incongruente in cui il *token counter* viene inizializzato a zero, ma nel codice del tokenizer viene ancora puntata la tabella delle keyword estese $C000 e non quella originale: ciò genera esattamente il tipo di errore evidenziato.

Effettuando tramite monitor un accurato reverse engineering del codice ROM interessato all'analisi di una linea di codice BASIC e alla tokenizzazione, si è quindi determinata la necessità di un ulteriore punto di iniezione nel codice del tokenizer stesso, in modo da garantire in qualsiasi caso la sincronizzazione tra l'inizializzazione del *token counter* (contenuto come abbiamo visto nella locazione di pagina zero $0B) e l'indirizzo della tabella delle keyword originali alla locazione $A09E. Il codice seguente, aggiunto al sorgente dell'estensione BASIC riveduta e corretta, risolve efficacemente anche tale bug.

```
;*****************************************************
;** Il codice seguente non fa parte del testo ed
;** e' stato aggiunto per risolvere un curioso bug
;** del BASIC Extender. L'associazione originale tra
;** tabella delle nuove istruzioni e contatore usato
;** dal tokenizer BASIC non e' sufficientemente
;** robusta, perché esiste almeno un caso in cui
;** si crea una desincronizzazione, ad esempio
;** digitando una linea di programma come segue:
;** 10 CLS:SFILL 0,20,8,8,65,1
;** il secondo comando viene tokenizzato in modo
;** errato. Questo avviene, in breve, perche' gli
;** autori non hanno considerato tutti i possibili
;** percorsi di esecuzione nel tokenizer ROM,
;** cio' che di norma si attua effettuando una
;** analisi full code coverage con un tracer.
;** Per ovviare basta comunque un ulteriore punto
;** di iniezione, in corrispondenza della
;** inizializzazione del token counter. Si aggiunge
;** una brevissima routine che garantisce che
;** all'azzeramento del counter la tabella corrente
;** delle keyword sia quella originale del BASIC.
;*****************************************************
        STX $A5AE
        LDA #<TOKSTR2
        STA $A5AF
        LDA #>TOKSTR2
        STA $A5B0
...
```

```
;*************************************************
;** Routine aggiuntiva per eliminare il bug
;** nel tokenizer.
;*************************************************

TOKSTR2   JSR PUTREG
          LDA #$A0
          LDX #$9E
          JSR TOKSTR
          JSR GETREG
          LDY #$00
          STY $0B
          JMP $A5B2
;*************************************************
```

2.6 Conclusioni.

Si è brevemente presentata la storia del BASIC Extender didattico di Lawrence & England, accompagnata da una succinta descrizione di una possibile espansione di notevole interesse per l'apprendimento e lo studio dell'Assembly. Si è anche illustrata l'insorgenza di un bug progettuale del tutto imprevisto e la sua risoluzione tramite reverse engineering, sempre lecitamente consentito quando si tratti di garantire l'interoperabilità del codice.

Il codice Assembly allegato, predisposto per il moderno ambiente di cross-development CBM Prg Studio, viene messo a disposizione per lo studio e la sperimentazione, con l'augurio che molti lettori studiandolo si sentano invogliati a progettare ed implementare nuovi comandi per migliorare i propri skill in Assembly e comprendere meglio l'interazione col codice BASIC e Kernal presenti in ROM. Il codice è brevemente commentato in alcuni punti salienti e quasi tutti gli indirizzi di memoria significativi (originariamente tutti hardcoded, senza distinzioni) sono referenziati tramite label descrittive, ma ci si attende comunque che il lettore studi il sorgente avendo a disposizione almeno i testi elencati in bibliografia per una comprensione puntuale.

Capitolo 3

Il giro del cavallo

3.1 Introduzione.

Nel 1985 veniva stampato un testo unico nel suo genere: «Artificial Intelligence Projects for the Commodore 64» [O'M85]. Presentiamo uno degli esempi più noti tratti da tale libro, il che ci fornisce un'ottima occasione per parlare di un problema scacchistico che ha interessato nei secoli matematici del calibro di Eulero, Legendre, Vandermonde e soprattutto della tecnica dimostrativa computazionale con la quale recentemente è stato risolto. Il testo (più unico che raro) tratta della IA così come conosciuta nei primi anni Ottanta, applicata al C64: uno dei primi testi applicativi di larga diffusione ad introdurre tecniche all'epoca altamente innovative, che solo molti anni dopo sarebbero state trattate sistematicamente in volumi di meritata fama come [RN09]. Il linguaggio, al solito, è intenzionalmente informale, divulgativo e didascalico, scevro da particolari pretese di rigore, al fine di raggiungere il più vasto uditorio informatico possibile.

3.2 Knight's tour.

Il problema originale, la cui prima traccia storicamente certa risale circa all'850 d.C., riguarda il pezzo degli scacchi noto come cavallo (*knight* nel mondo anglosassone), che notoriamente si muove lungo l'immaginaria diagonale di un rettangolo con lati di due e tre caselle rispettivamente. Si tratta di trovare (almeno) un percorso che porti il cavallo ad occupare tutte le caselle della scacchiera partendo da una casella qualsiasi e passando una e una sola volta su ciascuna altra casella. Se l'ultima mossa (anche lasciata implicita) porterebbe il pezzo ad occupare nuovamente la casella di partenza, il percorso viene detto «chiuso»: in caso contrario si parla di percorso «aperto».

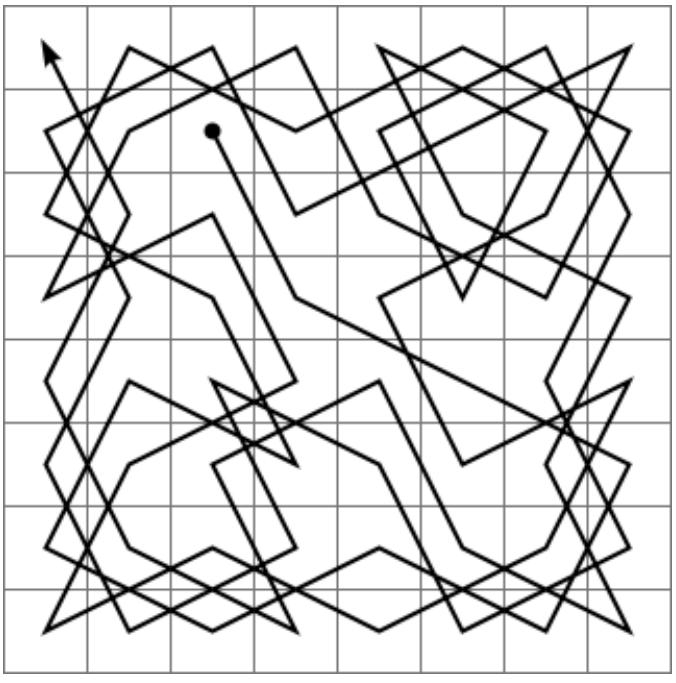


Figura 3.2.1: Esempio di percorso chiuso diretto.

Generare un singolo percorso, chiuso o aperto che sia, è un problema relativamente difficile che ha interessato molti tra i più grandi ingegni matematici, inclusi lo svizzero Eulero (Leonhard Euler, 1707–1783), Abraham de Moivre (1667–1754), Adrien-Marie Legendre (1752–1833), Alexandre-Théophile Vandermonde (1735–1796) e numerosi altri. La fama del problema è tale che si può tranquillamente asserire che non esista testo di ludomatematica e raccolta di rompicapo logici che non ne faccia menzione, in qualche forma. Nel 1823 H. C. von Warnsdorff descrisse una euristica [AW92] ancor oggi utilizzata, basata sulla scelta di una casella privilegiata in base al minimo numero di mosse disponibili: tale euristica, nonostante la sua scarsa efficienza computazionale, ha conosciuto una certa fortuna anche nell'epoca d'oro del retroprogramming, essendo usata come base di una implementazione in C (Acorn C, per l'esattezza) all'interno di un famosissimo testo di rompicapo informatici [Dal84]. Il lettore interessato troverà una trattazione pressoché completa della storia del problema in `http://www.mayhematics.com/t/1n.htm` e una raccolta di fonti in `http://www.velucchi.it/mathchess/knight.htm`.

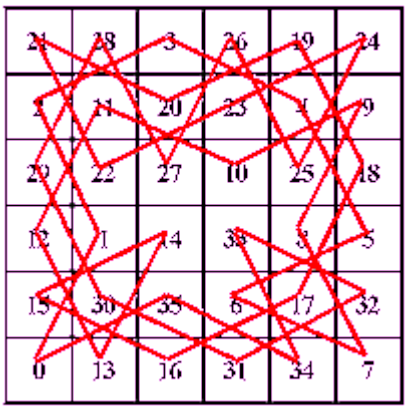


Figura 3.2.2: Esempio di percorso chiuso su una scacchiera non standard 6×6 con numerazione progressiva delle caselle visitate. A ciascuna cella viene assegnato un ordinale naturale che ne contraddistingue la posizione nella sequenza delle visite. In questo caso si è scelto di partire da zero, in modo che l'ordinale risponda alla domanda «quante celle precedono quella corrente?». Nell'ordine è implicito anche il verso di percorrenza, pertanto anche questo percorso, come quello illustrato in fig. 3.2.1, è un *percorso chiuso diretto*.

Dal punto di vista combinatorio e computazionale il problema si presta ovviamente a estensioni e generalizzazioni, tutte debitamente studiate da lungo tempo e oggetto di uno specifico teorema dovuto ad A. Schwenk [Sch91] che determina l'esistenza dei percorsi in base alle dimensioni generiche $m \times n$ della scacchiera. In termini generali, chiedere se con un cavallo sia possibile occupare in successione tutte le $n \times n$ caselle di una scacchiera equivale a ricercare l'esistenza di un cammino hamiltoniano nel grafo equivalente, nel quale gli archi rappresentano le mosse e i nodi le caselle della scacchiera. Tradurre il problema in termini di grafi hamiltoniani significa quindi verificare se esiste nel grafo considerato almeno un cammino chiuso detto «ciclo hamiltoniano» che contiene ogni vertice del grafo una ed una sola volta; nel caso di percorso «aperto» come definito sopra, si tratta invece di cercare un «cammino hamiltoniano». Purtroppo è ben noto che ambedue i problemi, nella loro forma generale, sono NP-completi: la buona notizia è che invece la specifica istanza costituita dal problema di nostro interesse è computabile in tempo lineare, come dimostrato in [CHMW94]. Ben più difficile risulta il problema dell'enumerazione completa di tutti i possibili percorsi chiusi e aperti, tanto che sono occorsi vari secoli per addivenire ad una risposta definitiva inerente il problema originale e quindi la scacchiera standard 8×8: i possibili *percorsi chiusi diretti* (ossia distinti anche per il *verso* in cui si attraversa una data sequenza di caselle, come quello in fig. 3.2.1) sono ben 26.534.728.821.064 e lapalissianamente, se ignoriamo il verso di percorrenza (si parla in questo caso di percorsi *non diretti*), tale valore si dimezza. Il conteggio è stato confermato circa venti anni fa unicamente tramite l'uso di sofisticate tecniche computazionali per l'enumerazione esaustiva. Nonostante l'enorme mole di energie profuse e la grande notorietà del problema da almeno undici secoli, infatti, solo in tempi recenti l'enumerazione dei percorsi chiusi ha trovato soluzione [Weg00, DM18], peraltro dopo una sfortunata falsa partenza a metà anni Novanta dovuta ad un errore di implementazione, mentre è recentissimo un tentativo di enumerazione computazionale dei percorsi aperti [MC15].

3.3 L'enumerazione delle soluzioni.

Un valore dell'ordine di ventiseimila miliardi di configurazioni per i giri del cavallo lascia facilmente intuire che una enumerazione esaustiva richiede elevate risorse computazionali per essere completata in tempi accettabili, come pure l'uso di appropriate strutture dati. In questo caso, a metà anni Novanta, sono stati utilizzati dei BDD (*Binary Decision Diagrams*, si veda [Knu11]) e loro varianti, l'unica struttura dati in grado di garantire al tempo stesso la compattezza e la facilità di elaborazione richieste per una enumerazione massiva del genere. I BDD sono stati inizialmente concepiti come formalismo per la verifica di software e firmware per applicazioni critiche, tuttavia il loro recepimento a livello applicativo in tale settore è stato piuttosto tardo, mentre paradossalmente si sono quasi immediatamente diffusi nel mondo dei CAD EDA per la verifica a livello hardware di chip logici VLSI come microcontroller, SoC, CPU e altri chip ad elevata complessità. Senza scendere nei dettagli, per non appesantire la trattazione, si propone solo un intuitivo esempio di elementare derivazione di OBDD (Ordered BDD) da una funzione booleana di tre variabili espressa tramite tabella di verità, nelle figure 3.3.1 e 3.3.2, tramite applicazione di una terna di regole di semplificazione e sintesi (che non specifichiamo esplicitamente). Ciò che desideriamo qui è solo incuriosire il lettore e trasmettere l'idea che la rappresentazione con OBDD consta nella quasi totalità dei casi di un numero nettamente inferiore di nodi rispetto all'albero completo di partenza, pur rappresentando esattamente la medesima funzione booleana, senza

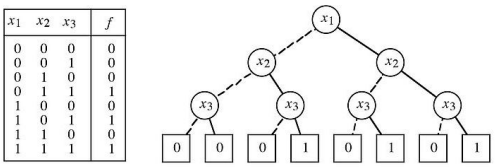


Figura 3.3.1: Costruzione di un albero di decisione binario BDT dalla relativa tabella di verità. Le linee tratteggiate, per convenzione, indicano il percorso da seguire quando la variabile nel nodo soprastante ha valore *0* o *FALSE*.

perdita d'informazione: ciò è sufficiente ad intuirne la maggiore efficienza in spazio, come pure una generale riduzione del tempo complessivo di attraversamento ed elaborazione.

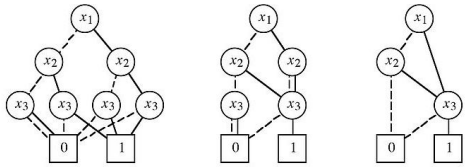


Figura 3.3.2: Applicando sequenzialmente le tre regole di eliminazione delle duplicazioni e delle ridondanze al BDT di fig. 3.3.1 si ottiene l'OBDD riportato più a destra.

La dimostrazione computazionale di Wegener e Löbbing sul numero complessivo di percorsi chiusi del cavallo ha speciale rilevanza anche perché è parte di un importantissimo filone che annovera alcuni dei più grandi successi dell'uso del calcolatore in campo dimostrativo: una panoramica su tali dimostrazioni ottenute tramite il calcolatore sarebbe del tutto fuori luogo nel presente contesto, ma possiamo almeno citare il teorema dei quattro colori [AH89] o il problema delle arance di Keplero [HAB+15], forse tra i più emblematici e discussi esempi. Non occorre certo che il lettore possieda una specializzazione in teoria della dimostrazione e dei modelli per intuire che in questi casi il problema principale, rispetto ad una usuale dimostrazione matematica come concepita fin dai tempi di Euclide, si sposta dal garantire la correttezza del procedimento logico e la consequenzialità tra i singoli passaggi della dimostrazione al dover garantire la correttezza di uno (o più) programmi software e relativo hardware. Gli approcci normalmente seguiti in questi casi si basano indirettamente sulla molteplicità di prove indipendenti effettuate con hardware e software distinti, oppure (più frequentemente) sulla *verifica formale* diretta della correttezza del software e del funzionamento dell'hardware, argomento interessante e vastissimo quanto misconosciuto da una maggioranza di addetti ai lavori nel mondo IT. Il problema di Keplero appena citato è un ottimo e ben noto esempio di quest'ultima strategia di convalida formale della metodologia di prova basata su hardware/software.

3.4 Cenni sui metodi formali.

I metodi formali, come appena accennato, consentono di *garantire* in modo logicamente rigoroso che un software, firmware o anche un hardware rispettino una specifica, a sua volta formalizzata in modo non ambiguo. I linguaggi utilizzati in questo ambito sono detti appunto linguaggi formali e sono 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 35-40 anni di studio sul software engineering e sui metodi informazionali, nella branca meglio nota come VVT ossia (Software) Validation, Verification and Testing:

- Linguaggi algebrici come LARCH, OBJ, LOTOS, CASL;
- Linguaggi di modellazione come Z, VDM, CSP;
- Reti di Petri, BDD, MDD, altri automi a stati finiti;
- Logiche temporali lineari, circolari, branching (CTL*, da cui derivano CTL e LTL...);
- Linguaggi più recenti, come OCL o Trio, basati sostanzialmente sui linguaggi di modellazione ibridati con altri simbolismi semiformali (es. UML). A tale proposito, il lettore presti particolare attenzione al fatto che sia UML che i diagrammi di flusso ISO 5807:1985 in tutte le loro varianti (es. Jackson) **non sono in alcun modo linguaggi formali**.

Esistono poi ulteriori approcci complementari alla verifica del software, sostanzialmente basati sulle semantiche formali dei linguaggi di programmazione, tra i quali spicca senz'altro l'*interpretazione astratta*, implementata in alcuni framework software di grande successo.

I linguaggi menzionati, e i loro 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. Si noti tuttavia che l'uso di linguaggi formali fornisce le garanzie richieste *senza l'uso di test*, che sono notoriamente insufficienti dal punto di vista logico a garantire il funzionamento a specifica di un software, come già sottolineava il noto Edsger Dijkstra (1930–2002) negli anni Settanta. Non a caso infatti Dijkstra, fisico teorico dedicatosi per l'intera carriera all'informatica, è considerato uno dei più importanti precursori dell'uso sistematico dei metodi formali.

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. Un linguaggio formale nasce sostanzialmente per assolvere a due scopi:

1) Fornire una descrizione *matematicamente corretta* e non ambigua delle specifiche (specifica 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).

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.

Siamo consapevoli del fatto che molti lettori vedono menzionati qui per la prima volta i principali linguaggi formali e soprattutto le loro straordinarie capacità nel garantire la correttezza di software, firmware e hardware. Chi opera nel mercato mainstream ed è abituato a considerare inevitabile l'uso continuato di espressioni come «bug», «errore software», «patch», «vulnerabilità» ed altro può trovare difficile perfino concepire una simile realtà. Tuttavia, i metodi e i linguaggi formali sono la normalità quotidiana in tutti i settori nei quali sono fondamentali l'elevata affidabilità e sicurezza del sistema, definite quantitativamente dalle varie normative come EN 60880 (nucleare ed energetica), MIL-STD-498 (sistemi d'arma e tattici), ESA PSS05 (aerospaziale), EN 50128 (ferroviario), EN 61508 (sicurezza funzionale in ambito industriale) e le sue numerose derivate quali EN 61511 per l'industria di processo, EN 60601 per l'elettromedicale, etc.. I metodi formali, oltre ad essere il principale pilastro della proverbiale affidabilità dei sistemi embedded critici e anche di alcuni sofisticati sistemi di elaborazione mission-critical in ambito bancario o assicurativo, sono anche estremamente importanti in ambito delle organizzazioni tecnico-scientifiche come IEEE, ACM, ISO ed in generale nella definizione degli standard.

La recente normativa avionica RTCA DO-333 (applicata in realtà in un'ampia casistica di sistemi in vari settori industriali, militari e civili), la più rigorosa attualmente in vigore, regola e armonizza l'utilizzo di metodi formali nell'intero ciclo di sviluppo hardware, firmware e software. Sperabilmente avremo modo di tornare sull'argomento.

3.5 Il programma del testo.

Si presenta nel seguito il completo listato in BASIC V2 così come proposto da [O'M85]. La data del testo, le inerenti limitazioni di potenza elaborativa della piattaforma e la stessa posizione del problema nel libro (si tratta in pratica del terzo listato del primo capitolo, dopo due semplici variazioni sulle torri di Hanoi ideate da F. Lucas nel diciannovesimo secolo) fanno immediatamente pensare che siamo ben lontani dalle vette di sofisticazione combinatoria ed enumerativa accennate nell'excursus sul problema affrontato nei paragrafi precedenti. Infatti, con doverosa onestà intellettuale, l'autore stesso sottolinea che non vi è traccia di «intelligenza» nel programma, in senso strettamente algoritmico: si tratta di un singolo percorso chiuso diretto pregenerato, i cui dati sono memorizzati negli statement DATA alle linee $601 \div 608$. L'unica elaborazione effettuata è il ricalcolo della sequenza data la casella di partenza. Non occorre tuttavia sottolineare come, trattandosi di un percorso chiuso che per definizione tocca tutte le 64 caselle, è in realtà del tutto indifferente da quale di esse si scelga di partire.

«This program is something of an expert system because the computer is able to quickly solve the problem from any first position.» ([O'M85], pag. 16). L'autore conclude poi la sua descrizione del listato con la considerazione «The reason this program is included is that the compuler uses one set of data to solve a wide range of possibilities. I think that makes it somewhat intelligent. Intelligence is really making the best use of stored information.» (*ibidem*, pag. 20).

Mantenendo ben chiare queste premesse, il programma ha pur sempre una sua rilevante valenza didattica grazie alla grafica accattivante che mostra in tempo reale un'animazione con la successione completa delle mosse, marcando ogni casella con l'ordinale che la contraddistingue nel percorso e tenendo conto della locazione di partenza scelta arbitrariamente dall'utente.

```
10 :::::::::::::::::::::::::::::::::::
20 REM KNIGHT'S TOUR
30 REM
40 REM COPYRIGHT 1983, TAB BOOKS INC.
50 REM WRITTEN FOR THE COMMODORE 64
60 :::::::::::::::::::::::::::::::::::
70 REM **** MAIN PROGRAM ****
80 GOSUB200:REM DISPLAY INSTRUCTIONS
90 GOSUB400:REM DEFINE GRAPHICS
100 GOSUB600:REM DISPLAY SOLUTION
110 OPEN1,0:INPUT#1,N$:CLOSE1:PRINTCHR$(147):POKE53281,6:POKES,0:POKES+1,0:END
120 :::::::::::::::::::::::::::::::::::
```

```
200 REM *** INSTRUCTIONS ***
210 PRINTCHR$(147);:REM CLS
220 PRINT:PRINTTAB(13);"knight's tour":PRINT
230 PRINT"     this program displays a knight's"
240 PRINT"tour that is, a knight in the game"
250 PRINT"of chess can jump on all 64 squares"
260 PRINT"of a chess board without landing on"
270 PRINT"any square more than once."
280 PRINT
290 OPEN1,0
300 PRINT"enter knight's starting position.":PRINT
310 PRINT"enter the row.....:";:INPUT#1,R:PRINT
320 PRINT"enter the column..:";:INPUT#1,C:PRINT
330 CLOSE 1
340 IFR<1ORR>8ORC<1ORC>8THEN290
350 RETURN
360 ::::::::::::::::::::::::::::::::
400 REM *** DEFINE GRAPHICS ***
410 PRINTCHR$(147);:REM CLS
420 FOR I=0TO63:READJ:POKE832+I,J:NEXTI
430 DATA,,,,,,,,,,
440 DATA,8,,,30,,,63,,,127,
450 DATA,111,,,15,,,31,,,62,
460 DATA,62,,,28,,,62,,,127,
470 DATA,127,,,,,,,,,,,,,,
480 S=53248
490 POKES+21,1:POKES+39,1
500 POKES+23,1:POKES+29,1:POKE2040,13:POKE53281,14
510 P=1016:L=-1:FORJ=1TO8:FORM=1TO3
520 P=P+7:FORK=1TO8
530 FORN=1TO4:P=P+1:POKEP,160:POKEP+54272,ABS(L)*3
540 NEXTN:L=NOTL:NEXTK:P=P+1:NEXTM
550 L=NOTL:NEXTJ:RETURN
560 ::::::::::::::::::::::::::::::::
600 REM *** DISPLAY SOLUTION ***
601 DATA 3,42,5,20,37,40,15,18
602 DATA 6,21,2,41,16,19,36,39
603 DATA 43,4,57,54,59,38,17,14
604 DATA 22,7,62,1,56,53,60,35
605 DATA 49,44,55,58,61,64,13,28
606 DATA 8,23,48,63,52,29,34,31
607 DATA 45,50,25,10,47,32,27,12
608 DATA 24,9,46,51,26,11,30,33
610 DIMBD(8,8),MD(8,8)
615 FORJ=1TO8:FORK=1TO8:READMD(J,K):NEXTK,J
620 N=0:X1=0:Y1=0
625 FC=MD(R,C)-1
626 FORJ=1TO8:FORK=1TO8:BD(J,K)=MD(J,K)-FC:IFBD(J,K)<1THENBD(J,K)=BD(J,K)+64
627 NEXTK,J
630 N=N+1:FORJ=1TO8:FORK=1TO8:IFBD(J,K)=NTHENR=J:C=K
633 NEXTK,J
635 P=1065+4*(C-1)+120*(R-1)
640 V=48+INT(N/10):POKEP,V
650 V=48+N-10*INT(N/10):POKEP+1,V
660 X2=32*C-18:Y2=24*R+16:DX=X2-X1:DY=Y2-Y1
670 IFABS(DY)>ABS(DX)THEN850
680 FORJ=SGN(DX)TODXSTEPSGN(DX)
690 X2=X1+J:Y2=Y1+J*DY/DX
700 POKES,X2:POKES+1,Y2
710 NEXTJ:IFN=64THENRETURN
750 X1=X2:Y1=Y2:GOTO630
```

```
850 FORJ=SGN(DY)TODYSTEPSGN(DY):Y2=Y1+J:X2=X1+J*DX/DY:GOTO700
860 :::::::::::::::::::::::::::::::::::
```

Si nota immediatamente come il semplice programma è organizzato in maniera molto ordinata, con una struttura fortemente modulare e una routine principale concisa e ben leggibile. Alle linee $200 \div 350$ si stampano le semplici istruzioni e si chiedono le coordinate della cella di partenza (si noti l'uso dell'istruzione `OPEN 1,0` per la lettura diretta della tastiera). Le linee $400 \div 500$ gestiscono i quattro sprite che rappresentano graficamente il cavallo, mentre le linee immediatamente successive $510 \div 550$ disegnano la scacchiera usando dei blocchi di 3×4 caratteri a colori alternati. Le linee $601 \div 608$ contengono, come già accennato, una soluzione precalcolata con partenza al centro della scacchiera (riga 4, colonna 4): è il codice alle linee $620 \div 627$ che effettua il semplice riordino della sequenza sulla base delle coordinate di partenza immesse dall'utente. Le linee $630 \div 850$ gestiscono l'intera animazione: calcolano le coordinate alle quali stampare a video l'ordinale della mossa corrente, effettuano tale stampa, tracciano una linea retta che unisce i centri della casella corrente e di quella di destinazione, e infine creano l'animazione degli sprite che fornisce l'illusione del movimento continuo a velocità rallentata.

3.6 Conclusioni.

Si è presentata una brevissima storia del problema detto giro del cavallo, illustrando telegraficamente le sofisticate tecniche computazionali e strutture dati richieste per enumerare tutti i possibili percorsi chiusi e aperti. Questo ci ha fornito anche l'occasione per introdurre l'argomento dei metodi e linguaggi formali di specifica e verifica. Si è infine presentato il semplice ma efficace listato proposto nell'unico testo esistente inerente l'intelligenza artificiale applicata al C64, grazie al quale abbiamo potuto almeno accennare a tematiche logico-matematiche di grande importanza, sperando di suscitare curiosità e interesse nei lettori al di là del semplice codice proposto.

Capitolo 4

COMAL: la strana storia di un linguaggio dimenticato.

4.1 Introduzione.

Ripercorriamo sinteticamente la storia e le caratteristiche di un linguaggio che ha avuto il suo momento di gloria negli anni Ottanta, in ambito didattico ma anche applicativo. *COMAL* (acronimo di *COMmon Algorithmic Language*) nasce in Danimarca per una precisa esigenza didattica: l'impiego in corsi di alfabetizzazione informatica agli inizi degli anni Settanta, per i quali il *BASIC* era considerato inadeguato a causa della sua mancanza di struttura, ma al contempo il Pascal risultava eccessivamente avanzato e complesso. Durante gli anni Ottanta *COMAL* è stato implementato su due dozzine di home computer: particolare fortuna hanno avuto la versione interpretata 0.14 *Public Domain* su disco per Commodore 64, seguita dalla versione 2.0 su cartridge (molto più perfezionata, con una notevole quantità di memoria a disposizione dei programmi utente), e il compilatore per Atari con il quale sono stati prodotti anche numerosi applicativi commerciali, sia per usi domestici che burotici in area *SOHO*, distribuiti in tutta Europa.

4.2 La genesi di COMAL.

In una lunga e semiseria intervista rilasciata a «COMAL Today» e apparsa sul numero 25 (1989), il matematico danese Børge R. Christensen illustra la genesi del linguaggio e la sua a tratti rocambolesca implementazione su un Data General NOVA 1200 dotato unicamente di una unità a nastro carta perforato per la memorizzazione. Vogliamo qui riassumerne almeno i passaggi più salienti, pur rifuggendo dall'idea di proporre una pedissequa traduzione integrale che in quest'ambito divulgativo e colloquiale finirebbe per divenire uno sterile esercizio di acribia filologica, annoiando i lettori.

Figura 4.2.1: Data General NOVA 1200

Nel 1972 il college danese di Tønder (vicino al confine con la Germania) presso il quale il professor Christensen insegna matematica decide di dotarsi di uno dei primi calcolatori commerciali per applicazioni di ricerca e per la didattica. I primissimi corsi di informatica non sembrano tuttavia avere un particolare successo, e questo viene attribuito in particolare alla versione notevolmente fallata di *Extended BASIC* preinstallata sul calcolatore. Christensen lascia in realtà trasparire che anche la sua limitata esperienza in materia ha il suo peso, rendendogli difficile la lettura dei programmi *BASIC*: per superare questi dubbi, si reca presso la vicina università di Årus, nota per avere un dipartimento di informatica piuttosto efficiente. Sfruttando la contingenza di dover preparare i problemi d'esame per il suo primo (piuttosto disastroso) corso di informatica, chiede una consulenza a Benedict Løfstedt che lavora appunto in tale università. La risposta di Løfstedt è perfettamente in linea col pensiero informatico dell'epoca: la colpa non è del docente né degli studenti, bensì del linguaggio. Si ricordi che in quell'epoca, tra gli anni Sessanta e Settanta, siamo ancora nel pieno di un dibattito molto acceso nella comunità informatica sulla contrapposizione tra programmazione strutturata e non, tra linguaggi come *ALGOL* da un lato e *BASIC* dall'altro, con numerosi punti di contrasto e posizioni fortemente contrapposte, ad esempio, in merito all'uso della keyword `GOTO`: dibattito dal quale, oltre a fondamentali articoli come quelli

di Edsger W. Dijkstra[1] [Dij87, Dij82] e accorati interventi di altri padri nobili dell'informatica come C. A. R. (Tony) Hoare (1934-), Niklaus Wirth (1934-) e Donald E. Knuth (1938-), scaturisce tra l'altro nel 1966 anche un fondamentale teorema di estensione della tesi di Church-Turing sulla computabilità, dovuto agli italiani Corrado Böhm (1923-2017) e Giuseppe Jacopini (1936-2001) [BJ66], il quale sancisce che qualsiasi algoritmo computabile secondo Church-Turing può essere implementato in un linguaggio minimale dotato unicamente di costrutti sequenziali, selezioni e iterazioni (cicli), il che quindi esclude la stretta necessarietà di uno statement come `GOTO`.

In breve, Christensen viene prontamente indirizzato ad un libro seminale di Niklaus Wirth [Wir73] che è una pietra miliare nella storia dell'informatica applicativa e ne riceve una vera e propria illuminazione. Tuttavia, pur affascinato dal Pascal e dalla programmazione imperativa strutturata, si rende presto conto che la difficoltà di utilizzo per i suoi allievi, nella fascia d'età del college, sarebbe comunque eccessiva. Decide quindi di mantenere un ambiente interattivo e un linguaggio interpretato, ma durante tutto il 1973 in una lunga serie di scambi epistolari col collega Løfstedt costruisce la definizione di un linguaggio sostanzialmente nuovo, ibridando la struttura del Pascal con l'intuitività del *BASIC* e gettando le basi per quello che diventerà lo standard *COMAL 75*.

Se la genesi della definizione risulta relativamente semplice, la prima implementazione del linguaggio si rivela ben presto assai più ardua. Dopo avere invano tentato di rivolgersi ad alcune aziende locali, Christensen incarica due dei suoi migliori studenti di aiutarlo a codificare in Assembly sul NOVA 1200 l'intera implementazione dell'ambiente interattivo e dell'interprete, ed è qui che il suo racconto assume un tono vagamente epico, sebbene sempre intriso di autoironia. La pressoché totale mancanza di esperienza dell'improvvisato team, nonostante la consulenza intermittente di Løfstedt, e la fissazione su alcune caratteristiche didatticamente utili ma onerose dal punto di vista computazionale, come i nomi di variabile estesi a ben (sic!) 8 caratteri rispetto ai due consentiti dal *BASIC*, provocano un accumulo di ritardi. Un primo prototipo risulta effettivamente operativo il 5 agosto 1974, ma una versione realmente utilizzabile viene completata solo nel febbraio 1975. Christensen indulge anche nel colorire il racconto sottolineando come uno dei due studenti prescelti, incaricato in particolare del debugging, avesse all'epoca il poco edificante vizio dell'alcool, risultando spesso fuori combattimento per lunghi periodi a causa di sbornie colossali e ritardando così ulteriormente lo sviluppo. Altro aspetto pionieristico dell'intera vicenda è sicuramente l'imbarazzante limitatezza dei mezzi a disposizione, decisamente primitivi anche per l'epoca: soprattutto il terrificante nastro carta perforato, in mancanza di qualsivoglia mezzo di memorizzazione magnetico nell'installazione Data General in uso, ma anche il costo complessivo dell'intera operazione, che si attesta su un budget di soli 300 dollari.

Figura 4.2.2: Esemplari di nastro carta perforato, larghezza 5 (giallo) e 8 (rosa) fori.

Tuttavia, proprio l'universale diffusione nelle scuole danesi di ogni ordine e grado di macchine NOVA identiche o simili a quella utilizzata per lo sviluppo risulta essere il fattore strategico che decreta una immediata diffusione di *COMAL* nel mondo della didattica: in meno di un anno la maggioranza degli istituti sul territorio nazionale ha abbandonato il *BASIC* dell'installazione di default in favore del nuovo linguaggio, anche se in molti casi il caricamento del nastro carta originale con una telescrivente con lettore a 10 caratteri/secondo richiede poco più di un'ora di tempo. Christensen si compiace nell'attribuire tale successo principalmente alla presenza di nomi di variabile di lunghezza relativamente significativa, come risulta dalle interviste agli studenti, e ovviamente alla possibilità di usare costrutti strutturati. Come terza motivazione viene citata la possibilità di attribuire un nome alle subroutines, in realtà implementata fin dai primissimi linguaggi di alto livello (*COBOL* in primis) ma all'epoca inesistente nei *BASIC* tradizionali. Tutto ciò sottolinea come *COMAL 75* nasce dal traumatico incontro con le limitazioni di un *BASIC* decisamente primitivo e quindi ne costituisce un tentativo di superamento, anche in termini prestazionali, pur rimanendo nell'ambito di un linguaggio interpretato. Accorgimenti come l'uso estensivo di jump tables dinamiche, puntatori e link diretti alle procedure nascono esplicitamente per superare meccanismi realmente arretrati e farraginosi come la ricerca sequenziale effettuata dall'interprete nella vera e propria lista linkata semplice costituita dall'insieme delle linee *BASIC* in caso di `GOSUB`, che nel decennio successivo saranno progressivamente abbandonati anche nella maggioranza degli home *BASIC* interpretati.

Si noti che, a questo punto della sua storia, il linguaggio è ancora caratterizzato da un impronunciabile nome danese e probabilmente, sotto tale egida e legato a doppio filo com'era con una serie di macchine non

[1] (1930-2002) Fisico teorico per formazione, è stato uno dei più influenti informatici teorici dell'ultimo mezzo secolo. Pioniere e anticipatore dell'uso dei metodi formali di specifica e verifica, è ben noto in ambito informatico generalista anche come autore di alcuni dei più taglienti aforismi in merito a taluni linguaggi di programmazione in voga all'epoca del dibattito di cui discutiamo, ancor oggi citati sovente - sebbene quasi sempre a sproposito, fuori dal loro precipuo contesto storico.

particolarmente diffuse su scala globale e con un mezzo di distribuzione poco pratico come il nastro perforato, non avrebbe mai superato i confini del regno di Danimarca. In realtà è lo stesso Christensen, secondo la sua narrazione, a ideare il nome *COMmon Algorithmic Language* per banale analogia con l'*ALGOL* (*ALGOrithmtic Language*), all'epoca uno dei protagonisti di maggior rilievo nel campo della programmazione strutturata: progettato e standardizzato accuratamente, tanto da essere usato (nella versione *ALGOL 60* in particolare) come linguaggio ufficiale per l'espressione di algoritmi negli articoli scientifici per oltre tre decenni. Anche la metasintassi conosciuta come Forma di Backus–Naur (BNF), ben nota a qualunque informatico che abbia compiuto un minimo di studi istituzionali, è stata in origine concepita specificamente per descrivere la sintassi dei programmi in *ALGOL*: al pioniere John Backus (1924-2007) si deve l'ideazione di tale formalismo per *ALGOL 58*, perfezionato poi da Peter Naur (1928-2016) con *ALGOL 60*, come magistralmente illustrato dall'ineffabile Donald Knuth [Knu64]. Nella pratica l'unica aggiunta alla BNF (standardizzata dalla ISO/IEC 14977:1996, rev. 2018) che non sia riconducibile direttamente ad *ALGOL* è semplicemente l'uso delle parentesti quadre [ ] per delimitare i parametri opzionali: introdotto in realtà pochi anni dopo *ALGOL 60* con la definizione del linguaggio *PL/I* di IBM e poi diffusosi universalmente anche nelle sinossi informali di comandi e funzioni d'ogni genere, dai comuni linguaggi di programmazione ai batch, alle shell Unix, fino all'help online di singoli programmi applicativi, eccetera.

Piccolo inciso: un aspetto decisamente saliente della BNF e delle sue estensioni (come EBNF o le specializzazioni come TBNF, ABNF o RBNF) è l'espressività. Essa risulta sufficiente a descrivere anche la meta-metasintassi stessa di BNF in modo molto conciso, autoreferenziale e ricorsivo, come riportiamo di seguito da un classico esempio presente nella maggioranza dei testi, abbreviato per mere ragioni tipografiche. Risulta poi intuitivo come ad esempio in EBNF, grazie all'uso delle espressioni regolari, l'autodefinizione risulti ancora più concisa a livello di <letter>, <digit>, <symbol>.

```
<syntax>          ::= <rule> | <rule>
<syntax> <rule>   ::= <opt-whitespace> "<" <rule-name> ">"
<opt-whitespace> "::=" <opt-whitespace> <expression> <line-end>
<opt-whitespace>  ::= " " <opt-whitespace> | ""
<expression>      ::= <list> | <list> <opt-whitespace> "|" <opt-whitespace> <expr>
<line-end>        ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list>            ::= <term> | <term> <opt-whitespace> <list>
<term>            ::= <literal> | "<" <rule-name> ">"
<literal>         ::= '"' <text1> '"' | "'" <text2> "'"
<text1>           ::= "" | <character1> <text1>
<text2>           ::= '' | <character2> <text2>
<character>       ::= <letter> | <digit> | <symbol>
<letter>          ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | ...
<digit>           ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<symbol>          ::=  "|" | " " | "!" | "#" | "$" | "%" | "&" | "(" | ")" | ...
<character1>      ::= <character> | "'"
<character2>      ::= <character> | '"'
<rule-name>       ::= <letter> | <rule-name> <rule-char>
<rule-char>       ::= <letter> | <digit> | "-"
```

Una volta ideato un nome convincente per il nuovo linguaggio, dopo la presentazione e la rapida diffusione di *COMAL 75* negli istituti scolastici danesi, il professor Christensen si dedica alla stesura della documentazione del progetto, creando così le basi per quello che sarà il primo libro di testo del linguaggio [Chr82], originariamente pubblicato in danese qualche anno dopo la prima versione del linguaggio, nel 1978. Nel frattempo però i minicomputer NOVA si avviano rapidamente verso l'obsolescenza, a causa dell'invasione del mercato da parte dei primi PET: il mero timore di poter essere dantescamente rispedito ad insegnare qualche *BASIC*, così ci racconta coloritamente il simpatico matematico danese, è sufficiente a spingere il suo gruppo di lavoro a perfezionare una ulteriore versione del linguaggio, più completa ed efficiente, destinata alle piattaforme emergenti. Così il gruppo di lavoro si amplia, unendo i suoi sforzi ad un team in un altro college e nel giro di un paio d'anni si arriva alla versione *COMAL 80*, così denominata sia in base al fattore cronologico (1980 è l'anno di presentazione), sia perché sviluppata prevalentemente su una macchina basata sullo Zilog Z80. Nel maggio 1982 viene ratificato lo standard internazionale denominato *COMAL KERNAL*, poi sottoposto a riapprovazione 1983. In quegli stessi anni, a maggior gloria del nazionalismo danese, una società locale ha iniziato a progettare e realizzare computer specificamente dedicati alla didattica, grazie soprattutto all'uso di un moderno bus che rende facile l'espansione della macchina e anche la sperimentazione elettronica negli istituti tecnici: ovviamente tale macchina è dotata di *COMAL*. Al linguaggio però in questo momento mancano ancora molte delle caratteristiche che lo hanno reso famoso nei due decenni successivi: grafica e sprites, suono, un editor avanzato, un ambiente completo che

consenta di gestire direttamente nastri, dischi e altre memorie di massa. Tuttavia, si è già ricavato una solida posizione di predominio nell'ambito didattico, tanto che a partire dai primi anni Ottanta nessun istituto danese accetta forniture di calcolatori che non abbiano *COMAL* a bordo.

Nel 1982 il danese Mogens Kjær, dietro consiglio di Christensen e sulla base dello standard *COMAL KERNAL*, inizia a sviluppare quella che sarà la famosa e diffusa serie di versioni 0.1x *Public Domain* per Commodore PET e successivamente per C64, avviando assieme ad un gruppo di pionieri ed entusiasti (Jens Erik Jensen, Helge Lassen e Lars Laursen) una società denominata *UniComal ApS*, che di fatto succede al team originale nello sviluppo del linguaggio e nel porting verso altre piattaforme. L'ambiente di lavoro viene espanso e perfezionato per sostituire completamente quello del *BASIC V2* e il relativo *DOS Wedge*, dando la possibilità di gestire con semplici e intuitivi comandi i file su nastro e disco, le stampanti e altre periferiche. Vengono aggiunti i moduli binari, denominati «packages», che sono librerie di routine in codice macchina richiamabili direttamente da *COMAL*.

Pochi anni dopo nasce anche una versione sviluppata appositamente per gli Amstrad, successivamente anche un compilatore per Atari e una lunga serie di porting per altri home computer, inclusa una versione avanzata di *COMAL 80* su cartridge (la 2.0) per C64 commercializzata nel 1985 che resterà in auge a lungo, praticamente per l'intera storia di tale home computer. Viene presto sviluppata anche una versione con compilatore per i potentissimi VAX-11, evento più unico che raro nella storia dei linguaggi didattici nati su home computer. Negli anni Ottanta e Novanta il linguaggio, nelle sue varie versioni, viene utilizzato per la didattica in numerosi paesi anglofoni del Nord Europa, ma anche per svariate applicazioni commerciali, in particolar modo su C64 con apposita cartridge e soprattutto su Atari e VAX nella versione compilata.

Chiudiamo questa succinta cronistoria con una doverosa citazione di Børge R. Christensen: «*COMAL* is first of all for people who are not professional... to make it possible for people to program computers, even if they were not programming people.». Come talora avviene, si tratta però di un linguaggio talmente ben concepito da attrarre immediatamente e per lungo tempo a venire l'attenzione di molti professionisti, che data l'epoca pionieristica della sua diffusione non ha rischiato di diventare un Santo Graal di pasticcioni e improvvisatori, come è invece è tristemente accaduto alla svolta del millennio con linguaggi concepiti nominalmente con le medesime intenzioni. Inoltre è indiscutibile merito di questo linguaggio l'aver abbreviato il percorso verso il conseguimento di solide competenze di *problem solving* e logica della programmazione per intere generazioni di studenti in gran parte del sistema scolastico europeo anglofono.

4.3 Un primo sguardo a *COMAL 80* per C64.

La fortunata serie di release per Commodore avviata dal *COMAL 0.11* rilasciato da *UniComal ApS* nel 1982 è culminata con una versione di larghissima diffusione, la 0.14, alla quale si riferiscono anche numerosi manuali. Tale versione *Public Domain*, caricata da floppy o da nastro, non lasciava tuttavia molto spazio disponibile per i programmi utente (circa 9.900 bytes sul C64): in ultima analisi, ciò non costituiva in realtà un problema per il principale utilizzo del linguaggio, quello didattico e formativo. Dopo una versione intermedia 1.2 destinata principalmente al Commodore PET 8096, compare la versione 2.0, meglio nota come *COMAL 80 per C64* prodotta nel 1984 e distribuita su cartuccia CBM nei primi mesi del 1985. Tale versione, disponibile anche su schede con ROM associate ad espansioni di memoria per i PET, garantiva sul C64 30.714 bytes di spazio per i programmi utente, più del triplo delle versioni precedenti. Essa risultava notevolmente più potente e completa delle prime release, ed è stata utilizzata con successo nell'arco di due decenni non solo per la didattica, ma anche per numerosi programmi commerciali.

L'ambiente interattivo sostituiva completamente l'originale del C64, rendendo disponibili potenti comandi per la gestione di dischi e nastri, la stampa, le periferiche su porta utente e seriali, librerie grafiche, perfino un sottosistema *LOGO Turtle*, editor di programma fullscreen e molto altro sotto forma di moduli binari abilitabili a piacimento. Risulta inoltre molto semplice espandere il linguaggio con nuove *keyword* create dall'utente e con moduli binari scritti in Assembly.

In questa sede preferiamo però evitare di iniziare con una sterile, pedissequa elencazione delle caratteristiche del linguaggio e delle sue peculiarità sintattiche (tutti aspetti peraltro ampiamente sviscerati nella manualistica indicata in bibliografia), lasciando invece «parlare» degli esempi concreti di codifica come primo assaggio della potenza e della reale modernità di *COMAL 80*. Senza voler appesantire la trattazione con puntuali citazioni dai testi di riferimento fondamentali in materia come [Seb15, FW08], si preferisce che il lettore valuti intuitivamente, *hands-on*, la **leggibilità** e la **manutenibilità** del linguaggio come caratteristiche oggettive di software engineering e language design. Per cominciare proponiamo quindi un brevissimo programma che in sole 23 LOC genera esaustivamente tutte le permutazioni dei simboli *PETSCII* immessi in input in una stringa arbitraria (lunghezza massima 20 caratteri, comunque sconsigliata anche ai più temerari, sia pure su emulatori con *warp mode*). Il programma è tratto direttamente dal disco di esempi che accompagnava la cartridge *COMAL 80*.

```
0010 // SAVE "permute"
```

```
0020 //
0030 // (c) 1984 by UniComal ApS.
0040 //
0050 PAGE
0060 DIM a$ OF 20
0070
0080 PROC permute(a$,k)
0090   IF k<=1 THEN
0100     PRINT a$
0110   ELSE
0120     permute(a$,k-1)
0130     FOR i:=1 TO k-1 DO
0140       permute(a$(:i-1)+a$(k)+a$(i+1:k-1)+a$(i)+a$(k+1:),k-1)
0150     ENDFOR i
0160   ENDIF
0170 ENDPROC permute
0180
0190 INPUT "Permute: ": a$
0200 PRINT
0210 permute(a$,LEN(a$))
0220
0230 END "End"
```

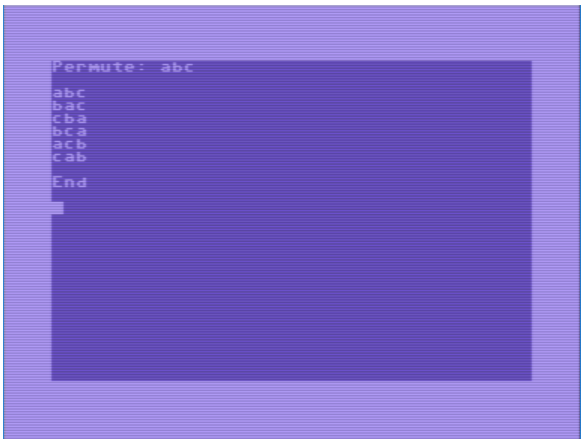


Figura 4.3.1: Permutazioni di tre elementi in COMAL.

Balzano immediatamente agli occhi la sintesi, l'intuitività e l'assoluta chiarezza della struttura logica del programma, tipiche della programmazione strutturata quanto antitetiche ai primitivi *BASIC* dei PET. I numeri di linea vengono generati automaticamente dall'editor e sono allineati per default a 4 cifre, aumentando notevolmente la leggibilità. Il programma non fa uso di nomi lunghi di variabile, in quest'occasione. Si nota immediatamente l'uso di un classico algoritmo ricorsivo, sul quale non ci soffermeremo se non per sottolineare come il parametro stringa passato di volta in volta alla funzione sia in realtà l'unione di sottostringhe ricavate dinamicamente con «modernissime» espressioni di *slicing*, che molti studenti e practitioners credono nate con Python negli anni Novanta o con linguaggi anche più recenti, ma in realtà supportate da oltre mezzo secolo dai benemeriti *FORTRAN*, *APL*, *ALGOL*, Ada e perfino da qualche home *BASIC* dei primi anni Ottanta, come ad esempio il *Sinclair BASIC*. Si notino inoltre i commenti che qualcuno potrebbe essere tentato di definire «in stile C++», in realtà altra «modernità» degli anni Settanta: alla quale ovviamente Bjarne Stroustrup nei primi anni Ottanta può essersi liberamente ispirato, a margine del notevole sforzo di concepire il suo *C with classes* come evoluzione e superamento delle possibilità di *SIMULA*.

Come secondo assaggio, nulla di meglio che un classico quicksort come proposto dal disco *COMAL 80*. Algoritmo oggettivamente non facile da implementare in *BASIC V2* in meno di 60 LOC, mantenendo l'assoluta chiarezza del seguente listato. Chiunque abbia programmato intensivamente in *BASIC V2* saprà sicuramente apprezzare, tra l'altro, la robustezza e l'intuitività dello statement `WHILE NOT EOD DO` che cancella con un colpo di spugna l'incubo dell'*off-by-one* e più in generale il temibile errore `OUT OF DATA`.

Si noti anche l'eleganza degli statement di assegnazione composita, che eliminano le inguardabili istruzioni di autoassegnazione come `A = A + 1`, probabilmente uno degli aspetti soggettivamente più odiati dai programmatori HLL alle prese con il *BASIC V2*.

```
0010 // save "quicksort"
0020 //
0030 // by UniComal ApS. may 1984
0040 //
0050
0060 PROC quicksort(REF a$(),left,right,reclen) CLOSED
```

```
0070    DIM pivot$ OF reclen, buffer$ OF reclen
0080    partition(left,right,left,right) // sort a$(left:right)
0090
0100    PROC partition(left,right,i,j)
0110      pivot$:=a$((left+right) DIV 2) // get middle element as pivot
0120      REPEAT // perform swappings
0130        WHILE pivot$>a$(i) DO i:+1
0140        WHILE pivot$<a$(j) DO j:-1
0150        IF i<=j THEN swap(a$(i),a$(j)); i:+1; j:-1
0160      UNTIL i>j
0170      IF left<j THEN partition(left,j,left,j) // sort a$(left:j)
0180      IF i<right THEN partition(i,right,i,right) // sort a$(i:right)
0190    ENDPROC partition
0200
0210    PROC swap(REF a$,REF b$)
0220      buffer$:=a$; a$:=b$; b$:=buffer$
0230    ENDPROC swap
0240 ENDPROC quicksort
0250
0260 PAGE
0270 DIM message$(1:50) OF 35
0280 messageno:=0
0290
0300 WHILE NOT EOD DO
0310    messageno:+1
0320    READ message$(messageno)
0330 ENDWHILE
0340
0350 // sort error messages:
0360
0370 PRINT "sorting started:"
0380 quicksort(message$(),1,messageno,35)
0390
0400 // print the sorted messages:
0410
0420 FOR i:=1 TO messageno DO PRINT i,":";message$(i)
0430
0440 END "End of sort"
0450
0460 // start-of-data
0470
0480 DATA "format error"
0490 DATA "syntax error"
0500 DATA "type conflict"
0510 DATA "function argument error"
0520 DATA "statement too long or too complicated"
0530 DATA "system error"
0540 DATA "name too long"
0550 DATA "bracket error"
0560 DATA "overflow"
0570
0580 // end-of-data
```

Un terzo esempio, sempre tratto dal disco originale, genera esaustivamente tutte le soluzioni al classico problema scacchistico detto «delle otto regine»: data una scacchiera standard bicolore da 64 caselle, si devono posizionare 8 regine in modo tale che nessuna di esse minacci le altre, ossia ponendo al più un solo pezzo per ogni riga, colonna e diagonale. Sebbene non si disponga di una formula chiusa per calcolare il numero delle soluzioni a partire dalla dimensione n della scacchiera data, è noto che con 8 regine esistono 92 soluzioni distinte in totale, derivate per rotazioni e riflessioni da 12 soluzioni di base, dette «uniche».

Il breve programma qui proposto genera sequenzialmente, con accattivanti animazioni grafiche, tutte le 92 possibili configurazioni che risolvono il problema. Si noti che i dati degli sprites vengono elegantemente letti da un file dati sequenziale e non sono incorporati nel codice.

Si invita il lettore ad apprezzare la chiarezza del programma, grazie alla forte strutturazione, all'uso estensivo di nomi lunghi per le varabili e per le procedure, oltre alla presenza dei commenti: ricordando inoltre che tutto ciò avveniva nel lontano 1985 (ed era comunque possibile anche prima su Commodore PET e C64, con *COMAL 0.1x*), quando gli home *BASIC* su hardware di pari classe offrivano livelli di leggibilità, sintesi e mantenibilità nettamente inferiori e i compilatori per HLL come Pascal erano una rarità. Il programma, nello specifico, fa uso di backtracking: una soluzione decisamente classica sebbene dalle prestazioni non entusiasmanti.

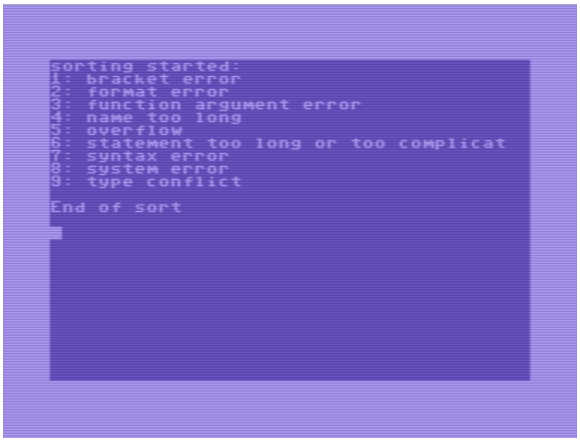


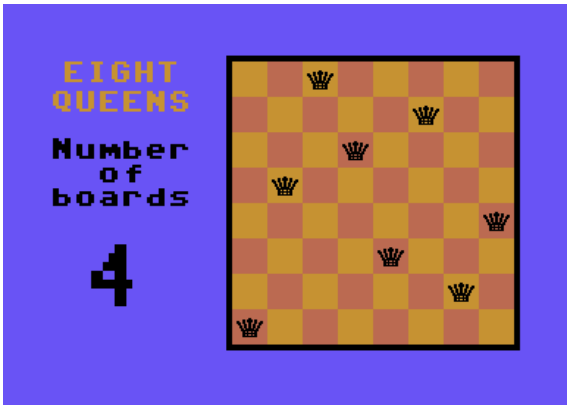
Figura 4.3.2: Quicksort in COMAL.

```
0010 // SAVE "queens"
0020 //
0030 // (c) 1984 by UniComal ApS.
0040 //
0050 // Find all possible placements
0060 // of 8 queens on a chess board
0070 // in such a fashion that none
0080 // is checking any other piece,
0090 // i.e. each row, column and
0100 // diagonal must contain at most
0110 // one piece.
0120
0130 PAGE
0140 m:=24 // size of squares //
0150 l:=8*m // size of board
0160 x0:=(320-l)-4
0170 y0:=(200-l) DIV 2
0180 homex:=x0-27
0190
0200 red:=2; blue:=6
0210 black:=0; orange:=8
0220
0230 USE graphics
0240 graphicscreen(1)
0250 // blue background //
0260 background(blue)
0270 border(-1)
0280 clearscreen
0290
0300 USE sprites
0310 queen:=0
0320 DIM xpos(0:7)
0330 // define queen shape //
0340 OPEN FILE 1,"queen.spr",READ
0350 define(queen,GET$(1,64))
0360 CLOSE
0370
0380 DIM row(1:8)
0390 // row(i) = "no queen on i-th row" //
0400 DIM d1(1:2*8)
0410 // d1(i) = "no queen on i-th upleft to lowright diagonal //
0420 DIM d2(1:2*8-1)
```

```
0430 // d2(i) = "no queen on i-th lowleft to upright diagonal" //
0440
0450 //-------main--------//
0460
0470 clear.and.draw.board
0480 trycol(1)
0490 FOR i:=0 TO 7 DO
0500    slideto(i,homex)
0510 ENDFOR i
0520 END "End Queens"
0530
0540 //--------procedures---------//
0550
0560 PROC paint(c,x,y,w,h)
0570    // paint rectangle //
0580    viewport(x,x+w-1,y,y+h-1)
0590    pencolor(c)
0600    fill(x,y)
0610 ENDPROC paint
0620
0630 PROC clear.and.draw.board
0640    FOR i:=1 TO 8 DO row(i):=TRUE
0650    FOR i:=1 TO 2*8 DO d1(i):=TRUE
0660    FOR i:=1 TO 2*8-1 DO d2(i):=TRUE
0670    // draw text //
0680    pencolor(orange)
0690    type(0,180,1,2,"EIGHT")
0700    type(0,160,1,2,"QUEENS")
0710    pencolor(black)
0720    type(0,128,1,2,"Number")
0730    type(0,112,1,2,"  of")
0740    type(0,96,1,2,"boards")
0750    boards:=-1
0760    newboard
0770    // draw border //
0780    paint(black,x0-4,y0-4,8+l,4)
0790    paint(black,x0-4,y0,4,l)
0800    paint(black,x0-4,y0+l,8+l,4)
0810    paint(black,x0+l,y0,4,l)
0820    // define colors of the queens //
0830    y:=y0+22
0840    FOR sprite:=0 TO 7 DO
0850       spritecolor(sprite,black)
0860       xpos(sprite):=homex
0870       spritepos(sprite,homex,y)
0880       identify(sprite,queen)
0890       showsprite(sprite)
0900       spritesize(sprite,0,0)
0910       y:+m
0920    ENDFOR sprite
0930    // draw board //
0940    y:=y0
0950    color:=red
0960    FOR row:=1 TO 8 DO
0970       x:=x0
0980       FOR col:=1 TO 8 DO
0990          paint(color,x,y,m,m)
1000          x:+m
1010          color:=red+orange-color
1020       ENDFOR col
1030       y:+m
```

```
1040      color:=red+orange-color
1050    ENDFOR row
1060    last:=-1
1070    viewport(0,319,0,199)
1080  ENDPROC clear.and.draw.board
1090
1100  PROC pause
1110    FOR i:=1 TO 7000 DO NULL
1120  ENDPROC pause
1130
1140  PROC slideto(sprite,x1)
1150    y1:=y0+sprite*m+22
1160    stp:=SGN(x1-xpos(sprite))
1170    IF stp<>0 THEN
1180      FOR x:=xpos(sprite) TO x1 STEP stp DO
1190        spritepos(sprite,x,y1)
1200      ENDFOR x
1210      xpos(sprite):=x1
1220    ENDIF
1230  ENDPROC slideto
1240
1250  PROC place.queen(i,j)
1260    row(i):=FALSE; d1(i+j):=FALSE; d2(8+i-j):=FALSE
1270    IF last>0 AND last<>j THEN
1280      slideto(last-1,homex)
1290    ENDIF
1300    slideto(j-1,x0+(i-1)*m)
1310    last:=-1
1320  ENDPROC place.queen
1330
1340  PROC remove.queen(i,j)
1350    row(i):=TRUE; d1(i+j):=TRUE; d2(8+i-j):=TRUE
1360    IF last>0 THEN
1370      slideto(last-1,homex)
1380    ENDIF
1390    last:=j
1400  ENDPROC remove.queen
1410
1420  PROC trycol(j)
1430    FOR i:=1 TO 8 DO
1440      IF row(i) AND d1(i+j) AND d2(8+i-j) THEN
1450        place.queen(i,j)
1460        IF j<8 THEN
1470          trycol(j+1)
1480        ELSE // all queens placed //
1490          newboard
1500          pause
1510        ENDIF
1520        remove.queen(i,j)
1530      ENDIF
1540    ENDFOR i
1550  ENDPROC trycol
1560
1570  PROC newboard
1580    // draw new number //
1590    pencolor(black)
1600    boards:+1
1610    type(0,24,3,6,STR$(boards))
1620  ENDPROC newboard
1630
1640  PROC type(x0,y0,xsize,ysize,text$)
```

```
1650    IF LEN(text$) MOD 2=1 THEN
1660      x0:+xsize*8
1670    ENDIF
1680    textstyle(xsize,ysize,0,0)
1690    plottext(x0,y0,text$)
1700 ENDPROC type
```



4.4 Bibliografia essenziale.

I testi originali di Børge R. Christensen, creatore del linguaggio con Benedict Løfstedt, sono [Chr82, Chr84]. Altri testi generici sul linguaggio includono [Ath82, BP88, Gra85, Kel84, LO90].

COMAL 80 per C64 è interamente documentato nel manuale originale Commodore «COMAL 80» (a cura di Frank Bason e Leo Højsholt-Poulsen, 1985) pubblicato in duplice edizione inglese e danese, relativo alla versione 2.0 su cartuccia con disco allegato. Specifici per C64 anche i testi di Lindsay [Lin83] e J. William Leary [Lea86].

4.5 Conclusioni.

Si è presentata una brevissima cronistoria della definizione e creazione del linguaggio *COMAL* (in particolare le versioni *milestone* 75 e 80 e lo standard 1982/83 noto come *COMAL KERNAL*) da parte del matematico danese Børge R. Christensen e del suo collega informatico Benedict Løfstedt, seguita dalle principali tappe dello sviluppo commerciale europeo del linguaggio in numerose versioni, concentrandoci in particolare sulle principali release per PET e Commodore 64. Si sono proposti alcuni semplicissimi sorgenti di esempio al fine di incuriosire il lettore, mostrando solo alcuni aspetti della potenza e della leggibilità di un linguaggio di notevole innovatività nell'ambito degli home computer: con la speranza di avere suscitato anche qualche gradito ricordo in chi all'epoca avesse provato ad utilizzare il linguaggio su C64 come alternativa al *BASIC V2* o su altre piattaforme. Si è infine esposta una bibliografia di riferimento sul linguaggio, comprendente pressoché tutti i testi in lingua inglese ritenuti più autorevoli e di maggiore diffusione.

Qualora dopo la pubblicazione del presente articolo si riscontrasse sufficiente interesse attorno al linguaggio, ciò che l'autore auspica, si potrà dare continuità alla trattazione con la presentazione di ulteriori esempi e quesiti risolti con l'uso di *COMAL 80* per C64, anche in parallelo con le relative soluzioni in Assembly 6510 e *BASIC V2*.

Capitolo 5

«1: I numeri dei salmi».

5.1 Introduzione.

Il Commodore 64 è indiscutibilmente l'home computer più venduto della storia, con oltre 17 milioni di unità prodotte. Il relativo mercato bibliografico è conseguentemente stato uno dei più sviluppati in assoluto, con centinaia di titoli stampati da decine di editori. Tra questi spiccavano testi di programmazione in BASIC, Assembly e altri linguaggi, testi sull'hardware e la riparazione, testi di grafica e game programming, testi applicativi in numerose aree.

Un posto speciale era però occupato dalle raccolte di puzzle e rompicapo: tra questi un best seller dell'epoca era "The Commodore Puzzle Book - BASIC Brainteasers" di Lee e Scrimshaw [LS83], che proponeva divertenti puzzle e rompicapo completi di soluzioni in linguaggio BASIC V2. Le soluzioni avevano sempre ampi margini di miglioramento: ci piace pensare che gli autori avessero volutamente lasciato spazio per ulteriori ottimizzazioni e per stimolare soluzioni alternative.

Con il presente articolo inauguriamo una serie che tratterà alcuni problemi tratti da tale testo, che molti di noi retroprogrammatori negli anni Ottanta si sono ampiamente divertiti a risolvere. Il linguaggio di questa serie di articoli è intenzionalmente informale, divulgativo e didascalico, scevro da particolari pretese di rigore, al fine di raggiungere il più vasto uditorio informatico possibile.

L'articolo è organizzato in tre parti principali. Nella prima parte si illustra semplicemente il problema nella sua forma originale, mentre nella seconda parte si presenta la soluzione proposta dagli autori del testo. Nella terza e ultima parte si discute una soluzione alternativa, che comporta un netto miglioramento prestazionale e fa uso di una delle strutture dati elementari maggiormente efficienti per la rappresentazione di sottoinsiemi, universalmente utilizzata nella combinatoria computazionale. Di tale soluzione vengono fornite implementazioni in BASIC V2 e in COMAL 2.01.

5.2 Hymn Numbers.

Il problema originale, a pagina 3 del testo [LS83], è di fatto il primo quesito proposto nel libro. Eccone di seguito il testo completo:

The other Sunday, the hymn numbers on the board appeared as shown. It caught my eye because I saw that all the digits were different. I then noticed that the second hymn number was twice the first, and the third was equal to the first two added together.

1	9	2
3	8	4
5	7	6

This made me wonder if there were any other sets of numbers, all different, that could be formed into 3 three digits numbers with this curious property.

Il testo è brevissimo e sufficientemente chiaro. Viene dato per scontato che si opera in base dieci e si richiede di individuare delle particolari terne di numeri naturali (interi positivi): per fissare le idee, siano essi a, b, c. Sappiamo che $b = 2 \cdot a$ e $c = a + b = 3 \cdot a$. Il testo precisa inoltre esplicitamente che tutti i numeri devono

essere di tre cifre: questo implica, in particolare, che $c < 10^3$ e di conseguenza $a < c/3$. Dal momento che non sono ammesse cifre ripetute, si deve in realtà considerare come limite superiore il valore naturale a tre cifre più prossimo a 999 composto con cifre tutte distinte, vale a dire 987, da cui $a \leq 329$. Con analogo ragionamento si stabilisce il limite inferiore, per cui in definitiva $102 \leq a \leq 329$. Scrivendo tali numeri tutti di seguito, inoltre, tutte le nove cifre devono essere diverse tra loro, ossia ciascuna cifra dell'intervallo naturale $[0, 9]$ deve comparire zero o esattamente una volta.

5.3 La soluzione originale.

La soluzione proposta a pag. 50 di [LS83] genera tutte le terne di numeri di tre cifre che godono delle proprietà desiderate.

```
1 rem soluzione originale dal puzzle book
2 rem problema 1: hymn numbers
3 s = ti: print "soluzione originale dal puzzle book"
10 for h = 1 to 3
20 for t = 0 to 9
30 if (h = t) then 260
40 for u = 0 to 9
50 if (u = t) or (u = h) then 250
60 n = 100*h + 10*t +u
65 if (n > 329) then print "run time: "; (ti-s)/60;"s": end
70 a = n*2
80 b = n*3
90 a$ = mid$(str$(a),2)
100 b$ = mid$(str$(b),2)
110 n$  = mid$(str$(n),2)
120 c$ = a$ + b$
130 for m = 1 to 5
140 for l = (m + 1) to 6
150 if (mid$(c$,m,1) = mid$(c$,l,1)) then goto 250
160 next l
170 next m
190 for m = 1 to 3
200 for l = 1 to 6
210 if (mid$(n$,m,1) = mid$(c$,l,1)) then goto 250
220 next l
230 next m
240 print n$; ", "; a$; ", "; b$
250 next u
260 next t
270 d(h+1) = 0: next h
```

Come è immediato rilevare dalla lettura del codice, il programma si può dividere concettualmente in due blocchi fondamentali: la generazione dei valori di base, e il calcolo dei valori derivati con la relativa validazione.

Il loop esterno genera esaustivamente tutti i valori a tre cifre nell'intervallo [100, 329] tramite un semplice generatore di tipo odometrico[1] che consta in realtà di tre loop annidati.

Per ogni valore di base, per costruzione già privo di cifre ripetute (ciò è garantito dai controlli effettuati rispettivamente alle linee 30 e 50), vengono poi calcolati il doppio e il triplo, ed infine si procede a verificare l'unicità di ciascuna cifra, secondo il paradigma combinatorio denominato *generate and test*. A questo provvedono le linee dalla 70 alla 230 incluse nel listato proposto.

[1] Tale elementare algoritmo combinatorio 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 azzerata 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.

Dal punto di vista combinatorio, i numeri di base da generare sono ovviamente caratterizzati per l'ordine delle cifre. L'ampiezza della presentazione (tre cifre) è strettamente minore del numero di simboli disponibili (dieci), e non sono ammesse ripetizioni come 11, 22, 33, 202, 244, etc. Pertanto, per definizione, si tratta di generare restrizioni di *disposizioni semplici* di dieci simboli su tre posizioni.

Ricordiamo qui per mera comodità la nota formuletta per le disposizioni semplici di n elementi su k posizioni:

$$D_{n,k} = \frac{n!}{(n-k)!}$$

Figura 5.3.1: La soluzione originale del testo.

In linea di principio, il programma eseguirà non meno di $329 - 100 + 1 = 230$ passi nel loop principale, saltando però l'elaborazione per quei rami improduttivi dell'albero computazionale che non corrispondono a disposizioni semplici (questa fondamentale tecnica di generazione combinatoria è detta *pruning* o anche *early filtering*). L'ampiezza dello spazio combinatorio in questione è ovviamente data da $2 \cdot D_{9,2} + 3 \cdot 8 = 168$ laddove il primo addendo tiene conto delle *disposizioni semplici* tra 102 e 298 costruite anteponendo rispettivamente 1 e 2 e utilizzando poi i rimanenti 9 simboli su due cifre, mentre il secondo addendo enumera le disposizioni semplici costruite col medesimo criterio, ma maggiori di 301 e inferiori a 329: infatti, dopo avere anteposto il 3 si hanno tre sole possibilità $\{0, 1, 2\}$ per la scelta della seconda cifra, a causa del vincolo, e per ciascuna scelta della seconda cifra rimangono altre otto possibilità per la cifra meno significativa.

La struttura fondamentale del programma risulta quindi molto razionale e in linea con i principi asseverati della generazione combinatoria, per quanto attiene la prima parte del problema.

Tuttavia, gli autori hanno scelto una soluzione implementativa per la convalida e il controllo di unicità delle cifre che ha un costo computazionale eccessivamente elevato. Osservando i loop annidati alle linee $130 \div 170$ e $190 \div 230$ risulta infatti immediatamente evidente la farraginosità ed inefficienza del metodo impiegato per il confronto diretto tra le cifre. Ciò si traduce, su Commodore 64, in un tempo di esecuzione dell'ordine di 45". In realtà, come vedremo a breve, è possibile (e informaticamente *doveroso*) fare di meglio con l'uso di una opportuna e assai più idonea struttura dati, che consente l'accesso in $O(1)$.

5.4 Soluzione alternativa in BASIC V2.

Si propone nel seguito una soluzione completa in BASIC V2 al quesito del testo, in grado di migliorare notevolmente la prestazione a runtime, pur mantenendo sostanzialmente inalterata la struttura di base del generatore combinatorio impiegato e rispettando l'impostazione fortemente didattica dell'intera soluzione.

Ragionando dal punto di vista meramente combinatorio e ponendo a margine le condizioni $b = 2 \cdot a$, $c = 3 \cdot a$ (che di fatto riducono notevolmente lo spazio combinatorio da esplorare, ma non cambiano la natura intrinseca del problema), il quesito si riduce essenzialmente alla generazione di disposizioni semplici di dieci elementi su nove posizioni. Poiché le ripetizioni sono proibite, si tratta quindi di selezionare un opportuno sottoinsieme proprio dall'insieme di partenza $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$: la struttura dati per eccellenza più adatta a rappresentare un sottoinsieme è l'array booleano, nel quale ad ogni posizione è associato biunivocamente un singolo elemento dell'insieme, secondo un ordine prefissato in modo arbitrario. Il valore booleano di ogni cella del vettore indica lapalissianamente se l'elemento appartiene o meno al sottoinsieme considerato[2]. Nel nostro caso infatti non

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

$$f_S(a) := \begin{cases} 1 & \text{se } a \in S \\ 0 & \text{se } a \notin S \end{cases} \tag{5.4.1}$$

In letteratura la funzione caratteristica è sovente indicata anche come 1_S, I_S o χ_S. Si noti che la cardinalità del sottoinsieme $\mathcal{S}$ è espressa dalla somma dei valori della sua funzione caratteristica:

$$|\mathcal{S}| = \sum_{a \in \mathcal{A}} f_S(a) \tag{5.4.2}$$

Si consideri ad esempio, per l'insieme delle dieci cifre naturali da noi utilizzato, il sottoinsieme $\{1, 3, 5\}$: la sua rappresentazione con un vettore binario, considerando l'ordine naturale, è $\langle 0, 1, 0, 1, 0, 1, 0, 0, 0, 0 \rangle$ dove appunto $V[1] = V[3] = V[5] = 1$.
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*, tipicamente codificato in modo

occorre alcun particolare sforzo immaginativo per scegliere un ordine da applicare all'insieme di partenza: ci accontenteremo del naturale ordine ascendente delle prime dieci cifre decimali per la implicita corrispondenza posizionale con le celle del vettore binario.

In generale la più naturale implementazione di questi array booleani è l'uso di bit array, un tipo di dato che molti linguaggi e piattaforme gestiscono efficientemente in modo nativo. Alcune CPU e molti microcontroller coevi al 6502/6510 gestiscono in hardware aree di memoria bit-addressable; in genere lavorando in Assembly si può ottenere la massima efficienza su quasi ogni CPU convenzionale usando istruzioni booleane atomiche per azzerare, impostare, invertire e verificare lo stato di un singolo bit in un registro. Alternativamente, usando linguaggi HLL, si possono utilizzare array di (piccoli) interi, raggiungendo un compromesso ingegneristico tra prestazioni e footprint di memoria generalmente accettabile, a maggior ragione in sede didattica ed illustrativa.

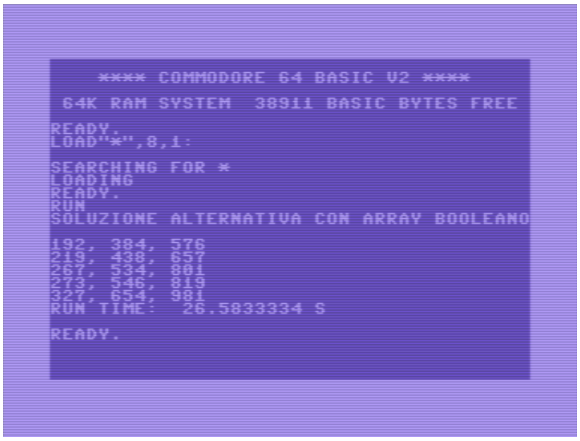


Figura 5.4.1: Soluzione alternativa in BASIC V2.

Con l'uso di tale struttura dati risulta in ogni caso immediato controllare se un dato elemento appartiene o meno al sottoinsieme considerato, e quindi nel caso specifico se una cifra è già stata eventualmente utilizzata: è sufficiente un accesso alla memoria, alla posizione $i-esima$ dell'array, dove i è la cifra attualmente in fase di controllo.

```
100 print "{clear} ************************************"
101 print " *** 1: hymn numbers                    ***"
102 print " ************************************"
120 s = ti : print " soluzione alternativa: array booleano" : ?
140 for h = 1 to 3
150 d%(h+1) = 1
160 for t = 0 to 9
170 if d%(t+1) = 1 then goto 350
180 d%(t+1) = 1 190 for u = 0 to 9
200 if d%(u+1) = 1 then goto 330
210 d%(u+1) = 1
220 n = 100*h + 10*t +u
230 if (n > 329) then print "run time: "; (ti-s)/60;"s": end
240 a$ = mid$(str$(n+n),2): b$ = mid$(str$(3*n),2): c$ = a$ + b$
250 for j = 1 to 6: i = 1 + val(mid$(c$,j,1))
260 if d%(i) = 1 then goto 300
270 d%(i) = 1
280 next j
290 print mid$(str$(n),2); ", "; a$; ", "; b$
300 if j > 1 then for k = 1 to j-1: d%(1 + val(mid$(c$,k,1))) = 0: next k
320 d%(u+1) = 0
330 next u
340 d%(t+1) = 0
350 next t
360 d%(h+1) = 0: next h
```

Come risulta evidente dallo screenshot in fig. 5.4.1, il guadagno prestazionale ottenuto col semplice impiego di una struttura dati più adatta, che consente l'accesso diretto all'informazione necessaria alla convalida di ogni singola cifra, è quantificabile nell'ordine del 40%: in pratica si consegue quasi un **raddoppio prestazionale**, pur lasciando inalterata la natura del loop principale a triplice annidamento.

---

compatto come *bitarray* sotto forma di (piccolo) numero intero senza segno. In ultima analisi, in combinatoria generativa e in numerosi altri scenari applicativi, i sottoinsiemi sono normalmente codificati tramite tale numero naturale, il cui valore binario corrisponde alla relativa funzione caratteristica, una volta fissato un ordine convenzionale per gli elementi.
Si richiama l'attenzione sulla universalità di questo metodo, che lo rende adatto alla rappresentazione di qualsivoglia insieme simbolico arbitrario finito, grazie al fondamentale lemma enumerativo che sancisce la possibilità di stabilire sempre una biezione arbitraria di qualsiasi insieme finito su un qualsiasi intervallo intero, e a fortiori naturale. Questo spiega la sua universale pervasività nella letteratura teorica e applicativa per un vastissimo numero di utilizzi in ambito discretistico e combinatorico.

Vale forse la pena di sottolineare a margine lo stupore e il dispiacere dell'autore nel constatare che, nonostante l'assoluta intuitività e banalità della soluzione proposta, la sua enorme pervasività nei testi di ogni livello inerenti l'algoritmica e la combinatorica (anche laddove non venga presentata esplicitamente, la si dà praticamente per acquisita) e il fatto incidentale che nel 1985 sia stata in assoluto la prima soluzione a balenare per la mente dell'allora adolescente studentello liceale che oggi scrive il presente articolo, ancora oggi ci si imbatte fin troppo spesso in disperate richieste di aiuto da parte di programmatori d'ogni età e livello di esperienza impantanati in soluzioni farraginose e dispersive quando messi alle prese per le più varie esigenze con la generazione di oggetti combinatori isomorfi a sottoinsiemi.

5.5 Soluzione in COMAL 80.

Si propone nel seguito una soluzione completa in COMAL 80 (versione 2.01 per Commodore 64) al quesito, che mantiene inalterate le scelte di design alla base della soluzione alternativa in BASIC V2 sopra proposta, la quale ha già da sola consentito un raddoppio prestazionale rispetto a quanto concepito dagli autori Lee e Scrimshaw.

L'array `digits(10)` è ovviamente il vettore booleano utilizzato per rappresentare il sottoinsieme delle cifre di volta in volta utilizzate, garantendone l'unicità. Le altre stringhe (che in COMAL devono essere dichiarate) sono utilizzate per la conversione testuale dei valori, in modo da facilitarne il controllo per singole cifre. Per il resto abbiamo solamente contatori e variabili di induzione dai nomi sufficientemente esplicativi, ove ciò abbia senso, grazie alle capacità sintattiche estese di COMAL.

Figura 5.5.1: Soluzione alternativa in COMAL 80.

Interessante notare come COMAL 80 segreghi automaticamente lo *scope* delle variabili di induzione, rendendole non visibili al di fuori del contesto FOR nel quale vengono utilizzate. Per tale scopo, dovendo controllare a valle il valore di `j`, si è fatto ricorso ad un costrutto di loop con inizializzazione a monte e incremento esplicito, in modo da estendere lo *scope* della variabile per quanto ci necessitava.

```
//*******************************
//  save hymns
//*******************************

DIM b'digits(10)
DIM hymn2$ OF 3
DIM hymn3$ OF 3
DIM t$ OF 6

TIME 0
PAGE
PRINT "*************************"
PRINT "*** I numeri degli inni ***"
PRINT "** Soluzione alternativa **"
PRINT "** con array booleano.   **"
PRINT "*************************"
PRINT

// Loop esterno: centinaia
FOR di2:=1 TO 3 DO
  b'digits(1+di2):=TRUE

  // Loop intermedio: decine
  FOR di1:=0 TO 9 DO
    IF b'digits(1+di1)=TRUE THEN GOTO next'di1
    b'digits(1+di1):=TRUE
```

```
    // Loop interno: unita'
    FOR di0:=0 TO 9 DO
      IF b'digits(1+di0)=TRUE THEN GOTO next'di0
      b'digits(1+di0):=TRUE
      hymn1:=100*di2+10*di1+di0

      IF hymn1>329 THEN
        PRINT
        PRINT "*************************"
        PRINT "Run time: ";
        PRINT TIME/60,"s"
        PRINT "*************************"
        END "Fine lavoro"
      ENDIF

      hymn2$:=STR$(hymn1+hymn1)
      hymn3$:=STR$(3*hymn1)
      t$:=hymn2$+hymn3$

      j:=1
      WHILE j<7 DO
        i:=1+VAL(t$(j))
        IF b'digits(i)=TRUE THEN GOTO undo
        b'digits(i):=TRUE
        j:+1
      ENDWHILE

      // Stampa la terna trovata
      PRINT hymn1,", ",hymn2$,", ",hymn3$

undo:
      IF j>1 THEN
        FOR k:=1 TO j-1 DO
          b'digits(1+VAL(t$(k))):=FALSE
        ENDFOR k
      ENDIF

      b'digits(1+di0):=FALSE
next'di0:
    ENDFOR di0

    b'digits(1+di1):=FALSE
next'di1:
  ENDFOR di1

  b'digits(1+di2):=FALSE
ENDFOR di2
```

Lo screenshot in fig. 5.5.1 è ottenuto con il programma in COMAL 80 V2.0 presentato nel listato soprastante. Sorprendentemente, nonostante la inerente lentezza generale dell'interprete sempre dichiarata dagli stessi autori dell'implementazione, si è conseguito **un ulteriore raddoppio prestazionale** semplicemente passando da BASIC a COMAL, senza modificare minimamente gli algoritmi e le strutture dati impiegate. Si tratta di un dato su cui riflettere, sicuramente significativo in questo contesto.

5.6 Conclusioni.

Si è presentato nella sua forma originale il primo problema, «I numeri degli inni» tratto dal testo [LS83], accompagnato dalla soluzione in BASIC V2 riportata nel medesimo testo. Avendo all'epoca notato che le prestazioni potevano facilmente essere migliorate con l'uso di strutture dati più idonee, si è presentata anche la

soluzione alternativa elaborata originariamente, sempre in linguaggio BASIC V2, che senza sforzo realizza una prestazione a runtime doppiamente migliore. Negli stessi anni, come ulteriore fonte di divertimento, l'Autore aveva implementato anche una soluzione in COMAL 80, parimenti qui presentata in forma originale, riuscendo ad ottenere un tempo di esecuzione di circa cinque volte inferiore rispetto alla soluzione del testo: risultato decisamente notevole per un linguaggio interpretato al pari del BASIC. Naturalmente sono possibili ulteriori miglioramenti, in special modo ricorrendo all'Assembly 6502/6510, e si invitano i lettori a far pervenire le loro implementazioni originali.

Capitolo 6

«11: Il numero di Sarah».

6.1 Introduzione.

Continuiamo la serie dedicata ai semplici problemi numerici tratti da un testo all'epoca molto diffuso e apprezzato, "The Commodore Puzzle Book - BASIC Brainteasers" di Lee e Scrimshaw [LS83] che proponeva puzzle e rompicapo logici completi di soluzioni in linguaggio BASIC V2 per Commodore 64. Il linguaggio di questa serie di articoli, lo ripetiamo per i nuovi lettori, è intenzionalmente informale, divulgativo e didascalico, scevro da particolari pretese di rigore, al fine di raggiungere il più vasto uditorio informatico possibile. Nel medesimo spirito, anche a rischio di annoiare qualche lettore più evoluto, si cerca in ogni puntata di squadernare in modo pedissequo quell'idea computazionale, quella strategia algoritmica, quella peculiare struttura dati o quel particolare approccio di problem solving che di volta in volta rendono la soluzione interessante, didatticamente valida e ragionevolmente efficiente, pur mantenendone la natura illustrativa e spesso rinunciando ad ulteriori ottimizzazioni più strettamente legate all'architettura.
L'articolo è organizzato in tre parti principali. Nella prima parte si illustra il problema nella sua forma originale, proponendone anche una immediata traduzione dall'inglese. Nella seconda parte si presenta la scarna soluzione originale del testo. Nella terza e ultima parte si discute una soluzione completamente autonoma da parte del calcolatore, senza alcun passaggio manuale, rivelando la reale natura del problema e dell'algoritmo risolutivo, che si presta ad una scelta totalmente automatizzabile della soluzione corretta tra quelle potenzialmente valide. Tale soluzione viene implementata in BASIC V2 e, come da tradizione in questa serie, anche in COMAL 2.01, sul quale – con la scusa di questi semplici quesiti – si vuole richiamare in particolare l'attenzione del lettore. Linguaggio ingiustamente poco noto e valorizzato, COMAL espone caratteristiche ottimali sia ingegneristicamente (leggibilità, concisione, strutturazione, espressività, manutenibilità...) che spesso anche dal punto di vista prestazionale.

6.2 Sarah's Number.

Il problema originale, a pagina 12 del testo [LS83], è caratterizzato dal numero progressivo 11. Eccone la traduzione, seguita dal testo originale nel riquadro: «La scorsa settimana, mentre stavamo navigando sullo yacht degli Henderson, Julie, Ken, Liz, Morris e Naomi (ovvero J, K, L, M, N) cercavano di ricordare il numero di telefono di Sarah. Tutti d'accordo sulla parte iniziale, ma nessuno ricordava esattamente le ultime quattro cifre. Ciascuno di loro fece una affermazione [inerente le sole ultime quattro cifre]:

J	Il numero contiene (almeno) un 9.
K	Il numero è molto maggiore di 5000.
L	Il numero è palindromo (non cambia se letto da destra a sinistra o viceversa).
M	Si tratta di un numero pari.
N	Il suo modulo gaussiano (la somma ricorsiva delle cifre) è pari a nove.

All'arrivo a casa, ho controllato il numero sulla rubrica e ho scoperto che tutte le affermazioni erano corrette, tranne una. Se avessi saputo chi aveva sbagliato, avrei potuto risalire immediatamente al numero in questione. Qual è dunque il numero di Sarah, e chi ha prodotto l'asserzione sbagliata?»

Last week while we were out sailing on the Henderson's yacht, Julie, Ken, Liz, Morris and Naomi were trying to remember Sarah's phone number.
They all agreed on the prefix, but none of them could remember exactly the last four digits. They came up with these statements:

Julie:	*There is a 9 in it.*
Ken:	*It is definitely higher than 5000.*
Liz:	*It is palindromic: it reads the same forwards as backwards*
Morris:	*The number is even.*
Naomi:	*It has a digital root of 9*.*

On arriving home, I looked up the phone number and found that they were all correct except for one. Had I known who was wrong, I could have discovered the number. What was Sarah's number, and who made the incorrect statement?
**The digital root of a number is obtained by adding together the digits until a single digit remains, e.g. the digital root of 8765 is:*
$8+7+6+5=26,\ 2+6=8$

6.3 La soluzione originale.

La soluzione proposta a pag. 62 di [LS83] si limita sostanzialmente a generare *tutti* i numeri di quattro cifre, selezionando quelli che soddisfano esattamente quattro delle cinque condizioni stabilite dalle asserzioni J, K, L, M, N. L'unica euristica adottata consiste nel limitare a priori la ricerca ai soli numeri palindromi (condizione L), il che delimita il range di generazione ai valori tra 0 e 99 inclusi, rendendo immediatamente il loop principale più efficiente di un fattore cento rispetto all'implementazione più *naïf*. Il semplice programma contiene varie ingenuità implementative, probabilmente per invogliare il lettore ad applicare le necessarie migliorie, e usa il più banale dei pattern per la verifica di proprietà binarie multiple, ossia un contatore threshold a somma non pesata. L'utilità del programma proposto dagli autori è ulteriormente limitata dal fatto che esso fornisce in output un mero elenco dei valori, senza indicazione esplicita delle condizioni di volta in volta verificate. Proponiamo di seguito le considerazioni che accompagnano il codice, seguite dal relativo listato BASIC V2.

It is possible to check all the numbers between 0000 and 9999 against a permutation of four out of the five statements to see which gives a unique answer, but doing so would be a rather lengthy process. However, we know that only one of the statements is incorrect, so the other four must be correct. Let's look at Liz's statement. Now, there are only 100 possible fourdigit palindromic numbers - those having the first two digits 00 to 99 followed by these two digits in reverse order. If we assume that Liz's statement is one of the correct ones, we need only test these hundred possibilities against the other four statements. In fact, we can deduce that her statement is correct as it is quite easy to find many possibilities that satisfy all of the other conditions. In other words, there is more than one number that is more than 5000, has a "9" in it, is even, and has a digital root of 9. Two such numbers that can be easily discovered are 7398 and 7938. As this fails to provide an unique answer, Liz's statement must be correct, and one of the others is the wrong one. Therefore, we can use this as a basis for our program. The program checks all four-digit palindromic numbers and prints out any that are found to agree with three of the four remaining statements.

```
10 FOR N = 0 TO 99
20 N$ = MID$(STR$(N),2)
30 IF (N < 10) THEN N$ = "0" + N$
40 N$ = N$+RIGHT$(N$,I)+LEFT$(N$,I)
50 T = 0
100 FOR M = 1 TO 4
110 IF (MID$(N$,M,1) = "9") THEN T = 1
120 NEXT
200 IF (VAL(N$) > 5000) THEN T = T+1
300 IF (VAL(N$)/2 = INT(VAL(N$)/2) THEN T = T+1
400 C = 0
410 FOR M = 1 TO 4
420 C = C + VAL(MID$(N$,M,I)
430 NEXT
440 IF (C = 9) OR (C = 18) OR (C = 27) OR (C = 36) THEN T = T+1
```

```
500 IF (T = 3) THEN PRINT N$
510 NEXT
```

Figura 6.3.1: La soluzione originale del testo.

Nell'appendice dedicata alle soluzioni vere e proprie, a pag. 118, gli autori riportano poi una soluzione tabulare completa, calcolata con carta e penna. Riproponiamo qui tale tabella, con una veste grafica leggermente diversa per evidenziare la natura di matrice booleana del problema, in quanto tale perfettamente computabile con minimo sforzo algoritmico. A tale scopo lavoriamo in logica negata e marchiamo con un simbolo unicamente le asserzioni che risultano **false** per ciascun valore numerico trovato, riportato nella colonna più a sinistra. Ad esempio, il valore 6996 verifica a priori la condizione L e anche le condizioni J, K ed M in quanto rispettivamente:

L) Si tratta di un numero palindromo;
J) Il numero contiene (almeno) un 9;
K) Risulta maggiore di 5000;
M) Si tratta di un numero pari.

In tabella risulterà quindi marcata la sola condizione falsa N, in quanto il modulo gaussiano di 6996 è pari a 3.

#	Julie	Ken	Morris	Naomi
0990		✖		
6336	✖			
6996				✖
8118	✖			
8998				✖
9009			✖	
9999			✖	

La soluzione è quindi 0990 e l'affermazione falsa è quella di Ken, perché la negazione di qualsiasi altra condizione non porta a soluzioni univoche, il che è caratteristica implicita nella descrizione del problema: «*Had I known who was wrong, I could have discovered the number.*». A questo punto risulta decisamente ovvia la natura del problema come ricerca dei valori singolari **per colonne** in una matrice binaria: un problema molto comune, che il nostro C64 può risolvere in piena autonomia, senza alcun passaggio manuale.

6.4 Soluzione alternativa in BASIC V2.

Si propone nel seguito una soluzione completa in BASIC V2 al quesito del testo, in grado di memorizzare i risultati parziali in una opportuna coppia di array ed analizzare autonomamente l'unicità della soluzione. In questo modo sarà il calcolatore a fornire direttamente la risposta finale. Si noti che questo genere di risoluzione si applica a decine di problemi eterogenei riconducibili alla medesima natura, da passatempi e rompicapo a reali problemi di ottimizzazione, logistica, automazione industriale, contabilità e molto altro, quindi vale la pena di considerare seriamente il metodo, seppure presentato in un contesto puramente ludico e con una implementazione puramente didattica.

Qualche breve nota sull'implementazione, che con le sue 30 linee di codice dovrebbe essere perfettamente intuitiva anche per i neofiti. Si è sostanzialmente

Figura 6.4.1: Soluzione alternativa in BASIC V2.

mantenuta l'impostazione fortemente didattica e didascalica della soluzione originale. La struttura dati principale destinata a contenere la matrice booleana è denominata `co%()`ovvero `conditions`. Associato indirettamente alla matrice per indici di riga corrispondenti l'array `so$` (lapalissianamente `solutions`) dei numeri che soddisfano esattamente tre delle quattro condizioni previste. Si è deliberatamente scelto di mantenere elementare e intuitiva l'intera implementazione, evitando di ricorrere ad una vera matrice binaria (bit array), come invece si farebbe in altri contesti e linguaggi, al solo fine di rendere più facile e immediata la comprensione dell'algoritmo e delle indicizzazioni. Lo stesso vale per la memorizzazione del numero telefonico (a rigore, la sua parte finale) sotto forma di stringa, con la preservazione degli zeri iniziali. Il tutto senza preoccuparsi di comprimere il tempo di esecuzione e/o il costo di memorizzazione i quali, date le irrisorie dimensioni dello spazio delle soluzioni esplorato, sono decisamente poco rilevanti e passano in secondo piano rispetto alla didatticità e chiarezza del codice.

Si noti infine che la scansione della matrice binaria si arresta alla prima colonna che contenga un singolo valore nullo, il che è una concessione puramente didattica alla fiducia nella corretta costruzione della matrice e in ultima analisi nella esistenza e unicità della soluzione: nel mondo reale un controllo di congruenza esaustivo sarebbe quantomeno doveroso per la robustezza dell'applicazione e la coerenza dei dati. Allo stesso modo si è dato per scontato il dimensionamento iniziale degli array: ma confidiamo che qui anche il lettore meno famigliare col «Testo Sacro» [Sta86] e poco a suo agio nel calcolo a priori della dimensione dello spazio delle soluzioni concorderà sulla ragionevolezza di una tale assunzione, alla quale si può facilmente giungere anche col più empirico dei metodi, ossia una esecuzione preliminare del loop di generazione ed enumerazione, strutturato in modo simile alla soluzione originale degli autori Lee e Scrimshaw.

Un'ultima nota riguardo al calcolo del modulo gaussiano. La formula per il calcolo diretto del modulo, senza sommatoria iterativa, è ben nota:

$$dr(n) = \begin{cases} n \,(mod\, 9) & n \not\equiv 0 \,(mod\, 9) \\ 9 & n \equiv 0 \,(mod\, 9) \end{cases} \tag{6.4.1}$$

Il che si riduce immediatamente alla singola forma chiusa che segue:

$$dr(n) = 1 + [n - 1 \,(mod\, 9)] \tag{6.4.2}$$

L'applicazione di tale formula, lasciata come semplice esercizio per il lettore interessato, consente di semplificare leggermente il codice della soluzione, comunque situandosi un passo al di là degli scopi prefissati di chiarezza e soprattutto completa autonomia della soluzione stessa. Il guadagno prestazionale che ne consegue è globalmente piuttosto modesto, sia a causa del numero ridottissimo di iterazioni, sia considerando che divisione e modulo non sono implementati in silicio sulla CPU 6502/6510 ma realizzati tramite il firmware di sistema detto Kernal.

```
100 print "{clear}************************"
101 print "** 11: sarah's number **"
102 print "************************"
103 print "####       j   k   m   n"
105 print "_____________________________"
110 so = 1 : dim co%(7,4), so$(7), na$(4)
120 na$(1) = "julie": na$(2) = "ken": na$(3) = "morris": na$(4) = "naomi"
130 for n = 0 to 99
140 n$ = right$("0" + mid$(str$(n), 2), 2)
150 n$ = n$ + right$(n$, 1) + left$(n$, 1)
160 for i = 1 to 4: co%(so, i) = 0: next
170 t = 0
200 if (mid$(n$, 1, 1) = "9") or (mid$(n$, 2, 1) = "9") then t = 1 : co%(so, 1)
    = 1
210 if (val(n$) > 5000) then t = t + 1 : co%(so, 2) = 1
220 if (val(n$) / 2 = int(val(n$) / 2)) then t = t + 1: co%(so, 3) = 1
230 c = 0
240 for m = 1 to 4
250 c = c + val(mid$(n$, m, 1))
260 next m
270 if (c > 9) then c = val(right$(mid$(str$(c),2), 1)) + val(left$(mid$(str$(c)
    ,2), 1))
280 if (c = 9) then t = t + 1: co%(so, 4) = 1
290 if (t <> 3) then goto 330
```

```
300 so$(so) = n$ : ? n$,
310 for i = 1 to 4: ? co%(so, i); " ";: next i:?
320 so = so +1
330 next n
400 for c = 1 to 4: t = 0
410 for r = 1 to 7
420 if co%(r, c) = 0 then t = t +1 : rs = r
430 next r
440 if t = 1 then print : print "il numero di sarah e' ";so$(rs);" e ";
na$(c); " ha sbagliato!": end
450 next c
```

6.5 Soluzione alternativa in COMAL 80.

Si propone nel seguito una soluzione completa in COMAL 80 (2.01 per Commodore 64) al quesito. Si è sostanzialmente mantenuta l'impostazione fortemente didattica e didascalica della soluzione in BASIC proposta sopra. Per non dilungare oltre il dovuto la trattazione, invitiamo il lettore a confrontare linea per linea le due implementazioni, traendo le dovute conclusioni.

Figura 6.5.1: Soluzione alternativa in COMAL 80.

Le strutture dati utilizzate sono identiche a quelle già impiegate: in particolare, la matrice booleana delle condizioni `conditions()` e l'array `solution$()` dei numeri che soddisfano esattamente tre delle quattro condizioni previste. Le differenze rispetto all'analogo in BASIC V2, in questo caso, sono minime e sono principalmente dovute alle caratteristiche sintattiche di COMAL: per taluni versi più prolisso, ma notevolmente più leggibile, ordinato e con costrutti più potenti, si veda ad esempio la sintassi per la manipolazione delle stringhe, che supporta l'indicizzazione diretta e perfino lo *slicing*. Ricordando che COMAL consente manipolazioni avanzate che includono l'assegnazione diretta di sottostringhe (impossibile in BASIC) e lo slicing di elementi di array di testo, si veda la tabella seguente che riporta i principali costrutti equivalenti per la manipolazione di sottostringhe:

BASIC V2	COMAL 80
`T$=LEFT$(A$, 4)`	`T$:=A$(:4)`
`T$=RIGHT$(A$,2)`	`T$:=A$(LEN(A$)-1:)`
`T$=MID$(A$,5,1)`	`T$:=A$(5)`
`T$=MID$(A$,3,3)`	`T$:=A$(3:6)`

Un'ultima nota riguardo al calcolo del modulo gaussiano. Come già ricordato in precedenza, l'applicazione della formula diretta consente in linea teorica di semplificare leggermente il codice della soluzione: tuttavia, il guadagno prestazionale che ne consegue è globalmente piuttosto modesto data la mancanza di supporto hardware per le operazioni di divisione e modulo sulle CPU MOS 6502/6510.

```
//*******************************
//** save Sarah
//*******************************

DATA "Julie","Ken","Morris","Naomi"
DIM conditions(7,4)
DIM solution$(7) OF 4
DIM name$(4) OF 6
DIM phone$ OF 4
```

```
TIME 0
PAGE
PRINT "************************"
PRINT "** 11: Sarah's Number **"
PRINT "************************"
PRINT "####        J    K    M    N"
PRINT "-----------------------------"

sol:=1
i:=1
WHILE NOT EOD DO
  READ name$(i)
  i:+1
ENDWHILE

FOR n:=0 TO 99 DO
  phone$:=STR$(n)
  IF n<10 THEN phone$:="0"+phone$
  phone$:+phone$(2)+phone$(1)
  number:=VAL(phone$)
  FOR i:=1 TO 4 DO conditions(sol,i):=FALSE
  cond'cnt:=0

  //** Condizione 1 (J)
  IF phone$(1)="9" OR phone$(2)="9" THEN
    cond'cnt:=1
    conditions(sol,1):=TRUE
  ENDIF

  //** Condizione 2 (K)
  IF number>5000 THEN
    cond'cnt:+1
    conditions(sol,2):=TRUE
  ENDIF

  //** Condizione 3 (M)
  IF number/2=number DIV 2 THEN
    cond'cnt:+1
    conditions(sol,3):=TRUE
  ENDIF

  //** Condizione 4 (N)
  d'sum:=1+((number-1) MOD 9)
  IF d'sum=9 THEN
    cond'cnt:+1
    conditions(sol,4):=TRUE
  ENDIF

  IF cond'cnt=3 THEN
    solution$(sol):=phone$
    PRINT phone$;"      ";
    FOR i:=1 TO 4 DO
      PRINT conditions(sol,i);" ";
    ENDFOR i
    PRINT
    sol:+1
  ENDIF
ENDFOR n

//*************************
//** Stampa dei risultati **
```

```
//*************************

FOR c:=1 TO 4 DO
  cond'cnt:=0
  FOR r:=1 TO 7 DO
    IF conditions(r,c)=FALSE THEN
      cond'cnt:+1
      rs:=r
    ENDIF
  ENDFOR r
  IF cond'cnt=1 THEN
    PRINT
    PRINT "Il numero di Sarah e'";
    PRINT solution$(rs)," e"
    PRINT name$(c)," ha sbagliato!"
    END "** Fine lavoro **"
  ENDIF
ENDFOR c
```

6.6 Conclusioni.

Dopo aver presentato il testo originale del problema n° 11 proposto in [LS83], completo di relativa soluzione e implementazione in BASIC V2, si sono introdotte alcune elementari considerazioni per una soluzione alternativa in grado di rendere più flessibile e intuitiva la soluzione stessa. Si è infine presentata una implementazione in COMAL della medesima soluzione, per evidenziarne in un confronto diretto le caratteristiche di leggibilità, potenza espressiva e strutturazione del codice.

Capitolo 7

«46: COMPUTER».

7.1 Introduzione.

Continuiamo la serie dedicata ai semplici problemi numerici tratti da un testo all'epoca molto diffuso e apprezzato, "The Commodore Puzzle Book - BASIC Brainteasers" di Lee e Scrimshaw [LS83] che proponeva puzzle e rompicapo logici completi di soluzioni in linguaggio BASIC V2 per Commodore 64. Il linguaggio dell'intera serie di articoli su puzzle e affini, lo ripetiamo ogni volta a beneficio dei nuovi lettori, è sempre tenuto intenzionalmente informale, divulgativo e didascalico, scevro da particolari pretese di rigore, al fine di raggiungere il più vasto uditorio (retro)informatico possibile. Nel medesimo spirito, anche a rischio di annoiare qualche lettore più evoluto, si cerca in ogni puntata di squadernare in modo pedissequo quell'idea computazionale, quella strategia algoritmica, quella peculiare struttura dati o quel particolare approccio di problem solving che di volta in volta rendono la soluzione interessante, didatticamente valida e ragionevolmente efficiente, pur mantenendone la natura illustrativa e spesso rinunciando ad ulteriori ottimizzazioni più strettamente legate all'architettura.

L'articolo è organizzato in tre parti principali. Nella prima parte si illustra semplicemente il problema nella sua forma originale, mentre nella seconda parte si presenta la soluzione proposta dagli autori del testo. Nella terza e ultima parte si discutono le soluzioni alternative, che comportano un netto miglioramento prestazionale e fanno uso di strutture dati elementari maggiormente efficienti: tali soluzioni vengono implementate in BASIC V2 e, come da tradizione in questa serie, anche in COMAL 2.01, sul quale – con la scusa di questi semplici quesiti – si vuole richiamare in particolare l'attenzione del lettore. Linguaggio ingiustamente poco noto e valorizzato, COMAL espone caratteristiche ottimali sia ingegneristicamente (leggibilità, concisione, strutturazione, espressività, manutenibilità...) che spesso anche dal punto di vista prestazionale.

7.2 «COMPUTER».

Il problema originale, il numero 46 a pagina 41 del testo [LS83], è uno degli ultimi tra i cinquanta quesiti proposti nel libro. Eccone di seguito il testo completo:

> *Professor Otto Hex, of the Institute of Advanced Mathematics, was recently amusing himself with one of the computers when he discovered the square root of COMPUTER.*
> *Before you accuse me of inaccurate reporting, let me explain. What the professor had done, was to substitute the letters of the world COMPUTER for numbers, and then find the square root of this number. Then by re-substituting letters for the digits in his result, he obtained a familiar English word.*
> *I ought to add that the word you are after has all of its letters different and, of course, contains only letters which appear in the the word COMPUTER.*
> *As is usual with all letter-for-digits puzzles of this type, each letter stands for a different digit, and the same letter stands for the same digit throughout.*

Il testo purtroppo non è particolarmente chiaro, e va interpretato secondo le abitudini delle riviste enigmistiche dell'epoca. Sotto l'apparenza degli anagrammi, il quesito cela in realtà un semplice problema di teoria del numero: si tratta di trovare un numero naturale di quattro cifre, espresso in base decimale, con le proprietà elencate di seguito.

1. Il numero n risulta composto di quattro cifre $d_3d_2d_1d_0$ tutte diverse tra loro, ciascuna delle quali compare una e una sola volta.

2. Il suo quadrato n^2 consta di otto cifre, anch'esse tutte differenti tra loro: in altri termini, l'insieme delle sue cifre è un sottoinsieme proprio dell'insieme delle cifre decimali $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

3. Tutte le cifre $\{d_3, d_2, d_1, d_0\}$ che compongono il numero n compaiono anche in n^2.

4. Usando n^2 come chiave di sostituzione, le lettere prese ordinatamente dalla parola *COMPUTER* e sostituite alle singole cifre $d_3 d_2 d_1 d_0$ formeranno un'altra parola inglese di senso compiuto.

Un esempio chiarirà meglio le intenzioni degli autori al punto 4. Posto $n = 4.913$, si ha $n^2 = 24.137.569$, una coppia di valori che rispetta le prime tre proprietà (le cifre di n risultano sottolineate per evidenziarle in n^2). Si ha quindi la seguente codifica:

$$\begin{array}{cccccccc} C & O & M & P & U & T & E & R \\ 2 & 4 & 1 & 3 & 7 & 5 & 6 & 9 \end{array} \tag{7.2.1}$$

Ne risultano ordinatamente le seguenti assegnazioni: $4 = O$, $9 = R$, $1 = M$, $3 = P$ e la parola risultante è *ORMP*, in questo caso un aggregato di lettere privo di senso compiuto. Sebbene esistano vari valori di n con le prime tre proprietà, solo uno di essi le espone tutte e quattro. Poiché la codifica stabilita dalla corrispondenza biunivoca tra le lettere della parola *COMPUTER* e le cifre di n^2 varia per ogni valore di n, è opportuno sottolineare subito che vi saranno delle collisioni nelle parole di quattro lettere risultanti: ad esempio, anche i valori $n = 4.967$ e $n = 4.987$ danno luogo al medesimo risultato *ORMP* (la dimostrazione è lasciata come banale esercizio per il lettore).

Aggiungiamo solo una breve considerazione sugli intervalli di generazione. Dal momento che non sono ammesse cifre ripetute, il più piccolo valore da considerare per n^2 è pari a 10.234.567 mentre il più grande è 98.765.432. Passando alle radici, si avrebbe quindi $3.199 \leq n \leq 9.938$, ma in realtà la proprietà 1. restringe l'intervallo a $[3.201,\ 9.876]$.

7.3 La soluzione originale.

La soluzione proposta a pag. 107÷108 di [LS83] genera direttamente tutti i valori numerici di base nell'intervallo $[3.201,\ 9.876]$, li eleva al quadrato ed effettua tutti i controlli previsti. Nel caso in cui le prime tre condizioni vengano rispettate, genera il relativo anagramma (a rigore, un sub-anagramma che usa solo 4 lettere) del termine *COMPUTER* e lo stampa a video, lasciando all'utente l'individuazione del singolo termine di senso compiuto tra quelli visualizzati.

L'idioma `str$(int(nu*nu))` che compare nel codice era un workaround molto noto all'epoca, e gli stessi autori fanno menzione del problema a pag. 46, insieme ad altri «programming tips», nel paragrafo dedicato all'accuratezza del calcolo floating point. Purtroppo il problema ha origine nel fatto che le CPU 6502/6510, come noto, non dispongono di un moltiplicatore hardware.

```
100 print "{clear}************************"
110 PRINT "***** 46: computer  *****"
120 print "** soluzione originale **"
130 print "************************"
140 s = ti
150 b$ = "computer"
160 for a = 3201 to 9876
170 a$ = mid$(str$(a),2)
180 l = 4
190 c = 0
200 gosub 450
210 if (c <> 0) then 430
220 a$ = MID$(STR$(INT(a*a)),2)
230 l = 8
240 c = 0
250 gosub 450
260 if (c <> 0) then 430
270 x$ = MID$(STR$(a)+A$,2)
280 c = 0
290 for n = 1 to 4
300 for m = 5 to 12
310 if (MID$(x$,n,1) = MID$(x$,m,1)) THEN c = c + 1
320 next
330 next
340 if (c <> 4) THEN 430
```

```
350 c$ = MID$(STR$(A),2)
360 d$ = ""
370 for m = 1 to 4
380 for n = 1 to 8
390 if (MID$(a$,n,1) = MID$(c$,m,1)) THEN d$ = d$ + MID$(b$,n,1)
400 next
410 next
420 print a$, c$, d$
430 next
440 print "run time: "; (ti-s)/60;"s": end
450 for n = 1 to (l-1)
460 for m = (n+1) to l
470 if (MID$(a$,n,1) = MID$(a$,m,1)) THEN c = c + 1
480 next
490 next
500 return
```

Figura 7.3.1: La soluzione originale del testo.

La struttura fondamentale del programma risulta molto semplice e presenta numerosi elementi comuni con un altro problema già affrontato in questa serie di articoli ("I numeri dei salmi"), sebbene con fattori di scala superiori: il tempo di elaborazione passa infatti dai 45 secondi all'ordine dei 40 minuti.

Dal punto di vista combinatorio, i valori di base da generare sono caratterizzati dalle seguenti proprietà:

1. L'ordine delle cifre è ovviamente importante.
2. L'ampiezza della presentazione (quattro cifre) è strettamente minore del numero di simboli disponibili (dieci).
3. Non sono ammesse ripetizioni.

Pertanto, per definizione, si tratta di generare restrizioni di *disposizioni semplici* di dieci simboli su quattro posizioni. Gli autori, tuttavia, in questo caso hanno scelto di generare indifferenziatamente tutti i valori numerici compresi nell'intervallo sopra ricordato, usando un singolo loop.

Come è immediato rilevare dalla lettura del codice, il programma fa poi uso di loop annidati per la verifica delle proprietà sulle stringhe corrispondenti al numero generato ed al suo quadrato. Osservando i loop annidati alle linee $290 \div 330$ e $370 \div 410$ risulta evidente la farraginosità ed inefficienza del metodo impiegato per il confronto diretto tra le cifre, considerando anche che vengono generati e convalidati più di 6.600 valori. Tutto ciò si traduce, come accennato poco sopra, in un tempo di esecuzione dell'ordine di ben $2.400''$! Fortunatamente è piuttosto semplice raddoppiare le prestazioni, sempre restando al caro vecchio BASIC V2 e senza sforzarsi eccessivamente.

Si noti, incidentalmente, che il valore cercato è $n = 4.097 = 2^{12} + 1$ e la parola risultante è `TERM` (che, conoscendo l'inclinazione faceta degli autori, possiamo anche abbinare semanticamente e logicamente al termine originale, `COMPUTER`, considerandola come la comune abbreviazione di `TERMINAL`).

7.4 Soluzione alternativa in BASIC V2.

Si propone nel seguito una soluzione completa in BASIC V2 al quesito del testo, in grado di migliorare già notevolmente la prestazione a runtime, pur rispettando l'impostazione fortemente didattica dell'intera soluzione.

Riprendiamo qui alcune considerazioni già viste per problemi strettamente analoghi tratti dal medesimo testo, come «i numeri degli inni» già richiamato in precedenza. Ragionando dal punto di vista meramente combinatorio, una parte del quesito si riduce essenzialmente alla generazione di disposizioni semplici di dieci elementi su quattro posizioni. Poiché le ripetizioni sono proibite, si tratta quindi di selezionare un opportuno sottoinsieme proprio dall'insieme di partenza $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$: si veda a tale proposito la nota 2.

Con l'uso di tale struttura dati risulta immediato controllare se un dato elemento appartiene o meno al sottoinsieme considerato, e quindi nel caso specifico se una cifra è già stata eventualmente utilizzata: è sufficiente

un accesso alla memoria, alla posizione $i - esima$ dell'array, dove i è la cifra attualmente in fase di controllo. A questo scopo sono previsti nel programma i due array `di()` e `dd()`, per determinare l'unicità delle cifre nel valore di base e nel suo quadrato rispettivamente. In questo modo anche il controllo della condizione 3. risulta grandemente semplificato, con l'ausilio di un semplice contatore `ma` (match) che viene incrementato ad ogni corrispondenza binaria.

```
100 REM ********************
101 REM 46: computer
102 REM ********************
105 print "{clear}**************************"
110 PRINT "****** 46: computer  ******"
115 print "** soluzione alternativa **"
120 print "**************************"
130 s = ti
140 dim di(10): REM booleano cifre valore base
150 dim dd(10): REM booleano cifre quadrato
160 dim rl(10): REM reverse lookup cifre
170 dim po(4): REM anagramma
180 s$ = "computer"
190 REM loop migliaia
200 for d4 = 3 to 9
210 if di(d4+1) = 1 then goto 660
220 di(d4+1) = 1
230 rl(d4+1) = 1
240 rem loop centinaia
250 for d3 = 0 to 9
260 if di(d3+1) = 1 then goto 630
270 di(d3+1) = 1
280 rl(d3+1) = 2
290 REM loop decine
300 for d2 = 0 to 9
310 if di(d2+1) = 1 then goto 600
320 di(d2+1) = 1
330 rl(d2+1) = 3
340 REM loop unita'
350 for d1 = 0 to 9
360 if di(d1+1) = 1 then goto 570
370 di(d1+1) = 1
380 rl(d1+1) = 4
390 nu = 1000*d4+100*d3+10*d2+d1
400 if nu < 3201 then goto 550
410 for i = 1 to 10: dd(i) = 0: next i
420 sq$ = mid$(str$(int(nu*nu)),2)
430 ma = 0: REM contatore match
440 cn = 0: REM conta cifre uniche
450 REM convalida valori generati
460 for i = 1 to 8
470 j = val(mid$(sq$,i,1))+1
480 if dd(j) = 1 then goto 550
490 dd(j) = 1
500 cn = cn +1
510 if dd(j) = di(j) then ma = ma +1: po(rl(j)) = i
520 next i
530 REM stampa i valori validi
540 if ma = 4 and cn = 8 then gosub 900
550 rl(d1+1) = 0
560 di(d1+1) = 0
570 next d1
580 rl(d2+1) = 0
590 di(d2+1) = 0
600 next d2
```

```
610 rl(d3+1) = 0
620 di(d3+1) = 0
630 next d3
640 rl(d4+1) = 0
650 di(d4+1) = 0
660 next d4
670 print "run time: "; (ti-s)/60;"s": end
900 REM stampa anagramma
910 print sq$;" ";nu;" ";
920 for k = 1 to 4
930 print mid$(s$,po(k),1);
940 next k
950 print
960 return
```

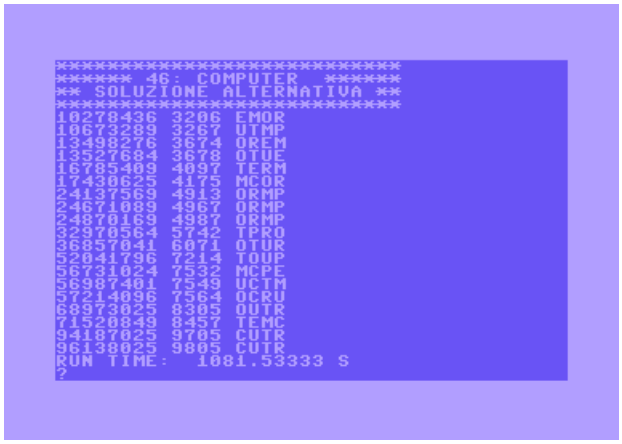


Figura 7.4.1: Soluzione alternativa in BASIC V2.

Ai fini di una maggiore efficienza, si è voluto evitare il loop annidato che troviamo alle linee $370 \div 410$ della soluzione originale per la ricerca lineare della posizione delle cifre di n entro n^2 e quindi, in ultima analisi, entro la parola data `COMPUTER` di volta in volta codificata da n^2 stesso. Pertanto si fa uso di altre due strutture dati lineari per effettuare il *reverse lookup* in $O(1)$ e costruire il sub-anagramma richiesto in modo efficiente per ogni potenziale match.

Riprendiamo l'esempio 7.2.1, dove $n = 4.913$ e quindi l'array `rl()` (**r**everse **l**ookup) sarà così popolato: $rl = \langle 0, 3, 0, 1, 4, 0, 0, 0, 0, 2 \rangle$ in modo tale cioè che, usando l'indicizzazione del BASIC caratterizzata da un offset unitario, $rl(4) = 1$, $rl(9) = 2$, $rl(1) = 3$, $rl(3) = 4$ rispecchiando la posizione delle singole cifre nel numero. L'assegnazione `po(rl(j)) = i` alla linea 510, in caso di esito positivo di tutti i confronti, produrrà al termine del loop $460 \div 520$ il seguente contenuto di `po()`: $po = \langle 2, 8, 3, 4 \rangle$. Si veda la figura 7.4.2 per una intuitiva visualizzazione delle assegnazioni.

Seguendo le frecce con i relativi ordinali in rosso e i colori di sfondo delle caselle, risulta immediato comprendere come viene costruito l'anagramma in base alla posizione delle cifre che compongono il valore n e il suo quadrato, senza applicare esplicitamente alcun riordinamento dei dati nel programma e senza eseguire onerose scansioni lineari per ogni potenziale soluzione trovata. Si noti solo che in questo caso si è scelto di ordinare le cifre in ordine inverso rispetto agli esponenti del dieci, partendo quindi dalla cifra decimale più significativa, per coerenza con l'approccio di scansione della stringa da sinistra a destra e quindi di costruzione dell'anagramma usato nella soluzione originale. Allo stesso modo, per la medesima forma di coerenza indiretta, anche la nomenclatura delle variabili di induzione $d4 \ldots d1$ si riconduce al fatto che in BASIC gli indici partono comunque da 1 (una sorta di contrappasso dantesco per chi, come l'Autore, programma in C da quattro decadi).

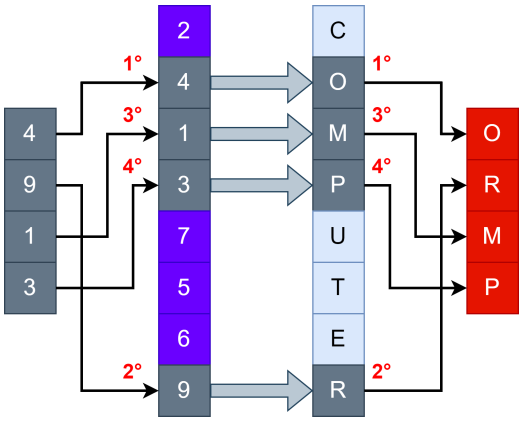


Figura 7.4.2: Costruzione del sub-anagramma dal numero dato e dal suo quadrato.

Per una grossolana valutazione del numero di iterazioni compiute, ricordiamo qui per mera comodità la nota formuletta per le disposizioni semplici di n elementi su k posizioni:

$$D_{n,k} = \frac{n!}{(n-k)!}$$

In questo caso $n = 10$, $k = 4$ e quindi $D_{n,k} = 5.040$, ma in realtà è evidente dal codice che il programma esegue effettivamente 7.000 iterazioni come sancito dai limiti dei quattro loop annidati, saltando però l'elabo-

razione per quei rami improduttivi dell'albero computazionale che non corrispondono a disposizioni semplici (questa fondamentale tecnica di generazione combinatoria è detta *pruning* o anche *early filtering*).

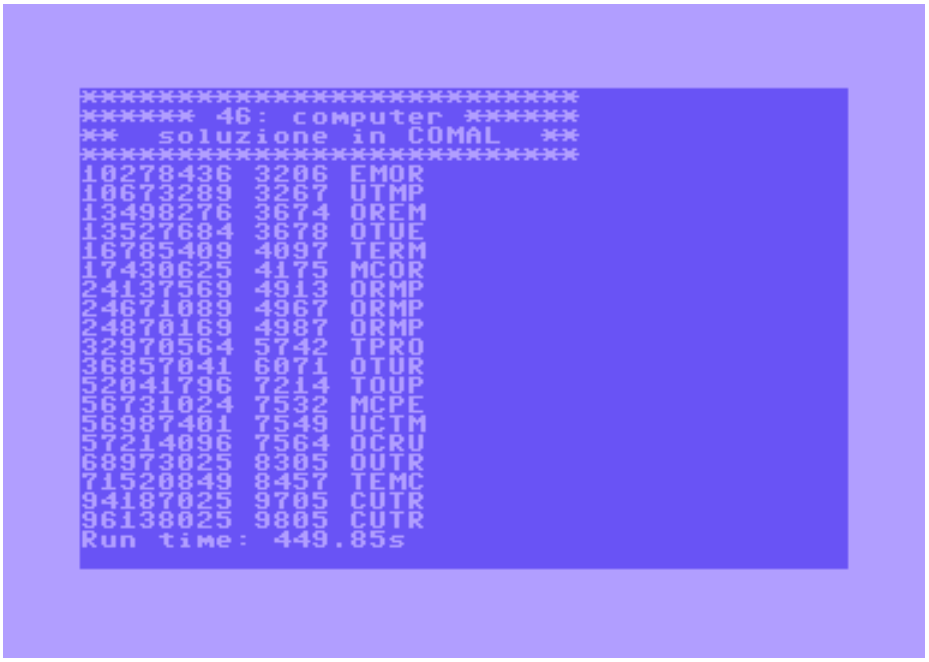
Come risulta evidente dallo screenshot, il guadagno prestazionale ottenuto col semplice impiego di strutture dati più adatte è quantificabile nell'ordine del 55%: in pratica si consegue più di un **raddoppio prestazionale**, pur avendo introdotto un loop a quadruplice annidamento in luogo della generazione sequenziale dei valori tramite singolo loop al fine di garantire per costruzione l'unicità delle cifre, senza onerosi controlli a valle. Tuttavia, è possibile fare di meglio: non solo con l'ovvio ricorso all'Assemby 6502/6510, ma anche con uno dei nostri linguaggi preferiti, il COMAL 80 (versione 2.01 su cartridge) per Commodore 64.

7.5 Soluzione alternativa in COMAL 80.

Il seguente listato, ottenuto con il comando `DISPLAY`, mostra il codice COMAL che realizza la nostra soluzione al quesito in meno di otto minuti, circa **un quinto** rispetto alla soluzione originale del testo, mantenendo esattamente le medesime scelte progettuali e strutture dati in base alle quali abbiamo creato la soluzione alternativa in BASIC V2. A parità di scelte progettuali rispetto alla soluzione BASIC, il listato inoltre risulta notevolmente più ordinato, leggibile, comprensibile ed a prova d'errore, oltre a fornire prestazioni superabili solamente con il ricorso al linguaggio Assembly o a qualche raro compilatore per C64. Anche qui la struttura del programma è stabilita dai quattro loop annidati per la generazione delle disposizioni semplici.

```
//*****************************
// save computer3
//*****************************
PAGE
PRINT "*************************"
PRINT "****** 46: computer ******"
PRINT "**  soluzione in COMAL  **"
PRINT "*************************"

TIME 0
DIM digits(10)
DIM sq'digits(10)
DIM r'lookup(10)
DIM pos(4)
DIM square$ OF 8
DIM word$ OF 8
word$:="COMPUTER"

// Loop migliaia
FOR d4:=3 TO 9 DO
  IF digits(d4+1)=TRUE THEN GOTO thous'end
  digits(d4+1):=TRUE
  r'lookup(d4+1):=1

  // Loop centinaia
  FOR d3:=0 TO 9 DO
    IF digits(d3+1)=TRUE THEN GOTO hunds'end
    digits(d3+1):=TRUE
    r'lookup(d3+1):=2

    // Loop decine
    FOR d2:=0 TO 9 DO
      IF digits(d2+1)=TRUE THEN GOTO tens'end
      digits(d2+1):=TRUE
      r'lookup(d2+1):=3

      // Loop unita'
      FOR d1:=0 TO 9 DO
        IF digits(d1+1)=TRUE THEN GOTO unit'end
        digits(d1+1):=TRUE
        r'lookup(d1+1):=4
```

```
        number:=1000*d4+100*d3+10*d2+d1
        IF number<3201 THEN GOTO exit'val
        IF number>9876 THEN GOTO job'end

        FOR i:=1 TO 10 DO
          sq'digits(i):=FALSE
        ENDFOR i
        square$:=STR$(number*number)
        match:=0
        cnt:=0

        // Convalida i valori generati
        FOR i:=1 TO 8 DO
          j:=VAL(square$(i))+1
          IF sq'digits(j) THEN GOTO exit'val
          sq'digits(j):=TRUE
          cnt:+1
          IF digits(j)=sq'digits(j) THEN
            match:+1
            pos(r'lookup(j)):=i
          ENDIF
        ENDFOR i

        // Stampa i valori validi
        IF match=4 AND cnt=8 THEN print'val

exit'val:
        r'lookup(d1+1):=0
        digits(d1+1):=FALSE

unit'end:
      ENDFOR d1
      r'lookup(d2+1):=0
      digits(d2+1):=FALSE

tens'end:
    ENDFOR d2
    r'lookup(d3+1):=0
    digits(d3+1):=FALSE

hunds'end:
  ENDFOR d3
  r'lookup(d4+1):=0
  digits(d4+1):=FALSE

thous'end:
ENDFOR d4

job'end:
PRINT "Run time: ",TIME/60,"s",
END "Fine lavoro"

PROC print'val
  PRINT square$," ",number," ",
  FOR k:=1 TO 4 DO
    PRINT word$(pos(k)),
  ENDFOR k
  PRINT
ENDPROC print'val
```



7.6 Conclusioni.

Si è presentato nella sua forma originale il problema n° 46 «COMPUTER» tratto dal testo [LS83], accompagnato dalla soluzione in BASIC V2 riportata nel medesimo testo. A fronte della performance tutt'altro che entusiasmante di tale codice, si era all'epoca realizzata una soluzione alternativa basata su alcune idee computazionali fondamentali, sempre in linguaggio BASIC V2, che forniva una prestazione a runtime doppiamente migliore. Nel medesimo periodo, senza modificare minimamente il progetto, si è implementata la stessa soluzione in COMAL 80, riuscendo ad ottenere un tempo di esecuzione di circa **cinque volte** inferiore rispetto alla soluzione di partenza: risultato decisamente notevole per un linguaggio interpretato al pari del BASIC. Tali soluzioni originali, così come presenti negli archivi dell'Autore, sono state qui presentate senza modifiche sostanziali. Naturalmente sono possibili ulteriori miglioramenti, in special modo ricorrendo all'Assembly 6502/6510, e si invitano i lettori a far pervenire le loro implementazioni originali: il nostro intento qui è principalmente quello di mostrare la potenza e l'espressività di COMAL, usando come pretesto dei problemi-giocattolo.

Capitolo 8

«Cielo... mio marito!».

8.1 Introduzione.

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.

Sfatiamo subito un mito: scorrendo l'ormai vastissima lista delle applicazioni nate attorno a questo problema e sue variazioni ([Man13], pagg. $30 \div 31$), 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.

8.2 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:

$$H_n = \sum_{k=1}^{n} k^{-1} = \gamma + \psi_0(n+1) \tag{8.2.1}$$

Dove H_n è appunto l'ennesimo numero armonico[1]. Detto questo, il numero medio di proposte nell'algoritmo GS è dato da $n \cdot H_n + O((logn)^4)$. Ne consegue che la complessità media è dell'ordine di $\Theta(nlogn)$.

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:

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

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 lungi 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.

8.3 Gli anni Settanta e Ottanta.

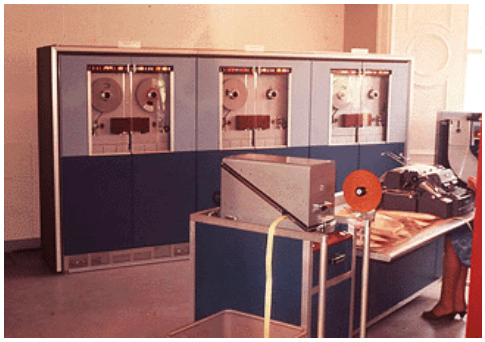


Figura 8.3.1: Il KDF-9 EEC a Newcastle

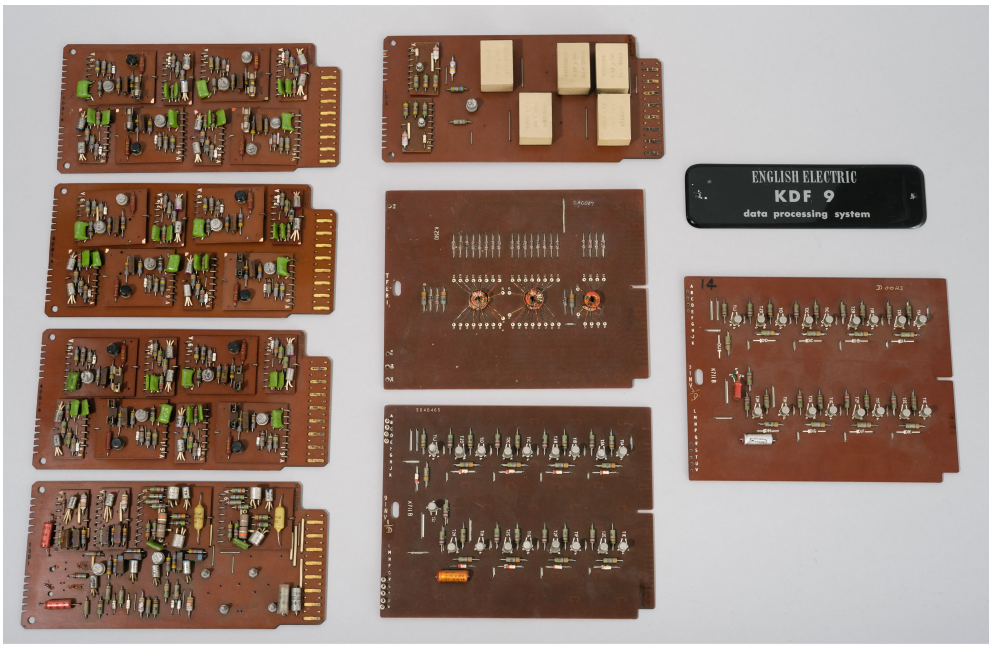
Pochi anni dopo la pubblicazione dell'articolo di Gale & Shapley, McVitie e Wilson nel 1970/71 [MW70, MW71b, MW71a] presentano vari algoritmi in ALGOL60, il più importante dei quali consente la generazione esaustiva di tutte le soluzioni stabili ad una data istanza di SM (per la versione del problema 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

[1] Nell'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.

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

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 hanno proposto l'algoritmo ancora attualmente in uso per la generazione esaustiva anche su istanze di grandi dimensioni, con prestazioni[2] in $O(n^2 + n \cdot |TSM|)$, 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.



2010/1/115
23/05/2013

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.

[2] Tale rivoluzionario incremento prestazionale è stato possibile grazie ad un attento e innovativo studio della natura algebrica dello spazio delle soluzioni come reticolo parzialmente ordinato e come anello d'insiemi, ed a margine grazie all'uso accorto di strutture dati *order-preserving* in grado di garantire contemporaneamente sia l'accesso diretto indicizzato, sia cancellazioni in $O(1)$. Tutto ciò rende possibile una pre-elaborazione in $O(n^2)$ e una fase finale di generazione esaustiva che richiede, piuttosto ovviamente, $O(n \cdot |TSM|)$, ovvero il prodotto tra il numero dei proponenti (dimensione dell'istanza) e il numero complessivo di soluzioni stabili esistenti.

8.4 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. $9 \div 10$).

```
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.

8.5 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 $w \in W$e 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' prolisso (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.

8.6 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.

Capitolo 9

«A scuola in scooter»... con COMAL.

9.1 Introduzione.

Abbiamo scelto un semplice problema logico, pubblicato come «2199ª prova d'intelligenza» sulla Settimana Enigmistica, per proporre una completa soluzione in linguaggio COMAL 80 (versione 2.01 per C64). Approfittiamo dell'occasione anche per un veloce ripasso su concetti e algoritmi fondamentali per la generazione di oggetti combinatori elementari.

Il presente articolo si accoda, con qualche variazione, alla fortunata serie tratta da testi storici di puzzle e rompicapo logici per C64, che pare avere conquistato qualche apprezzamento tra i lettori più assidui. Il problema, in questa occasione, è preso direttamente dalla più nota e longeva rivista di enigmistica italiana: la soluzione vuole mettere alla prova il lettore grazie alle peculiarità del linguaggio COMAL, del quale recentemente abbiamo tracciato l'origine storica e l'evoluzione. Si esortano i lettori, come in passato, a proporre proprie soluzioni originali in BASIC V2 e in Assembly 6510.

Il linguaggio dell'intera serie di articoli su puzzle e affini, lo ripetiamo ogni volta a beneficio dei nuovi lettori, è sempre tenuto intenzionalmente informale, divulgativo e didascalico, scevro da particolari pretese di rigore, al fine di raggiungere il più vasto uditorio (retro)informatico possibile. Nel medesimo spirito, anche a rischio di annoiare qualche lettore più evoluto, si cerca in ogni puntata di squadernare in modo pedissequo quell'idea computazionale, quella strategia algoritmica, quella peculiare struttura dati o quel particolare approccio di problem solving che di volta in volta rendono la soluzione interessante, didatticamente valida e ragionevolmente efficiente, pur mantenendone la natura illustrativa e spesso rinunciando ad ulteriori ottimizzazioni più strettamente legate all'architettura.

9.2 A scuola in scooter.

Riportiamo integralmente il testo del quesito originale, pubblicato come «2199ª prova d'intelligenza» sulla Settimana Enigmistica n° 4533 datata 7 febbraio 2019.

> I quattro amici Aldo, Bruno, Carlo e Daniele abitano tutti nello stesso palazzo e ognuno di loro, dal lunedì al sabato, porta a scuola con lo scooter la propria sorella.
> In una settimana un po' particolare succede che nessuno accompagni mai la propria sorella: ognuno porta infatti a scuola le sorelle dei tre amici, ciascuna per due volte. In nessuno dei sei giorni si ripete la medesima formazione di coppie di un altro giorno, e in nessun giorno succede che vi sia un ragazzo che accompagna la sorella dell'amico che accompagna la sua: ad esempio, nei due giorni in cui Aldo accompagna la sorella di Bruno, sullo scooter con Bruno non vi è la sorella di Aldo.
> La sorella di Carlo va a scuola con Aldo mercoledì e sabato. Lunedì e giovedì Daniele ha con sé la sorella dell'amico che martedì accompagna sua sorella. Lunedì e sabato è con Carlo la ragazza che mercoledì è con Daniele.
> **In quali giorni Bruno accompagna a scuola la sorella di Carlo?**

9.3 Alcune utili definizioni combinatorie elementari.

A beneficio dei lettori più giovani inseriamo qualche breve richiamo informale di combinatorica elementare che aiuterà a leggere le condizioni del problema, espresse in linguaggio naturale, nella loro reale veste di vincoli matematici che limitano lo spazio delle possibili permutazioni, garantendo l'unicità della soluzione. Il lettore che abbia già familiarità con la materia può tranquillamente saltare al paragrafo 9.4.

Ovviamente entriamo in punta di piedi nel meraviglioso reame della Matematica Discreta, e qui gli unici «numeri» di cui si parla *per default* sono sempre e soltanto naturali ossia *numeri interi non negativi*, insiemi e intervalli sono sempre e unicamente *finiti* e di conseguenza i relativi indici sono sempre implicitamente *opportuni*.

Dato un insieme, ovviamente finito e non vuoto, 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 banale formula che esprime il numero di permutazioni (http://oeis.org/A000142) in funzione dell'ampiezza data dall'intero non negativo n, ovvero equivalentemente l'ordine del gruppo simmetrico S_n, è ben nota e basata sul fattoriale:

$$P(n) = n! := \begin{cases} 1 & \text{per } n \in \{0,1\} \\ \prod_{j=1}^{n} j & \text{per } n \in \mathbb{N}\backslash\{0,1\} \end{cases} \tag{9.3.1}$$

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* nella quale la prima linea è costante e contiene la cosiddetta *permutazione identità*, ovvero gli ordinali interi positivi che identificano le posizioni.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 1 & 7 & 6 & 4 & 5 & 8 & 3 \end{pmatrix} \tag{9.3.2}$$

Tale notazione si legge intuitivamente per colonne, ad esempio da sinistra a destra, e dal basso verso l'alto: usando la notazione $a \leftarrow b$ per indicare che il numero b nella seconda riga «prende il posto» di a sovrastante, avremo quindi $1 \leftarrow 2, 2 \leftarrow 1, 3 \leftarrow 7, 4 \leftarrow 6, 5 \leftarrow 4, 6 \leftarrow 5, 7 \leftarrow 8, 8 \leftarrow 3$ e l'ovvio significato 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. Equivalentemente, usando i pedici ed indicando ogni elemento come a_i, la notazione può essere letta come $a_1 = 2$, $a_2 = 1, \cdots$, $a_8 = 3$. Si sottolinea come tale notazione risulti estremamente espressiva e flessibile.

Dalla notazione a due linee appena esposta risulta immediato definire anche il concetto di cicli disgiunti. Riordiniamo le assegnazioni dell'esempio (9.3.2) come segue: $\underline{1} \leftarrow 2, 2 \leftarrow \underline{1}$; $\underline{3} \leftarrow 7, 7 \leftarrow 8, 8 \leftarrow \underline{3}$; $\underline{4} \leftarrow 6, 6 \leftarrow 5, 5 \leftarrow \underline{4}$. Si nota facilmente come le assegnazioni tornano ciclicamente ad un valore di partenza, 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 ordinati in modo crescente in base a tale valore. Applicando tale convenzione, la permutazione (9.3.2) 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 $\underline{2}1\underline{6}54\underline{8}37$ 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: $(\underline{2}1\underline{6}54\underline{8}37)$. 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: $(\underline{2}1)(\underline{6}54)(\underline{8}37)$. Ecco così facilmente ricostruita la scrittura originale[1].

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 $\left[\begin{matrix} n \\ k \end{matrix} \right] = s(n, k)$ 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]).

[1] Allo 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).

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 (9.3.2), si può quindi scrivere in vari modi la medesima permutazione:

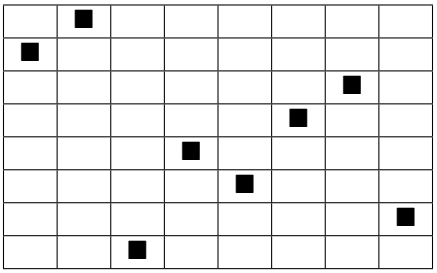
$$\langle 2,1,7,6,4,5,8,3\rangle$$
$$2,1,7,6,4,5,8,3$$
$$2\ 1\ 7\ 6\ 4\ 5\ 8\ 3$$
$$21764583$$

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.

Un altro diffuso metodo utilizzato per la rappresentazione di permutazioni consiste nell'usare una matrice quadrata $n \times n$ contenente unicamente valori dall'insieme $\{0,1\}$, ovvero una matrice binaria, in modo tale che per ciascuna riga e per ciascuna colonna vi sia un unico valore unitario. Consideriamo ancora la permutazione (9.3.2). La sua rappresentazione matriciale, convenendo di assegnare le *righe* alla permutazione identità, si scriverà come segue:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{9.3.3}$$

Si presti attenzione al fatto che (naturalmente!) esistono almeno due convenzioni discordanti sulla lettura di tale matrice *sparsa* (su un totale di n^2 locazioni contiene unicamente n valori pari a 1, uno solo per ogni data riga e colonna), 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 $1 \leftarrow 2$, $2 \leftarrow 1$, $3 \leftarrow 7$, $4 \leftarrow 6, \ldots$ poiché si ha, rispettivamente, $M_{1,2} = 1$, $M_{2,1} = 1$, $M_{3,7} = 1$, $M_{4,6} = 1, \ldots$ e più in generale, $R \leftarrow C$ 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, che copre un'area molto vasta (rook polynomials) e meriterebbe certamente ben altri spazi di approfondimento.

Tornando invece al problema specifico, chiamiamo *derangement* una particolare permutazione nella quale nessun elemento rimane al proprio posto di partenza, ovvero una permutazione priva di punti fissi (oppure, ciò che fa lo stesso, priva di cicli di lunghezza unitaria). 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 $a_3a_2a_1\underline{a_4}$ non lo è, perché (almeno) un elemento a_i occupa la $i-esima$ posizione (in questo banale esempio, $i = 4$, come evidenziato), ove opportunamente $i \in [1, n]$. Si noti incidentalmente che la permutazione (9.3.2), in modo assolutamente non casuale, è anche un derangement e che pertanto la relativa matrice binaria ha la diagonale principale nulla.

Il numero totale dei derangement (http://oeis.org/A000166) per una data ampiezza n si indica generalmente con $D(n)$, ed è definito un apposito operatore «cofattoriale» per indicare tale quantità:

$$D(n) =!n := n!\cdot \sum_{k=0}^{n} \left[(k!)^{-1}(-1)^k\right] \tag{9.3.4}$$

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

$$D(n) = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor \quad \text{per } n \geq 1 \tag{9.3.5}$$

Come immediata conseguenza, abbiamo che il rapporto tra $D(n)$ e il numero totale di permutazioni $P(n)$ della (9.3.1) può scriversi:

$$\frac{D(n)}{P(n)} \approx e^{-1} \tag{9.3.6}$$

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

9.4 Uno sguardo alle proprietà matematiche della nostra soluzione.

Dal punto di vista più prettamente combinatorio la soluzione al problema è data da un particolare tipo di *block design completo con ordinamento* con 4 *varietà* e 6 blocchi. Si tratta di un oggetto combinatorio non comune e non propriamente elementare, la cui ampissima e ben studiata famiglia (che include tra gli altri BIBD, codici a correzione d'errore, lotto designs, schemi d'esperimento random, quadrati latini, matrici ortogonali, MOLS e molto altro) risulta fondamentale in innumerevoli settori: geometria finita, progetto e analisi di algoritmi e di reti, verifica esaustiva di software e hardware, reliability engineering, crittografia, tornei e lotterie, esperimenti incrociati sistematici di ogni genere nelle scienze dell'uomo e naturali, biologia molecolare, animale e vegetale, agronomia e molto altro.

Design combinatori. La «data di nascita» ufficiale di questa vasta famiglia di oggetti matematici è tradizionalmente collocata tra gli anni Venti e Trenta dello scorso secolo, associata al lavoro di una coppia di statistici britannici che studiavano esperimenti combinatori per l'agronomia: Sir Ronald A. Fisher (1890 – 1962) e Frank Yates (1902 – 1994), autori dell'arcinoto testo [Fis92] la cui prima edizione risale al 1925.
Per chiarire meglio alcuni acronimi e termini appena citati:

- BIBD è l'acronimo di Balanced Incomplete Block Design.
- Un quadrato latino è un array quadrato $n \times n$ contenente n simboli, ciascuno dei quali compare esattamente una volta per ogni riga e per ogni colonna. Ecco ad esempio un tipico *latin square* di lato 3:
  $$A = \begin{array}{ccc} 2 & 1 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \end{array}.$$
- Due quadrati latini di ordine n formano una coppia MOLS (Mutually Orthogonal Latin Squares) quando, se sovrapposti, ciascuna tra le possibili n^2 coppie ordinate appare esattamente una volta. Ad esempio, la coppia di quadrati latini $A = \begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{array}$, $B = \begin{array}{ccc} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{array}$ costituisce una coppia MOLS $AB = \begin{array}{ccc} 11 & 22 & 33 \\ 23 & 31 & 12 \\ 32 & 13 & 21 \end{array}$ nella quale compaiono con ogni evidenza tutte le $3^2 = 9$ possibili coppie ordinate del prodotto cartesiano $\{1,2,3\} \times \{1,2,3\} = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\}$.

Data la vastità del campo applicativo e l'enorme importanza dell'argomento, l'Autore confida che non mancherà occasione di approfondire questi multiformi e sofisticati oggetti combinatori, nonostante l'elevata complessità computazionale che medialmente caratterizza i relativi algoritmi generatori ed enumerativi, rendendone ardua e sovente proibitiva l'implementazione in ambito retrocomputing anche per dimensioni assai modeste dei problemi.
Per il momento, il lettore interessato è caldamente invitato a fare riferimento innanzi tutto a [Sti03, Wal07, Lin09] per una veloce introduzione con alcuni cenni agli aspetti algoritmici e implementativi, passando poi a [CD06] e [BJL99, BJL10] per uno sguardo d'insieme omnicomprensivo all'intero settore di ricerca.

In questa sede, la volontà di mantenere l'approccio il più semplice e comprensibile possibile ci induce invece a fingere di chiudere un occhio sulla reale natura di tale oggetto matematico, scendendo ad un livello di astrazione minore e considerandone solo le componenti combinatorie elementari. Alla luce delle sole definizioni telegraficamente richiamate al paragrafo 9.3, possiamo riconsiderare i vincoli del problema sotto una nuova prospettiva: si tratta di una particolare permutazione i cui elementi sono in realtà sei derangement ristretti da 4 simboli ciascuno. Sebbene esistano solo $6! = 720$ di tali permutazioni, un approccio *bruteforce* sarebbe ovviamente fuori discussione per lapalissiane ragioni didattiche e di principio.

Usando per brevità le sole iniziali A, B, C, D dei protagonisti, codifichiamo con le corrispondenti lettere minuscole a, b, c, d le sorelle. Si noti che non occorre esplicitare l'accoppiamento, perché una volta fissato un ordine (quello lessicografico ascendente) per le iniziali dei fratelli, la sola posizione del simbolo in minuscolo è già sufficiente a determinare la coppia, come richiamato nel seguente esempio che utilizza sia una variazione della notazione a due linee 9.3.2 che la corrispondente *1-line*:

$$\begin{pmatrix} A & B & C & D \\ d & c & a & b \end{pmatrix} \longrightarrow d\,c\,a\,b$$

Dunque, senza perdita di generalità, scrivendo sinteticamente $d\,c\,a\,b$ intendiamo riferirci implicitamente ma in modo non ambiguo all'elenco di coppie (A,d), (B,c), (C,a), (D,b). Rivediamo ora sinteticamente i vincoli espressi nel testo del problema:

1. «In una settimana un po' particolare succede che nessuno accompagni mai la propria sorella».
   Questo significa semplicemente che dobbiamo prendere in considerazione soltanto i derangement di quattro simboli, che assommano a $D(4) = 9$ e sono oggetti combinatori banalissimi da generare efficientemente (e da filtrare per applicarvi ulteriori condizioni).

2. «ognuno porta [...] a scuola le sorelle dei tre amici, ciascuna per due volte».
   Leggendo la soluzione per colonne, questo implica che vedremo comparire - per due volte ciascuna - tre delle quattro ragazze, mentre il vincolo 1 ci ha già confermato che apparirà zero volte la sorella del giovane a cui corrisponde detta colonna. Ad esempio, considerando la colonna B, troveremo ripetuti due volte ciascuno i simboli a, c, d corrispondenti alle sorelle di Aldo, Carlo e Daniele.

3. «In nessuno dei sei giorni si ripete la medesima formazione di coppie di un altro giorno».
   Ciò indica che la soluzione comprende esattamente sei derangement distinti, sui nove possibili. Le ulteriori restrizioni imposte dal testo provvedono appunto ad eliminare dal totale alcuni di tali derangement: in particolare, si veda il punto successivo.

4. «in nessun giorno succede che vi sia un ragazzo che accompagna la sorella dell'amico che accompagna la sua».
   Per mera comodità referenziale indichiamo con s_i (dove ovviamente vale $i = 1, 2, \ldots$) i simboli $\{a, b, c, d\}$ utilizzati per le sorelle e chiamiamo p_i le rispettive posizioni di tali simboli nel derangement, associate biunivocamente alle iniziali dei fratelli $\{A, B, C, D\}$. Il presente vincolo si traduce allora come segue: nel formare il derangement, abbiamo inizialmente tre sole possibilità per assegnare un simbolo s_i alla posizione p_i, dove per la definizione di derangement si ha $p_i \neq i$. Una volta effettuata tale scelta, diciamo per fissare le idee $p_i = k$, viene però esclusa anche la possibilità di effettuare l'associazione complementare di s_k alla posizione $p_k = i$. Se ad esempio abbiamo effettuato l'abbinamento di $s_3 = c$ con la posizione A ovvero $p_3 = 1$, non potremo avere nel medesimo derangement anche $s_1 = a$ in posizione $p_1 = 3$ e questo per ogni possibile valore di i e k. In altri termini possiamo anche dire che, una volta scelta una coppia Xy, non potrà essere selezionata nella medesima giornata la coppia complementare Yx, o - ciò che fa lo stesso - che (ignorando sia maiuscole e minuscole che l'ordine degli elementi) non possono aversi coppie sostanzialmente duplicate di tipo xy, xy.

Ma cosa significa effettivamente il vincolo 4, dal punto di vista combinatorio? Lo vediamo con estrema chiarezza scrivendo $c\,d\,a\,b$, uno dei derangement «proibiti», in forma matriciale binaria, come ricordato in formula 9.3.3:

	a	b	c	d
A	0	0	1	0
B	0	0	0	1
C	1	0	0	0
D	0	1	0	0

In breve, il vincolo considerato impone che la matrice binaria, per come l'abbiamo orientata, contenga uno e un solo valore unitario per ogni *antidiagonale*[2]. Seguendo la normale convenzione per indici r, c crescenti

[2]Secondo le convenzioni dell'algebra lineare e matriciale sono dette *antidiagonali* quelle che vanno, informalmente, da sinistra in basso a destra e in alto. Le antidiagonali sono caratterizzate dall'avere somma costante degli indici per ciascuna delle loro celle, come è reso evidente dal seguente esempio nel quale ogni cella contiene esplicitata la somma delle proprie coordinate secondo lo standard algebrico $a_{r,c}$ (riga, colonna) nel quale, come noto, la prima cella in alto a sinistra è contrassegnata da indici unitari $a_{1,1}$. Si noti come, leggendo ad esempio lungo l'antidiagonale principale, $4+1 = 3+2 = 2+3 = 1+4$.

$1+1$	$1+2$	$1+3$	$1+4$
$2+1$	$2+2$	$2+3$	$2+4$
$3+1$	$3+2$	$3+3$	$3+4$
$4+1$	$4+2$	$4+3$	$4+4$

verso destra e verso il basso, si è evidenziata in colore rosso la prima antidiagonale che viola tale imposizione (in realtà, data la limitatissima ampiezza dei derangement considerati, avviene che per simmetria gli scambi proibiti si presentino a coppie in tutti e tre i casi notevoli). Per mera pedanteria didattica si sottolinea come gli altri vincoli a monte impongono rispettivamente uno e un solo valore unitario per riga e colonna (per la definizione di permutazione) e diagonale principale nulla (perché si tratta di derangement): usando il principio di inclusione/esclusione è banale mostrare che sotto tali condizioni esistono solamente 6 matrici siffatte di dimensione pari a 4×4.

A questo punto, non è certo difficile preselezionare i sei derangement che soddisfano i vincoli del problema. Elenchiamo in ordine lessicografico ascendente tutti i derangement alfabetici di ampiezza pari a 4 ed eliminiamo quelli che presentano (almeno) uno scambio «proibito», contrassegnando gli altri con un numero intero positivo:

×	b	a	d	c	(ba)(dc)
1	b	c	d	a	(dabc)
2	b	d	a	c	(dcab)
3	c	a	d	b	(dbac)
×	c	d	a	b	(ca)(db)
4	c	d	b	a	(dacb)
5	d	a	b	c	(dcba)
6	d	c	a	b	(dbca)
×	d	c	b	a	(cb)(da)

Accanto a ciascun derangement è riportata l'equivalente notazione ciclica secondo Stanley: risulta così immediato notare come i derangement scartati sono, in questo caso, tutti e soli quelli caratterizzati dall'avere due cicli. Il che, pur non essendo computazionalmente molto efficiente da rilevare, riveste comunque interesse dal punto di vista combinatorio e compendia quanto già detto in merito alla matrice binaria. Si noti altresì come, leggendo l'elenco per colonne, sia rispettato il vincolo 2 dopo l'eliminazione di quei derangement con configurazioni proibite dal vincolo 4. Il resto della soluzione consiste poi, senza sorprese, nell'applicare un ordinamento ai sei derangement selezionati tale da soddisfare i vincoli minori e le ulteriori indicazioni del testo del problema.

Vediamo brevemente come applicare tali ulteriori vincoli all'elenco già filtrato dei derangement, il quale semplifica notevolmente il lavoro:

5. La sorella di Carlo va a scuola con Aldo mercoledì e sabato.

6. Lunedì e giovedì Daniele ha con sé la sorella dell'amico che martedì accompagna sua sorella.

7. Lunedì e sabato è con Carlo la ragazza che mercoledì è con Daniele.

Il vincolo 5 ci dice semplicemente che dobbiamo assegnare al mercoledì e al sabato i due derangement 3 e 4, ma non ci consente ancora di determinare in che ordine. Il vincolo 7, che in realtà va applicato per secondo, ci fornisce informazioni su ben tre assegnazioni dei derangement al rispettivo giorno della settimana. Indichiamo con x la lettera incognita:

lun	-	-	x	-
mer	c	-	-	x
sab	c	-	x	-

A prima vista, x potrebbe essere sostituita da a, b oppure d, essendo c esclusa a priori. Dovendo automatizzare la soluzione, si potrebbe essere tentati di provare esaustivamente tutte le possibilità, data anche la modestissima estensione dello spazio delle soluzioni. Ma il computing degli anni Ottanta, con l'inerente limitatezza delle risorse computazionali, ci ha insegnato ad aguzzare l'ingegno e ad evitare il più possibile il *bruteforce*. Cosa ci dicono in realtà questi tre template? Quello del mercoledì indica chiaramente che $x \neq d$, per il vincolo 1 (derangements). Allo stesso modo, il template del sabato ci dice anche che $x \neq a$, per il vincolo 4. Ne consegue, per esclusione, $x = b$. Questo scioglie ogni residua ambiguità del vincolo 5. Si ricordi inoltre che in ogni colonna eccetto la B, e segnatamente nella colonna C, vi sono unicamente due b, una delle quali appartiene al derangement 4 appena assegnato al sabato. Quindi anche l'assegnazione del lunedì è meccanicamente determinata, col derangement 5, l'unico altro che contenga la b in terza posizione. In definitiva, abbiamo le tre assegnazioni in tabella seguente:

lun	d	a	b	c	(5)
mer	c	a	d	b	(3)
sab	c	d	b	a	(4)

Come da tradizione in questa serie di articoli, decidiamo di sfruttare queste semplici deduzioni iniziali per semplificare la stesura del software, usandole in modo hardcoded per evitare ricerche esaustive o altri approcci. Non ci interessa infatti un risolutore generalizzabile, magari basato su ricorsione e backtracking, quanto piuttosto evidenziare la «traduzione» in COMAL dei vincoli e delle regole (quindi della specifica) restando focalizzati sull'istanza del problema-giocattolo proposto.

Rimangono a questo punto da assegnare i derangement 1, 2 e 6. Il vincolo 6 ci fornisce in realtà due distinte indicazioni, sufficienti a completare lo schema.

a. Il derangement da assegnare al giovedì ha la c in quarta posizione, come quello assegnato al lunedì, ovvero il 5. Tra i rimanenti, non può che essere il 2.

b. Il derangement da assegnare al martedì ha la d nella colonna (fratello) corrispondente al quarto simbolo dei derangement 5 e 2, ovvero c, al cui fratello corrisponde la terza posizione nel derangement. Dei due rimanenti, solo l'1 ha tale proprietà.

Per esclusione, il derangement 6 è assegnato infine al venerdì. Ne risulta la seguente soluzione completa, nella quale, per rispondere alla domanda del quesito, Bruno (colonna 2) accompagna a scuola c, sorella di Carlo, il **martedì** e il **venerdì**:

	A	B	C	D	
lun	d	a	b	c	(5)
mar	b	c	d	a	(1)
mer	c	a	d	b	(3)
gio	b	d	a	c	(2)
ven	d	c	a	b	(6)
sab	c	d	b	a	(4)

9.5 L'algoritmo generatore utilizzato.

Per la generazione dei derangements si è scelto di impiegare l'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 penalità prestazionale nella sua implementazione in COMAL. Tuttavia l'assoluta 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. Tali caratteristiche, apprezzabili fin dalla primissima occhiata allo pseudocodice, dovrebbero già da sole 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 quasi tre 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 $0 \le i < n$.

```
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. Analizziamo la $k-esima$ chiamata, con $0 < k < n$: quando viene chiamata `DERANGE2(k)`, gli elementi $D[k+1] \cdots D[n-1]$ 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 non produca una permutazione che viola il vincolo fondamentale del derangement. Questa costruzione a ritroso è anche il motivo fondamentale per cui la generazione non avviene nel più comune ordine lessicografico.

Da notare il caratteristico procedimento con *backtracking*, che al termine del loop annulla lo scambio effettuato prima della chiamata ricorsiva: un idioma assolutamente tipico di una intera classe di algoritmi generativi esaustivi.

Analizzando la $k-esima$ chiamata, è molto facile calcolare che il numero totale di chiamate ricorsive per il livello successivo $k+1$ è dato da:

$$d_{n,k} = \begin{cases} (n-1)\cdot d_{n-1,k} + (n-k-1)\cdot d_{n-2,k} & \text{per } 1 \leqslant k < n \\ 1 & \text{per } k = n \\ 0 & \text{per } k > n \end{cases} \tag{9.5.1}$$

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 (9.5.1):

$$T_{Akl} = \sum_{k=1}^{n} [(n-1)d_{n-1,k} + (n-k-1)d_{n-2,k}] \tag{9.5.2}$$

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

$$T_{Akl} = \sum_{k=1}^{n}\sum_{r=0}^{n-k} \frac{(-1)^r}{k!}\binom{n-k}{r}(n-r)! \approx (1-e^{-1})n! \cong 0,63212 \cdot n! \tag{9.5.3}$$

In questo caso 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.

9.6 La soluzione in COMAL 80.

Si presenta una soluzione completa in COMAL 80 (versione 2.01) per C64.

```
//*******************************
//** save "Scooter"             **
//*******************************

DATA "Lun","Mar","Mer","Gio","Ven","Sab"
DIM a(4)          // Array ausiliario
DIM der(6,5)      // Derangements selezionati
soln:=1           // Totalizzatore soluzioni
total:=1          // Totale derangements
DIM sol(6)        // Array soluzioni
DIM dow$(6) OF 3 // Giorni della settimana

//*******************************
//** Procedura principale      **
//*******************************
PAGE
FOR i:=1 TO 6 DO der(i,5):=0
FOR i:=1 TO 4 DO a(i):=i
i:=1
WHILE NOT EOD DO
  READ dow$(i)
  i:+1
ENDWHILE

PRINT "Generazione derangements........"
PRINT "Convalida derangements..........."
PRINT "Applicazione delle regole......."

derange2(a(),4)

//** Vincoli 5 e 7 **
FOR i:=1 TO 2 DO
```

```
  PRINT AT 3,33: i
  posn:=lfind(3,1)
  IF der(posn,3)=2 THEN
    //** Sabato
    der(posn,5):=6
    sol(6):=posn
  ELSE
    //** Mercoledi'
    der(posn,5):=3
    sol(3):=posn
  ENDIF
ENDFOR i

//** Lunedi'
PRINT AT 3,33: "3"
posn:=lfind(2,3)
der(posn,5):=1
sol(1):=posn

//** Vincolo 6 **
//** Giovedi'
PRINT AT 3,33: "4"
posn:=lfind(der(sol(1),4),4)
der(posn,5):=4
sol(4):=posn

//** Martedi'
PRINT AT 3,33: "5"
posn:=lfind(4,der(sol(1),4))
der(posn,5):=2
sol(2):=posn

//** Venerdi'
PRINT AT 3,33: "6";CHR$(186)
posn:=lfind(0,5)
der(posn,5):=5
sol(5):=posn

display'sol
PRINT
PRINT "****SOLUZIONE****"
PRINT "Bruno accompagna a scuola la sorella di"
PRINT "Carlo nei giorni:";
FOR i:=1 TO 6 DO
  IF der(i,2)=3 THEN
    PRINT dow$(der(i,5));
  ENDIF
ENDFOR i
PRINT
END "** Fine lavoro **"

//*******************************
//** Subroutine                **
//*******************************
PROC derange2(d(),m)
  //**
  //** Algoritmo di S.G. Akl
  //** per i derangements.
  //**
  IF m>1 THEN
    le:=m
```

```
    IF d(m)=m THEN le:-1
    FOR j:=le TO 1 STEP -1 DO
      t:=d(m)
      d(m):=d(j)
      d(j):=t
      derange2(d(),m-1)
      t:=d(m)
      d(m):=d(j)
      d(j):=t
    ENDFOR j
  ELIF d(1)<>1 THEN
    PRINT AT 1,33: total
    IF total=9 THEN PRINT AT 1,35: CHR$(186)
    total:+1
    IF check'sol(d())=TRUE THEN
      FOR i:=1 TO 4 DO
        der(soln,i):=d(i)
      ENDFOR i
      soln:+1
    ENDIF
  ENDIF
ENDPROC derange2

FUNC check'sol(d())
  //**
  //** Convalida un derangement
  //** secondo il vincolo 4.
  //**
  PRINT AT 2,33: soln
  IF soln=6 THEN PRINT AT 2,35: CHR$(186)
  FOR i:=2 TO 4 DO
    IF d(1)=i AND d(i)=1 THEN RETURN FALSE
  ENDFOR i
  RETURN TRUE
ENDFUNC check'sol

FUNC lfind(va,pos)
  //**
  //** LFIND() cerca per valore e
  //** posizione tra le soluzioni.
  //** Restituisce 0 o l'indice
  //** del derangement cercato.
  //**
  FOR i:=1 TO 6 DO
    IF der(i,pos)=va AND der(i,5)=FALSE THEN RETURN i
  ENDFOR i
  RETURN 0
ENDFUNC lfind

PROC display'der(p)
  //**
  //** Stampa derangement come
  //** sequenza di lettere.
  //**
  FOR i:=1 TO 4 DO
    PRINT CHR$(64+der(p,i));
  ENDFOR i
  PRINT
ENDPROC display'der

PROC display'sol
```

```
  //**
  //** Visualizza schema soluzione.
  //**
  PRINT
  PRINT "     A B C D"
  FOR i:=1 TO 6 DO
    PRINT dow$(i);
    display'der(sol(i))
  ENDFOR i
ENDPROC display'sol
```

Se questa è la prima volta che il lettore si trova di fronte ad un listato COMAL, noterà un numero rilevante di differenze rispetto al CBM BASIC V2. I numeri di riga sono stati soppressi usando l'istruzione `DISPLAY`, che genera un listato in tutto e per tutto simile ai moderni HLL per PC e affini. Per limitarci a riassumere gli aspetti più macroscopici: il codice è indentato automaticamente, il linguaggio ammette una singola istruzione per riga, vi sono costrutti condizionali e di loop avanzati in stile ALGOL o Pascal, sono presenti nomi lunghi per variabili, procedure, funzioni. L'operatore di assegnazione := è distinto da quello di confronto =, c'è una sintassi specifica per il passaggio di array come parametri. Nell'insieme il linguaggio è un po' prolisso (come ALGOL, COBOL o FORTRAN) ma estremamente chiaro, leggibile, ordinato. Perfetto per l'uso didattico ma anche nel mondo professionale: infatti e non a caso COMAL ha un lungo e glorioso stato di servizio nel mercato gestionale e non solo, fino alla seconda metà degli anni Novanta, su una varietà di piattaforme che va dal C64 agli Atari, fino ai Vax 11.

La soluzione del nostro simpatico problemino è basata quasi interamente sullo sfruttamento delle proprietà combinatorie degli oggetti interessati e sulla scelta di una opportuna coppia di strutture dati che congiuntamente consentono anche il reverse lookop e l'ordinamento in-place, senza scomodare algoritmi di ordinamento. L'array `der(6,5)` contiene infatti i sei derangements selezionati, un valore per ognuna delle prime quattro locazioni colonna, mentre la quinta cella di ciascuna riga contiene inizialmente uno zero e successivamente il giorno della settimana al quale il derangement relativo viene assegnato. Ricordando che in COMAL l'indicizzazione parte da 1, si è scelto di memorizzare i valori sotto forma di piccoli interi nell'intervallo $[1, 4]$ per la massima praticità ed efficienza: in questo modo, infatti, si ottiene la massima intercambiabilità tra valore e posizione, senza ricorrere a dispendiose operazioni di conversione a base di `ORD()` e `CHR$()`. L'array abbinato `sol(6)` contiene, intuitivamente, l'indice di `der()` corrispondente al derangement assegnato al giorno della settimana $1 \cdots 6$. I due array così abbinati consentono quindi direct e reverse lookup secondo le necessità, al prezzo ottimale di due scritture in memoria per ogni assegnazione effettuata, come risulta palese dalle coppie di assegnazioni nella parte finale del codice.

Il codice è sostanzialmente autodocumentante e organizzato secondo la massima modularità. I derangement vengono generati col noto algoritmo, poi selezionati sfruttando una delle varie caratteristiche evidenziate dal vincolo 4, e infine le assegnazioni ai giorni della settimana vengono eseguite in modo pedissequo e scolastico applicando le immediate conseguenze logiche dei vincoli 5, 7 e 6 in questo preciso ordine, il che si traduce di

volta in volta nel cercare tramite l'apposita funzione lineare `LFIND()` quel derangement ancora non selezionato (da qui l'importanza della quinta colonna dell'array `der()`) che abbia un dato valore in una certa posizione, se necessario ricavata per via indiretta da altro derangement. Si noti che, grazie alla ricerca lineare, possiamo rendere il programma sostanzialmente indipendente dall'ordine di generazione dei derangements, con un aggravio prestazionale pressoché insignificante.

9.7 La soluzione a pag. 46...

Per un veloce riscontro si propone integralmente la soluzione pubblicata sul medesimo numero della rivista. Si tratta di una soluzione *ad hoc*, per esclusioni e sostituzioni progressive basate sui vincoli: un approccio risolutivo classico, che non sfrutta esplicitamente le caratteristiche intrinseche degli oggetti combinatori coinvolti.

Bruno accompagna la sorella di Carlo martedì e venerdì.
Infatti, innanzi tutto si indichino i ragazzi con la rispettiva iniziale maiuscola e le sorelle con l'iniziale minuscola del nome del loro fratello, e in base ai dati si costruisca la seguente tabella nella quale in ogni riga di fianco ai ragazzi vanno inserite le ragazze che con loro formano, giorno per giorno, i quattro equipaggi (x e y sono da determinare):

Lunedì	A	B	Cy	Dx	
Martedì	A	B	C	D	(Xd)
Mercoledì	Ac	B	C	Dy	
Giovedì	A	B	C	Dx	
Venerdì	A	B	C	D	
Sabato	Ac	B	Cy	D	

Si noti che non può essere $y = c$ né $y = a$, perché sabato ci sono i due equipaggi Ac e Cy, e non esistono due coppie con le stesse lettere invertite; né può essere $y = d$ perché mercoledì c'è Dy. Per esclusione è $y = b$. A questo punto, tornando alla giornata di sabato, le altre due coppie sono Bd e Da e quindi, nella colonna di D, è facile trovare $x = c$. Dopo questa sostituzione, si può procedere facendo attenzione che ogni riga non contenga due volte la stessa coppia di lettere: mercoledì le coppie mancanti sono Ba e Cd; nei giorni di giovedì e venerdì con C può andare solo a, e quindi venerdì sullo scooter di D sale b mentre martedì è il turno di Da. Si possono ora completare le colonne di A e di B: la coppia Ab va il martedì e il giovedì, e la coppia Ad il lunedì e il venerdì; lunedì tocca a Ba, giovedì a Bd, mentre Bc salgono insieme martedì e venerdì.

Lunedì	Ad	Ba	Cb	Dc
Martedì	Ab	Bc	Cd	Da
Mercoledì	Ac	Ba	Cd	Db
Giovedì	Ab	Bd	Ca	Dc
Venerdì	Ad	Bc	Ca	Db
Sabato	Ac	Bd	Cb	Da

Dallo schema si evince la soluzione completa.

9.8 Conclusioni.

Si è presentato un semplice quesito logico-combinatorio tratto dalla Settimana Enigmistica, usandolo come ottimo pretesto per un veloce ripasso di definizioni e concetti elementari di combinatorica, per passare poi a proporre una rapida analisi del procedimento risolutivo. 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 del problema e alla evidenziazione delle proprietà degli oggetti combinatori elementari coinvolti nella soluzione. La speranza dell'Autore è che i lettori apprezzino così meglio tale linguaggio, sentendosi stimolati a proporre a loro volta soluzioni alternative come pure implementazioni nei linguaggi tradizionali BASIC V2 e Assembly 6502/6510.

Bibliografia

[AH89] Kenneth Appel and Wolfgang Haken, *Every planar map is four colorable*, American Mathematical Society, 1989.

[Akl80] Selim G. Akl, *A new algorithm for generating derangements*, BIT **20** (1980), 2–7.

[And85] Mark Andrews, *Commodore 64/128 assembly language programming*, H.W. Sams, 1985.

[Ath82] Roy Atherton, *Structured programming with comal (ellis horwood series in computers and their applications)*, Ellis Horwood Ltd , Publisher, 1982.

[AW92] Karla Alwan and Kelly Waters, *Finding re-entrant knight's tours on n-by-m boards*, Proceedings of the 30th Annual Southeast Regional Conference (New York, NY, USA), ACM-SE 30, ACM, 1992, pp. 377–382.

[BJ66] Corrado Boehm and Giuseppe Jacopini, *Flow diagrams, turing machines and languages with only two formation rules*, Commun. ACM **9** (1966), no. 5, 366–371.

[BJL99] Thomas Beth, Deiter Jungnickel, and Hanfried Lenz, *Design theory: Volume 1 (encyclopedia of mathematics and its applications)*, Cambridge University Press, 1999.

[BJL10] Thomas Beth, Dieter Jungnickel, and Hanfried Lenz, *Design theory, volume ii*, Cambridge University Press, 2010.

[BP88] Marcus D. Bowman and Stephen Pople, *Computing studies in context: Comal programming guide*, Heinemann Educational Publishers, 1988.

[Bre85] Robert C Brenner, *The commodore 64 troubleshooting & repair guide*, H.W. Sams, 1985.

[CD06] Charles J. Colbourn and Jeffrey Dinitz, *Handbook of combinatorial designs (discrete mathematics and its applications)*, Chapman and Hall/CRC, 2006.

[CHMW94] Axel Conrad, Tanja Hindrichs, Hussein Morsy, and Ingo Wegener, *Solution of the knight's hamiltonian path problem on chessboards*, Discrete Applied Mathematics **50** (1994), no. 2, 125 – 134.

[Chr82] Borge Christensen, *Beginning comal*, Harwood & Charles Pub Co, 1982.

[Chr84] Borge R Christensen, *Comal reference guide*, Toronto PET users group, 1984.

[Dal84] Simon Dally, *Century/acorn user book of computer puzzles: v. 1 (the century/acorn user)*, Century, 1984.

[Dav84] Danny Davies, *Commodore 64 machine language for the absolute beginner*, Imprint unknown, 1984.

[Dij82] Edsger W. Dijkstra, *How do we tell truths that might hurt?*, SIGPLAN Not. **17** (1982), no. 5, 13–15.

[Dij87] Edsger W. Dijkstra, *Go to statement considered harmful*, Commun. ACM **30** (1987), no. 3, 195–196.

[DM18] Brendan D. Mckay, *Knight's tours on an 8x8 chessboard.*

[Eng85] Lothar Englisch, *The advanced machine language book for the commodore 64*, Abacus Software Inc, 1985.

[Fab84] Tony Fabbri, *Animation, games, and sound for the commodore 64*, Prentice-Hall, 1984.

[Fis92] R. A. Fisher, *Statistical methods for research workers*, Springer Series in Statistics, Springer New York, 1992, pp. 66–70.

[FW08] Daniel P. Friedman and Mitchell Wand, *Essentials of programming languages, 3rd ed.*, MIT Press, 2008.

[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.

[GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete mathematics: A foundation for computer science, 2nd ed.*, Addison-Wesley, Reading. MA, 1994.

[Gra85] Ingvar Gratte, *Starting with comal*, Simon & Schuster (Paper), 1985.

[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.

[HAB+15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller, *A formal proof of the Kepler conjecture*, ArXiv e-prints (2015), arXiv:1501.02155.

[Hee84] D. Heeb, *Vic and commodore 64 tool kit: Basic*, Compute!, 1984.

[Hee85] Dan Heeb, *Compute!'s vic-20 and commodore 64 tool kit: Kernal*, Compute, 1985.

[Hei83] John Heilborn, *Compute's reference guide to commodore 64 graphics*, Compute, 1983.

[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.

[Jon84] Marvin L. De Jong, *Assembly language programming with the commodore 64*, Brady, 1984.

[Kel84] John Kelly, *Foundations in computer studies with comal*, The Educational Company of Ireland, Ltd., 1984.

[Knu64] Donald E. Knuth, *Backus normal form vs. backus naur form*, Commun. ACM **7** (1964), no. 12, 735–736.

[Knu97] Donald E. Knuth, *Stable marriage and its relation to other combinatorial problems*, American Mathematical Society, Providence, R.I, 1997.

[Knu11] Donald E. Knuth, *The art of computer programming, volume 4a: Combinatorial algorithms, part 1*, ADDISON WESLEY PUB CO INC, 2011.

[Kru86] Stan Krute, *Commodore 64/128 graphics and sound programming*, TAB Books Inc, 1986.

[LE83] David Lawrence and Mark England, *Commodore 64 machine code master: A library of machine code routines*, Reston Pub Co, 1983.

[LE85] ———, *Il linguaggio macchina del commodore 64. con floppy disk*, Jackson Libri, 1985.

[Lea86] J. William Leary, *Introduction to computer programming with comal 80 and the commodore 64/128*, COMAL Users Group, Ltd, 1986.

[Lee87] Sheldon Leemon, *Mapping the commodore 64 & 64c*, Compute, 1987.

[Lev86] Lance A. Leventhal, *6502 assembly language programming*, McGraw-Hill Osborne Media, 1986.

[Lin83] Len Lindsay, *Comal handbook*, Brady (Robert J.) Co ,U.S., 1983.

[Lin09] Charles Lindner, *Design theory*, Chapman & Hall/CRC Press, Boca Raton, 2009.

[LO90] Thomas Lundy and Rory O'Sullivan, *Beginning structured programming in basic and comal*, Gill & Macmillan, 1990.

[LS83] Gordon Lee and Nevin B. Scrimshaw, *The commodore puzzle book - basic brainteasers*, Birkhauser, Boston, USA, 1983.

[Man13] David Manlove, *Algorithmics of matching under preferences*, World Scientific Publishing Company, March 2013.

[Mar85] Art Margolis, *Troubleshooting and repairing your commodore 64*, Tab Books, 1985.

[MC15] Ernesto Mordecki and Hector Cancela, *On the number of open knight's tours*.

[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.

[O'M85] Timothy J. O'Malley, *Artificial intelligence projects for the commodore 64*, Tab Books, Inc., 1985.

[PK84] Paul Pavelko and Tim Kelly, *Master memory map for the commodore 64*, Prentice Hall, 1984.

[Ple84] Axel Plenge, *The graphics book for the commodore-64*, Abacus Software Inc, 1984.

[RN09] Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach (3rd edition)*, Pearson, 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.

[Sch91] Allen Schwenk, *Which rectangular chessboards have a knight's tour?*, Mathematics Magazine **64** (1991), 325–332.

[Seb15] Robert W. Sebesta, *Concepts of programming languages (11th edition)*, Pearson, 2015.

[SH11] Maureen Sprankle and Jim Hubbard, *Problem solving and programming concepts (9th edition)*, Pearson, 2011.

[Sin84] Ian Robertson Sinclair, *Introducing commodore 64 machine code*, Prentice-Hall, 1984.

[Smi85] Bruce Smith, *Commodore 64 assembly language*, Chapman and Hall, 1985.

[Sta86] Richard P. Stanley, *Enumerative combinatorics*, vol. 1, Wadsworth Publ. Co., Belmont, CA, 1986.

[Sti03] Douglas Stinson, *Combinatorial designs: Construction and analysis*, SPRINGER NATURE, 2003.

[Sut85] James Sutton, *Power programming the commodore 64: Assembly language, graphics, and sound*, Prentice-Hall, 1985.

[Wal07] W.D. Wallis, *Introduction to combinatorial designs (discrete mathematics and its applications)*, Chapman and Hall/CRC, 2007.

[Weg00] Ingo Wegener, *Branching programs and binary decision diagrams*, SIAM - Society for Industrial and Applied Mathematics, 2000.

[Wir73] Niklaus Wirth, *Systematic programming: An introduction*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.

[Zak83] Rodnay Zaks, *Programming the 6502*, SYBEX, 1983.