# Linguaggi, codici e alberi binari

**Mauro Torelli**

Note per il corso di *Algoritmi e strutture di dati* – 2011

## 1 Linguaggi

Un ***linguaggio*** è semplicemente un insieme di ***parole*** costruite su un determinato ***alfabeto*** di *simboli* (numerici, caratteri o altro). Chiameremo *lunghezza* di una parola il numero di simboli che la compongono, e denoteremo con $|p|$ la lunghezza di una parola $p$.

Talvolta si indica la lunghezza di una parola $p$ con $\lg(p)$. Menzioniamo questo fatto per notare l'*analogia con il logaritmo*: l'equazione $\lg(p \bullet q) = \lg(p) + \lg(q)$ vale per entrambi i casi (e può aiutare a ricordare questa *proprietà caratteristica dei logaritmi*) se con l'operatore $\bullet$ denotiamo, nel caso delle parole, il *prodotto di concatenazione,* che appunto "concatena", ossia "scrive di seguito", due o più parole. Nel caso del prodotto di parole è del tutto naturale *omettere* il simbolo di prodotto, proprio come spesso si fa per la moltiplicazione ordinaria. Facciamo notare che la lunghezza della rappresentazione binaria di un intero positivo $n$ è *circa* $\lg n$, e precisamente $\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$ (in questa frase lg denota il logaritmo in base 2, mentre $\lfloor x \rfloor$ denota la *parte intera* di $x$ e $\lceil x \rceil$ l'arrotondamento all'intero superiore: $\lfloor 3.14 \rfloor = 3$, $\lceil 3.14 \rceil = \lceil 4 \rceil = 4$). La dimostrazione di questo fatto è immediata osservando che la lunghezza della rappresentazione binaria di un intero positivo $n$ aumenta di 1 (rispetto a quella di $n - 1$) quando (e solo quando) $n$ è una potenza di 2 e che per $n = 1$ la lunghezza è 1.

I simboli dell'alfabeto $A$ sono essi stessi parole? Sono *indistinguibili*, se non concettualmente, dalle parole di lunghezza 1. A questo punto possiamo considerare anche $A$ come un linguaggio, costituito da tutte le parole di lunghezza 1. Quali sono invece le parole di lunghezza 0? Una parola di lunghezza 0 per definizione non ha simboli di $A$, dunque non si può differenziare tramite $A$ e sarà pertanto *unica*; si pone poi il problema di visualizzarla in qualche modo. Useremo il simbolo $\varepsilon$ (talvolta si preferisce $\lambda$) per indicare questa ***parola vuota***: il simbolo $\varepsilon$ *non* deve far parte dei simboli di $A$, altrimenti denoterebbe una parola di lunghezza 1!

Una *convenzione* comoda e abbastanza diffusa, cui cercheremo di attenerci, prevede che, dovendo usare dei simboli per indicare intere parole di un linguaggio, si usino le lettere *in fondo* all'alfabeto (per esempio da $p$ a $z$) per denotare *parole* di lunghezza arbitraria, quelle *all'inizio* per denotare invece *simboli* di $A$, ovvero parole di lunghezza 1.

La nozione di prodotto introdotta per le parole ci consente di definire coerentemente la nozione di ***fattore***: la parola *$x$ è un fattore di $y$ se $y = uxv$*. In altri termini, un fattore di $y$ è una *sottoparola* (contigua) di $y$: il termine *fattore* è più comodo e generalizza le nozioni di prefisso e suffisso. Il *fattore sinistro* (come alcuni autori francesi preferiscono chiamarlo) $u$ è un ***prefisso*** di $y$ mentre il *fattore destro* $v$ è un ***suffisso***.

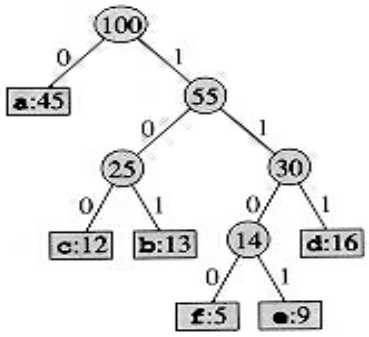
## 2 Linguaggi ereditari e alberi binari

Vediamo ora qualche esempio interessante di linguaggio, applicando i concetti introdotti sopra. Un linguaggio $L$ in cui se $xy \in L$ allora $x \in L$ si chiama ***linguaggio ereditario***. La ragione di questa denominazione è evidente: l'appartenenza al linguaggio è *ereditata* da ogni *prefisso.* Notiamo che, se $L$ è un linguaggio ereditario, o $L = \emptyset$, oppure la parola vuota $\varepsilon \in L$, perché la parola vuota è prefisso di ogni altra.

Prendiamo ora come alfabeto l'*alfabeto binario* $A = \{0, 1\}$ e consideriamo per esempio il linguaggio ereditario $B = \{\varepsilon, 0, 1, 10, 11, 100, 101, 110, 111, 1100, 1101\}$: è facile controllare che i prefissi $\varepsilon$, 1, 11 e 110 dell'ultima parola elencata 1101 appartengono tutti a $B$, così come quelli di ogni altra parola di $B$.

Sorprendentemente, almeno a prima vista, possiamo interpretare il nostro linguaggio $B$ (e ogni altro linguaggio ereditario binario!) come un ***albero binario***: $\varepsilon$ è la *radice* dell'albero e ogni parola $x0$ di $B$ è *figlio sinistro* di $x$ mentre $x1$ ne è *figlio destro* (l'*ereditarietà* ci garantisce che tanto $x0$ quanto $x1$, se appartengono a $B$, hanno un *genitore* $x$, di solito chiamato, con un po' di maschilismo, *padre*). Immaginando di non avere già una nozione di albero binario, potremmo assumere come *definizione* che *un albero binario è un linguaggio binario ereditario*.

L'interpretazione di un linguaggio ereditario come un albero può naturalmente essere estesa ad alberi non binari. Essa può apparire artificiosa o inutile, ma costituisce in realtà, oltre che un interessante esercizio di… ginnastica mentale, un approccio astratto che può semplificare alcune considerazioni sugli alberi, evitando (*astraendo* da) particolarità e dettagli (come per esempio i puntatori ai figli o al padre o NIL per le foglie, irrilevanti in molte considerazioni teoriche).




Pensare l'albero come un linguaggio può risparmiare la necessità di figure. Per chi, invece, è convinto che "una figura val mille parole", ecco un disegno dell'albero associato al linguaggio $B$ (ignorare il contenuto dei nodi, che ci servirà più avanti). Ciascuna parola di $B$ corrisponde all'etichettatura del *cammino* dalla radice al nodo corrispondente, etichettando con 0 gli archi sinistri e con 1 i destri.

L'*altezza* dell'albero è (*per definizione*, se non la conosciamo già diversamente) la *lunghezza massima delle parole* del linguaggio (4 nel nostro esempio). Poiché le parole binarie di lunghezza *esattamente k* sono $2^k$ e quelle di lunghezza *inferiore a k* sono $2^k - 1$ (come si dimostra immediatamente per induzione), per avere $2^k$ parole distinte ci dev'essere *almeno una parola di lunghezza k*, e in generale per avere $n$ parole binarie *distinte*, almeno una di esse dovrà avere lunghezza $\lfloor \lg n \rfloor$ o più. Dunque *qualunque albero binario con n nodi ha altezza almeno pari a* $\lfloor \lg n \rfloor$ (e *al più* pari a $n - 1$: lo si dimostri per esercizio).

## 3. Codici

Dato un qualsiasi linguaggio $C$, possiamo usare le sue parole come *fattori* per produrne altre e ottenere così nuovi linguaggi. Se accade che le fattorizzazioni sono *uniche* (come nel caso dei *primi* in aritmetica, commutatività a parte) allora diciamo che $C$ è un ***codice***. In altri termini, $C$ è un codice se *non* è possibile scrivere *equazioni* tra prodotti di parole di $C$, ovvero fattorizzare (*decodificare*) in più modi.

Spieghiamoci meglio con un esempio che riprenderemo anche più avanti. Abbiamo un insieme $A$ di *caratteri* da *codificare* in binario, $A = \{a, b, c, d, e, f\}$. Poiché i caratteri sono 6, saranno necessarie parole binarie di lunghezza almeno 2. Potremmo assumere $C = \{0, 00, 01, 1, 10, 11\}$ e la *corrispondenza* $a = 0$, $b = 00$, $c = 01$, $d = 1$, $e = 10$, $f = 11$. Perché *C non è un codice*? Perché, per esempio, $ad = c = 01$: codifichiamo allo stesso modo le due parole diverse $ad$ e $c$, e quindi, nella *decodifica*, quando troviamo la parola 01 non sappiamo se corrisponde a $ad$ oppure a $c$: una situazione ovviamente *ambigua* e inaccettabile.

Un ***codice prefisso*** è un linguaggio in cui *nessuna parola è prefisso di un'altra*. Questa condizione assicura che il linguaggio *è* un codice: lo dimostriamo *per assurdo*. Supponiamo che il linguaggio *non* sia un codice, e quindi esista un'equazione del tipo $x\,y\,\ldots\,z = u\,v\,\ldots\,w$, dove $x, y, \ldots, z, u, v, \ldots, w$ sono parole del linguaggio. Se fosse $x = u$ potremmo eliminare $x$ e $u$ ottenendo ancora un'equazione e continuare eventualmente a eliminare coppie di parole eguali: supponiamo quindi $x \neq u$. Ma allora, se $x$ e $u$ sono parole *diverse* con la *stessa lunghezza*, come può valere l'uguaglianza? L'unica possibilità è che $x$ e $u$ abbiano *lunghezze diverse ma comincino allo stesso modo*, ossia che una parola sia *prefisso* dell'altra. Se questo non può succedere non può esserci neppure l'uguaglianza, e quindi il linguaggio "senza prefissi" è un codice.

Attenzione! La condizione (di non avere prefissi) è *sufficiente* ma non *necessaria*. Il modo più semplice di mostrare che qualcosa non è vero è di esibire, quando possibile, un *controesempio*. Qui è facile: quello che abbiam detto per i prefissi lo possiamo dire, simmetricamente, per i suffissi, e un *codice suffisso* (ossia *senza* suffissi) può non essere prefisso (ossia può ammettere parole che *sono* prefissi di altre), come per esempio $C = \{0, 01, 11\}$: 0 è prefisso di 01, ma non ci sono suffissi. Anche $C = \{00, 001, 011, 01, 11\}$ è un codice, pur contenendo 00, prefisso di 001, e 11, suffisso di 011. Per la dimostrazione si può vedere il libro *Algebraic Combinatorics on Words*, (esempio 6.1.5) disponibile in rete all'URL (occhio ai trattini!) www-igm.univ-mlv.fr/~berstel/Lothaire/

## 4 Codici di Huffman

L'algoritmo di Huffman, pubblicato nel lontano 1952, costruisce un codice prefisso *ottimo*. Ormai sappiamo cosa significa "codice prefisso", quindi non resta da chiarire che il termine "ottimo": una soluzione è ottima se è *la migliore possibile*, ossia se non è possibile fare di meglio rispetto all'obiettivo posto.

L'obiettivo scontato di una codifica è quello della *non ambiguità*; un secondo obiettivo è la *facilità della*

*decodifica*, che nei codici prefissi è acquisito, dato che basta arrestarsi appena trovata una parola del codice (non può essercene un'altra che inizi allo stesso modo). L'obiettivo principale, a questo punto, è quello di risparmiare memoria: un codice ottimo è un codice che usa la quantità di memoria *minima* possibile per codificare un dato testo. Si può inoltre dimostrare che non solo un codice di Huffman è ottimo *tra quelli prefissi*, ma anche che non vi sono codici che usino meno memoria, pur abbandonando il comodo requisito dell'essere prefissi.

Facciamo una breve digressione. Tutti probabilmente abbiamo usato programmi di *compressione*, che riducono l'occupazione di memoria di molti file. Possiamo chiederci come *funziona* in dettaglio un programma di compressione dei dati, oppure quali sono le *idee* su cui è basato, o come vengono realizzate in maniera *efficiente*, o quale *interfaccia* è più opportuno fornire all'utente. Un corso di *algoritmi* offre qualche risposta alle due domande centrali (e tipicamente presenta l'*algoritmo di Huffman*: per algoritmi diversi, prodotti commerciali e *molto* altro si può vedere per esempio la pagina http://www.data-compression.info/).

Porsi di punto in bianco l'obiettivo di *scrivere un programma* di compressione non è probabilmente la via migliore (salvo forse per qualche genio): non si tratta di trovare il linguaggio giusto e scarabocchiare un po' di diagrammi o seguire schemi di programmazione. Prima di tutto questo occorre avere un'idea di come procedere, ed è difficile partire da zero: anche in informatica, ormai, possiamo "appoggiarci sulle spalle di giganti" e far tesoro delle esperienze acquisite, pur di conoscerle.

Cerchiamo ora di arrivare all'algoritmo di Huffman usando alcune delle idee esaminate in precedenza. In realtà dobbiamo prima concepire l'idea che ci permetta di *risparmiare* spazio, *rispetto* alla situazione di partenza. Qual è la situazione di partenza? Quella in cui i caratteri del testo da comprimere (ci limitiamo ai testi) sono codificati tramite *codici a lunghezza fissa* (come il *codice ASCII*, si veda, per esempio, http://www.telacommunications.com/nutshell/ascii.htm: il "telacommunications" con la "a" è corretto! O il più moderno *Unicode* http://www.unicode.org/).

Idea base: usare parole di codice di *lunghezza variabile*, riservando i *codici brevi* ai caratteri *più frequenti*. Osservazione base: se le frequenze sono uguali, c'è poco da comprimere, ma anche in quel caso, se il numero di caratteri diversi non è una potenza di 2, si può fare qualcosina. Per fortuna, nei testi in linguaggio naturale le frequenze sono in generale abbastanza diverse (si veda la *legge di Zipf*).

Seconda idea. Vogliamo ottenere un codice *prefisso*: c'è modo di farlo "automaticamente", ossia di escludere la possibilità di avere parole che siano prefissi di altre? Se una parola è vista come un cammino dalla radice a un nodo, dobbiamo escludere le parole che corrispondono ad *antenati* (padri, nonni, ecc.) del nodo stesso: l'approccio più radicale è quello di usare esclusivamente i nodi "terminali" o *foglie* dell'albero.

Terza idea. Se partiamo dunque dalle foglie, dobbiamo cominciare a costruire l'albero dai caratteri *meno* frequenti. Abbiniamo i due caratteri meno frequenti e li sostituiamo con un "metacarattere" di frequenza pari alla somma dei due, rappresentato da un albero con una radice che ha come figli i due caratteri. Procediamo così fino ad esaurire i nostri caratteri.

Primo problema: chi ci garantisce che alla fine abbiamo un codice ottimo, ossia che la codifica è la più breve possibile? Dovremo dimostrarlo!

Secondo problema: come troviamo i due caratteri o metacaratteri di minima frequenza? Un programmatore questo problema non se lo pone nemmeno: è facile, basta tenere tutto ordinato o trovare di volta in volta il minimo. Ma è il modo più efficiente? Trovare il minimo o inserire il nuovo metacarattere tra $n$ elementi ordinati può richiedere tempo dell'ordine di $n$ e, iterando $n$ volte, il tempo risulta quadratico. Invece usando un heap per il minimo lo estraiamo in tempo costante e sistemare le cose dopo l'estrazione richiede solo tempo logaritmico: $n$ estrazioni richiederanno tempo dell'ordine di $n \lg n$, molto meglio.

Per i dettagli dell'algoritmo si può vedere il testo. La dimostrazione dell'ottimalità nel libro di Cormen et al. è abbastanza lunga e la conclusione che dai due lemmi dimostrati discenda il risultato non è proprio evidente. Proviamo dunque a dare *una dimostrazione* completamente *diversa*. È utile fare prima una breve digressione sulle catene di addizioni.

## 5. Catene di addizioni

Supponiamo di voler calcolare il valore di $(1+1/n)^n$ per interi $n$ grandi senza ricorrere ai logaritmi (per verificare per esempio l'approssimazione di $e$ al variare di $n$), oppure di usare il "*piccolo*" *teorema di Fermat* per dimostrare che un intero $n$ è *composto*, calcolando $2^{n-1} \bmod n$ (che è diverso da 1 solo se $n$ è *composto*): occorre disporre di un algoritmo efficiente per calcolare potenze $n$-esime anche per valori "grandi" di $n$, effettuando molto meno di $n$ prodotti.

Il *metodo binario* (o *dicotomico*) risolve il problema con un numero di prodotti proporzionale a $\lg n$. Lo illustriamo con un esempio. Dovendo calcolare $x^{15}$ possiamo calcolare $x^2$ con un prodotto, $x^3 = x\,x^2$ con un ulteriore prodotto, $x^6 = x^3\,x^3$ con un terzo, $x\,x^6$ ci dà $x^7$ con 4 prodotti, $x^7\,x^7$ ci dà $x^{14}$ e finalmente $x\,x^{14}$ ci dà $x^{15}$ con 6 prodotti in totale. Poiché interessano sostanzialmente gli *esponenti*, conviene considerare la sequenza di tali esponenti e organizzare le cose per esempio come segue: partendo da $n$, se l'esponente è pari, si dimezza, se è dispari, si sottrae uno. Per esempio, 15→14→7→6→3→2→1: come prima, in 6 passaggi si arriva da 15 a 1 o viceversa.

Sequenze di questo tipo sono note come ***catene di addizioni***, perché si richiede che *ogni elemento della sequenza sia ottenibile come somma di due elementi* (eventualmente coincidenti) che lo seguono nella sequenza, o lo precedono se le sequenze sono considerate in ordine crescente, come fa Knuth nel secondo volume di *The Art of Computer Programming – Seminumerical Algorithms* (Addison-Wesley, 1981[2]), formula (1) a pagina 444.

Knuth ricorda anche che alcuni autori hanno scritto incautamente che il metodo binario è il migliore possibile. Tuttavia, la catena 1→2→3→6→12→15 richiede soltanto 5 passaggi ed è pertanto più efficiente per $n = 15$, il più piccolo *controesempio*. Non è nota una formula che dato $n$ fornisca la catena di addizioni più corta, e vi sono anzi numerosi problemi aperti nella teoria delle catene di addizioni.

È interessante conoscere il *metodo dicotomico* (utile anche per altri algoritmi: la ricerca dicotomica, mergesort…) e sapere che una potenza *n*-esima può essere calcolata con un numero di prodotti dell'ordine di lg *n*, e anche che il metodo binario non è necessariamente il migliore. Noi ci ispireremo alle catene di addizioni per cogliere gli aspetti essenziali dell'algoritmo di Huffman e dare così una dimostrazione semplice della sua ottimalità.

## 6. I codici di Huffman sono ottimi

Data una lista *L* di interi positivi (con possibili ripetizioni) chiameremo *catena di addizioni generata da L* una lista o sequenza *non decrescente* di interi positivi comprendente *L* e tale che *ogni elemento non in L sia la somma di due elementi precedenti nella sequenza*. Per evidenziare gli elementi di *L* nella catena li sottolineeremo.

Per esempio, (5, 9, 12, 13, 16, 45) è una catena che coincide con la lista generatrice *L*; (5, 9, 12, 13, 14, 16, 45) è una catena generata dalla medesima lista *L*, con l'aggiunta dell'elemento 14 ottenuto da 5 + 9.

Chiamiamo *alberi pienamente binari* gli alberi binari in cui tutti i nodi che hanno figli (i nodi *interni*) ne hanno due (non ci sono figli unici, ma solo nodi interni e nodi *esterni*, detti anche *foglie*). Gli alberi pienamente binari che hanno gli elementi di *L* nelle *foglie* e la somma dei nodi figli in ciascun *nodo interno* corrispondono a *particolari* catene tali che *ogni elemento della catena* (tranne l'ultimo) *viene usato come addendo in una e una sola somma* (quella corrispondente all'elemento *padre* nell'albero). In altre parole, possiamo pensare di *eliminare* due elementi di *L* sostituendoli con la loro somma, poi sostituire altri due elementi (inclusi eventualmente i nuovi) con la loro somma, e così via, proprio come si fa per costruire l'albero di Huffman.

Dopo quanti passi resta *un solo elemento* (di valore pari, ovviamente, alla *somma degli elementi di L*)? Si tratta in sostanza di un *torneo a eliminazione* un po' insolito, poiché il numero di partite necessarie per vincere può variare da giocatore a giocatore e per di più *ogni partita elimina entrambi i giocatori*, sostituendoli con… la loro somma; se i giocatori sono *n*, il vincitore si avrà dopo $n - 1$ partite. Sappiamo dunque *quanto sono lunghe queste catene/tornei* e anche *quanti sono i nodi interni in un albero pienamente binario*: *se vi sono n foglie, vi sono* $n - 1$ *nodi interni*!

Chiameremo semplicemente ***tornei*** queste catene che contengono i "giocatori" iniziali (i valori sottolineati) e via via i nuovi giocatori (somme) fino al vincitore. Per esempio, la catena $C$ = (5, 9, 12, 13, 14, 16, 25, 39, 45, 61, 100) è un torneo generato dalla lista *L* di giocatori già usata come esempio, lista che, guarda caso, contiene i valori di frequenza (in migliaia) dell'esempio del testo per l'algoritmo di Huffman, che sono anche i valori riportati nella figura al §2 di queste note. Abbiamo cercato di indicare con colori in qualche modo analoghi, di volta in volta, gli addendi e il risultato. Notiamo che occorre "costruire" 25 come 12 + 13, non come 9 + 16, per rispettare il vincolo dell'*eliminazione*: nessun valore è sommato due volte, 9 è usato in 14 = 5 + 9 e 16 in 61 = 16 + 45.

In un certo senso, dunque, il torneo contiene “meno informazione” di un albero corrispondente, o meglio l’informazione del torneo richiede di essere elaborata (determinando le coppie corrette da sommare), mentre nell’albero è esplicita (“vediamo” un nodo 25 padre di 12 e 13). Ma se l’obiettivo è quello di arrivare a una dimostrazione più semplice della correttezza dell’algoritmo di Huffman, allora è preferibile *ridurre l’informazione agli elementi essenziali* (i valori da minimizzare) invece di “distrarsi” con elementi accessori (la forma dell’albero).

Il problema risolto dall’algoritmo di Huffman può essere ora riformulato come quello di *determinare un torneo ottimo*, ossia *avente somma minima* (dove per “somma” intendiamo la somma di *tutti* gli elementi), data la lista *L* delle frequenze. Perché questo è vero? È l’esercizio 16.3-3 (o 16.3-4) del libro: *la somma è il* ***costo*** dell’albero! Infatti, la frequenza di una foglia al livello *k* è contata *k* volte nella somma (viene *sommata* su tutti i livelli superiori!). Poco importa che sommando tutti gli elementi del torneo si aggiunga anche la somma degli elementi di *L*, perché, fissata *L*, questa somma è costante! Il torneo dell’esempio del libro è (5, 9, 12, 13, 14, 16, 25, 30, 45, 55, 100), con somma 324 = 224 + la somma 100 degli elementi sottolineati.

È ora evidente che, dovendo utilizzare tutti gli elementi di *L* e *tutti* quelli via via generati (tranne l’ultimo, corrispondente alla somma delle frequenze), per ottenere la somma minima *basta* sommare di volta in volta i due elementi disponibili (ricordiamo che si possono usare *una volta sola*!) *più piccoli*: se non lo faccio, posso introdurre nella sequenza un elemento più grande del necessario (ma non è detto che la somma finale sia *maggiore*: quello che mi interessa è il fatto che non può essere *minore*). Il torneo di Huffman è semplicemente quello in cui la somma dei primi *k* elementi è minima, per ogni *k*.

Un torneo *ottimo* per la lista *L* del nostro esempio è (5, 9, 12, 13, 14, 16, 25, 30, 45, 55, 100), che è quello ottenuto con l’algoritmo di Huffman. Un altro è, per esempio, (5, 9, 12, 13, 14, 16, 26, 29, 45, 55, 100): pur essendoci svincolati dalla *forma* dell’albero, grazie alle semplici *sequenze* dei tornei, non esiste però in generale un *unico torneo ottimo*, e vi sono *molti* più alberi ottimi e codici ottimi (le etichettature con 0 e 1 sono arbitrarie: possiamo cambiare in molti modi le etichette degli archi).

*Esercizi* (semplicissimi).

1. Stabilire se il torneo visto in precedenza, (5, 9, 12, 13, 14, 16, 25, 39, 45, 61, 100), è ottimo.

2. La catena (5, 9, 12, 13, 14, 16, 26, 29, 45, 55, 100) è davvero un torneo? Il 29 non è sommato sia in 16 + 29 = 45 sia in 26 + 29 = 55?