Strutture Dati e Codifiche

Ivano Salvo
Sapienza Università di Roma
email: salvo@di.uniroma1.it

Anno Accademico 2011-12

Presentazione: La presente dispensa introduce ai vantaggi di usare strutture dati *astratte*. In un secondo momento, si argomenterà che da un punto di vista di cosa si può calcolare, i numeri naturali sono l'unico tipo di dato necessario e si affronterà il problema delle codifiche.

1 Strutture Dati: Introduzione

Forse avete già sentito il bisogno di qualche struttura dati più sofisticata oltre alle semplici variabili intere o puntatori. Ad esempio, implementando la funzione `mcdDellaMaestra` in un primo momento avete sentito il bisogno di *memorizzare* i divisori primi di ciascun numero. Tuttavia ciò sembrava difficile, perchè avremmo avuto bisogno di un numero di variabili non staticamente prevedibile. Oppure nella generazione di anagrammi ci sarebbe stato utile manipolare dati complessi, come insiemi o sequenze.

Più in generale, è chiaro che risulta utile poter trattare *dati aggregati* (immaginate una sequenza di temperature, la tabella dei voti che riassume i risultati di un esame etc.) in cui le singole componenti siano accessibili ad esempio mediante un *numero*, un *codice*, o effettuando una *ricerca*, o ancora reperendoli in base all'ordine in cui tali elementi sono stati inseriti nel dato aggregato.

1.1 Tipi di dato: astrazione e implementazione

Un famoso slogan coniato dal Turing Award[1] Nicklaus Wirth[2] recita: "Algoritmi + Strutture Dati = Programmi[3]. Lo slogan evidenzia il principio secondo cui la corretta progettazione delle strutture dati manipolate da un programma è essenziale per rendere i programmi facili da scrivere e soprattutto da leggere, e quindi modificare. In questa sezione ci limiteremo a introdurre nozioni quali *struttura dati*, o *tipo*

[1] Una sorta di premio Nobel per l'Informatica.

[2] Nicklaus Wirth, è stato uno dei grandi studiosi dei linguaggi di programmazione. E' noto soprattutto in quanto ha definito e implementato, tra gli altri, il linguaggio Pascal ed il Modula 2.

[3] è anche il titolo di un celeberrimo libro, di cui consiglio la lettura agli interessati.

(astratto) di dato, lasciando a futuri esempi il compito di dare piena consapevolezza a questi concetti.

Esempio: Labirinto Immaginiamo di avere un labirinto e di voler scoprire il percorso che conduce dalla stanza di ingresso a quella di uscita. Immaginiamo che in ogni stanza ci siano alcuni corridoi che escono, andando verso altre stanze. Supponiamo di voler scrivere una procedura di visita di un labirinto, che trovi sempre, se esiste, una strada per uscire, come sequenza di corridoi percorsi.

Ora, indipendentemente da quale sia la *rappresentazione* del labirinto come struttura dati, quali sono le *azioni* che dobbiamo saper compiere per uscire dal labirinto? Innanzitutto ci serviranno le operazioni di percorrere un corridoio e riconoscere se una stanza sia un'uscita o meno. Ci servirà inoltre, un meccanismo per poter marcare le strade percorse e quindi uno per riconoscere quelle già percorse (il nostro filo di Arianna); sarà necessario inoltre saper tornare indietro quando si è stabilito che da una certa stanza non c'è via d'uscita (ad esempio, quando vi torniamo e non troviamo nuovi corridoi uscenti non già percorsi.

Avendo a disposizione questo tipo di operazioni, e supponendo che il progettista della struttura dati "labirinto" le abbia messe a nostra disposizione, possiamo scrivere un algoritmo per uscire dal labirinto come illustrato in Fig.1.1. Osserviamo che non è necessario conoscere l'*implementazione* dei tipi `labirinto, stanza` o `cammino`, ma solo la *specifica* delle operazioni definite su oggetti di questi tipi. Non solo, ma **non** conoscendo la struttura dati usata per implementare questi tipi, siamo *costretti* ad usare le operazioni rese disponibili da chi ha definito questi tipi di dato: questa perdita di libertà, tipicamente si rivela un vantaggio. Un po' come le regole del galateo.

1.2 Invarianti di Tipo di Dato

Il mascheramento dell'implemetazone di una struttura dati assicura che l'utilizzatore acceda in modo coerente alla struttura dati: sotto la responsabilità del suo progettista, rispetterà un *invariante di tipo di dato*: accessi incoerenti, infatti, potrebbero generare strutture dati con rappresentazioni non previste dal progettista della struttura dati e quindi minare la correttezza delle operazioni definite sulla struttura dati. Discuteremo più a fondo questa questione più avanti. Limitiamoci per ora ad un esempio.

Esempio: Insiemi Supponiamo di dover progettare un tipo di dato insiemi di interi. Siamo alle prime armi e conosciamo solo i vettori. Siamo un po' infastiditi dal dover decidere una dimensione massima a priori, ma supponiamo che per l'utilizzo previsto questo non sia un problema.

Ci rendiamo subito conto che alcune operazioni (ad esempio test di appartenenza o test di inclusione) risultano estremamente più efficienti qualora l'insieme sia

```
percorso esciLabirinto(labirinto L)
  stanza s = entrata(L);
  corridoio c;

  p = camminoVuoto();

  while(1){
    if (uscita(s)) return p;          /* FINE: sono arrivato all'uscita */
    if (c=prossimoCorridoio(s)){      /* se esiste un corridoio NON percorso */
         marca(L, c);                 /* marca il passaggio in c */
         s = percorriAvanti(c);       /* vado nella prossima stanza */
         p = aggiungi(c, p);          /* aggiorna cammino percorso */
        }
   else /* non ci sono nuove strade da esplorare da s*/
     { if (nonVuoto(p))               /* se non e' una stanza iniziale ...*/
          { c = ultimoPercorso(p);    /* seleziona ultimo corridoio percorso */
            s = percorriIndietro(c);  /* torna indietro */
            togli(c, p);              /* aggiorna cammino percorso */
            }
       else return camminoVuoto();    /* FINE: non c'e' soluzione */
      }
   }
 }
```

Figura 1: Programma per uscire da un labirinto

rappresentato da un *vettore ordinato* (ovviamente *senza ripetizioni*, trattandosi di un insieme).

Se il programmatore "utente" può costruire insiemi solo usando le funzioni messe a disposizione dal programmatore progettista della struttura dati (ad esempio `void inserisci(int x, insieme S)`, o `insieme creaInsiemeVuoto()`) si suppone venga mantenuto l'invariante di rappresentazione da tali funzioni (insieme rappresentato con un vettore ordinato senza ripetizioni).

Se invece il programmatore "utente" avesse accesso al vettore che rappresenta l'insieme, potrebbe aggiungere nel vettore un doppione oppure mettere un elemento fuori posto minando la correttezza di tutte le altre funzioni, che richiedono la precondizione di operare su un vettore ordinato.

Un secondo importante vantaggio di questo approccio è legato al cosidetto *riuso del software*: se il progettista della struttura dati labirinto (o insieme) volesse cambiare la *rappresentazione* del tipo di dato labirinto, ad esempio per implementare in modo efficiente una nuova operazione non prevista all'inizio, avendo cura di salvaguardare la specifica delle operazioni già definite, non renderebbe necessario riscrivere i programmi che usano la struttura dati, per i quali la modifica della rappresentazione sottostante sarebbe del tutto *trasparente*.

Ad esempio, il progettista degli insiemi di interi potrebbe a un certo punto imparare a usare strutture dati più sofisticate (liste o alberi binari di ricerca) e cambiare

la rappresentazione degli insiemi. Ma se ha cura di lasciare intatta l'*interfaccia* cioè l'insieme delle operazioni definite sugli insiemi, programmi che utilizzano questa struttura dati continueranno a funzionare correttamente.

Tipicamente, solo gravi ragioni di efficienza, potrebbero sporadicamente suggerire di violare questa gerarchia di *astrazioni*. Nel seguito vedremo le strutture linguistiche offerte dal C per definire nuove strutture dati e progetteremo alcune strutture dati.

2 Tiny C

Non so se qualche lettore se lo sta già chiedendo, ma le variabili che contengono numeri naturali arbitrariamente grandi (non le povere variabili `int` del C) sono sufficienti a esprimere qualunque calcolo! In realtà ne bastano due! Eccovi, finalmente, tutto insieme, il TinyC, sufficiente ad esprimere un qualunque calcolo ($i \in \{1, 2\}$):

$$\begin{array}{lcl} C & ::= & \textbf{while}\ (E)\ C; \quad | \quad C_1; C_2 \quad | \quad x_i = E \\ E & ::= & x_i + 1 \quad | \quad E_1 == E_2 \quad | \quad E_1 ! = E_2 \quad | \quad x_i \end{array}$$

Onde evitare di introdurre funzioni di input/output, possiamo sempre supporre che l'eventuale input del programma sia memorizzato all'inizio nella variabile x_1 e che anche l'output, al termine dell'esecuzione, sia memorizzato nella variabile x_1.

Se vi sentite più portati per la ricorsione, potete sostituire il ciclo **while**, con la definizione e chiamata di funzione (si può far vedere che in questo caso potete anche fare a meno dell'assegnazione e della sequenza, richiamando dalla panchina una espressione o comando condizionale).

3 Codifiche

Cerchiamo di argomentare con un esempio sufficientemente generale (senza dimostrazioni formali che sono ben aldilà dello scopo di queste note) come sia possibile memorizzare qualunque dato nei numeri naturali. Supponiamo per esempio di aver bisogno di memorizzare una sequenza di interi (eventualmente negativi) $\langle a_1, a_2, \ldots, a_n \rangle$. Prima di tutto, osserviamo che esiste una facile biezione tra l'insieme dei naturali $\mathbb{N}$ e l'insieme dei numeri interi $\mathbb{Z}$. Ad esempio, potremo convenire che i numeri pari rappresentino interi positivi, mentre i numeri dispari rappresentino interi negativi. La funzione $c : \mathbb{Z} \mapsto \mathbb{N}$ definita da:

$$c(z) = \begin{cases} 2z & \text{se } z \geq 0 \\ -2z - 1 & \text{se } z < 0 \end{cases}$$

è una funzione che codifica ogni numero intero in un naturale. Essendo una funzione biiettiva, c è una codifica "economica" (non ci sono naturali che non sono immagine di nessun intero), ed ammette l'inversa $d : \mathbb{N} \mapsto \mathbb{Z}$:

$$d(n) = \begin{cases} \frac{n}{2} & \text{se } n \text{ è pari} \\ -\frac{n+1}{2} & \text{se } n \text{ è dispari} \end{cases}$$

A questo punto, possiamo codificare la sequenza $\langle a_1, a_2, \ldots, a_n \rangle$ con il numero naturale $p_1^{c(a_1)} \times p_2^{c(a_2)} \times \ldots \times p_n^{c(a_n)}$, dove p_i è l'i-esimo numero primo. Se stiamo codificando le sequenze esattamente lunghe n questa codifica non è suriettiva (ci sono chiaramente numeri naturali che non sono immagine di nessuna sequenza, precisamente tutti gli infiniti primi $p_{n+1}, p_{n+2}, \ldots$ e tutti i loro composti), viceversa è biiettiva se stiamo considerando sequenze di lunghezza arbitraria.

Non tutto ciò che è possibile è anche opportuno. Abbiamo visto, ad esempio, che le funzioni sono un potente mezzo di astrazione che rendono più leggibili i programmi e permettono di progettare i programmi riferendosi ad un esecutore più astratto di quello offerto dal linguaggio C – vedi esempio degli anagrammi, dove abbiamo progettato il programma riferendoci a un esecutore che ancora non abbiamo in effetti a disposizione. Questa metodologia, che consiste nell'esprimere la soluzione di un problema complesso in termini di soluzioni di problemi più semplici, va sotto il nome di *top-down*. L'approccio opposto è quello con cui si scrivono le librerie, che risolvono problemi la cui soluzione è di uso generale. Queste vengono scritte dal "basso", seguendo una metodologia detta *bottom-up*. In entrambi i casi, tuttavia, le soluzioni di problemi complessi diventano semplici, grazie ad una *gerarchia di astrazioni*.

Allo stesso modo, vedremo che i linguaggi di programmazione offrono meccanismi di *astrazione sui dati* che permettono di scrivere le soluzioni dei problemi in un linguaggio quanto più possibile *naturale*, rispetto al problema in esame. Il lettore, tuttavia, dovrà sempre tenere bene a mente due cose:

1. qualsiasi tipo di dato (come visto nella Parte 1) è sempre e comunque una *codifica*, compresa la notazione decimale posizionale dei numeri naturali che vi è familiare fin dall'infanzia!
2. codifiche che permettono di comprimere la rappresentazione di alcuni dati possono essere necessarie, nelle applicazioni in cui la memoria potrebbe essere una risorsa critica;
3. codifiche apparentemente poco naturali possono suggerire procedure risolutive semplici e inaspettate.

3.1 ⋆ Rivisitazione del problema degli anagrammi

Codifiche astute a volte sono state la chiave di progressi epocali nell'informatica. Ad esempio, la verifica di sistemi (soprattutto di microprocessori, e in misura più limitata di software), ha fatto progressi significativi negli anni '80 e '90 grazie all'impiego di un'ingegnosa codifica delle funzioni booleane, che permette di cogliere in modo "automatico" le regolarità di un sistema. Se preferite un esempio ancora più evidente, tutta l'ingegneria dei calcolatori e le più recenti applicazioni multimediali, hanno trovato comodo sfruttare il fatto che qualunque informazione può essere codificata e manipolata convenientemente con un alfabeto di soli due simboli, cioè in

```
void anagrammi(n){
   int f,i;

   f=fatt(n);
   for (i=1; i<=f; i++)
        stampaDecodificaSeq(i, n);
}
```

Figura 2: Funzione per stampare gli anagrammi

modo "digitale". Nel nostro piccolo, vedremo una soluzione alternativa al problema degli anagrammi di una parola.

Indichiamo con $\text{Perm}(A) \subseteq \text{Seq}(A)$ l'insieme di sequenze che formano le *permutazioni* di A, cioè quelle sequenze che contengono *tutti* gli elementi di A *senza ripetizioni*. Supponiamo, per semplicità di dover produrre tutte le sequenze dei primi n numeri naturali $[n] = \{1, 2, \ldots, n-1, n\}$. Questa ipotesi non è ovviamente restrittiva, perchè avendo da anagrammare n simboli distinti $\Sigma = a_1, a_2, \ldots a_n$, possiamo semplicemente considerare la biezione $c : [n] \mapsto \Sigma$, definita da $c(i) = a_i$.

Siccome sappiamo che i possibili ordini in cui possiamo disporre n oggetti sono $n!$, se riuscissimo a trovare una codifica biettiva $c : \text{Perm}[[n]] \mapsto [n!]$, saremo già a metà dell'opera. Infatti, sapendo scrivere in C una procedura `stampaDecodificaSeq(int c, int n);` che decodifica un codice c relativo a una sequenza di n interi e stampa la sequenza associata a c, la funzione in Fig. 3.1 stamperebbe correttamente tutte le sequenze in $\text{Perm}[[n]]$.

A questo punto ci poniamo il problema di trovare la codifica c, la sua inversa, la decodifica d, e poi di implementare opportunamente quest'ultima nella funzione `stampaDecodificaSeq(int c, int n);`. Per non fare confusione, come sempre andiamo con *ordine*. In questo caso, l'ordine è quello *lessicografico* (che corrisponde a quello del dizionario). Dato un insieme ordinato $(A, \leqslant)$, posso ordinare l'insieme $\text{Seq}[A]$ lessicograficamente, convenendo che una sequenza s_1 è minore di s_2 se:

1. s_1 è un prefisso di s_2;
2. il primo elemento di s_1 è minore di s_2;
3. il primo elemento di s_1 è uguale al primo elemento di s_2, e la coda di s_1 è minore della coda di s_2.

Siccome siamo dei forzati dell'induzione, formalizziamo la descrizione informale scritta sopra, definendo in modo rigoroso l'ordine lessicografico sulle sequenze $(\text{Seq}[A], \preceq)$ indotto dall'ordinamento $(A, \leqslant)$:

$$\begin{array}{rcl} & & \langle\,\rangle \preceq s \\ a \leqslant b & \Rightarrow & a \cdot s \preceq b \cdot s' \\ s \preceq s' & \Rightarrow & a \cdot s \preceq a \cdot s' \end{array}$$

dove le equazioni scritte sopra vanno lette quantificate per ogni $a, b \in A$ e per ogni $s, s' \in \mathrm{Seq}[A]$.

Alternativamente, nel caso in cui consideriamo sequenze di ugual lunghezza (che è il caso delle permutazioni), potremo dare la seguente semplice definizione "iterativa": Una sequenza $s = \langle a_1, \ldots, a_n \rangle$ è minore lessicograficamente di $s' = \langle a'_1, \ldots, a'_n \rangle$ se il minimo k per cui $a_k \neq a'_k$ allora ho $a_k < a'_k$.

Siccome tutte le sequenze che cominciano per 1 precedono in quest'ordine tutte quelle che cominciano con un altro numero, saranno numerate da 1 a $(n-1)!$, e analogamente tutte le sequenze che cominciano per 2 saranno numerate da $(n-1)!+1$ a $2(n-1)!$, e quindi tutte le sequenze che cominciano per 3 saranno numerate da $2(n-1)!+1$ a $3(n-1)!$ ...fino alle sequenze che cominciano per n che saranno numerate da $(n-1)(n-1)!+1$ a $n(n-1)! = n!$. Le sequenze che cominciano con un dato numero k, saranno quindi numerate da da $(k-1)(n-1)!+1$ a $k(n-1)!$. Questo suggerisce che la codifica può essere definita ricorsivamente come segue[4] (ricordate che l'operazione $\cdot$ concatena un elemento con una sequenza):

$$\begin{aligned} c(k \cdot s, n) &= (k-1)(n-1)! + c(s, n-1) \\ c(\langle\,\rangle) &= 1 \end{aligned}$$

C'è tuttavia un piccolo problema. La regola che abbiamo scritto si fonda sulla precondizione che io stia numerando l'insieme $[n]$ dei primi n numeri, mentre la sequenza s su cui io applico ricorsivamente la codifica c non rispetta questa condizione. Infatti, supponendo che la sequenza inizi per $k \neq n$, il resto della sequenza non è nell'insieme $\mathrm{Perm}[[n-1]]$ delle sequenze formate a partire dai primi $n-1$ numeri, quanto piuttosto nell'insieme $\mathrm{Perm}[[n] \setminus \{k\}]$. Questo piccolo problema si può risolvere semplicemente codificando la sequenza $s \in [n] \setminus \{k\}$ in una sequenza $s' \in [n-1]$ sottraendo 1 a tutti i numeri maggiori di k che vi compaiono. Ancora una volta possiamo scrivere questa codifica $c' : \mathrm{Perm}[[n] \setminus \{k\}] \mapsto \mathrm{Perm}[[n-1]]$ con facili equazioni ricorsive:

$$\begin{aligned} c'(m \cdot s, k) &= (m-1) \cdot c'(s, k) && \text{se } m > k \\ c'(m \cdot s, k) &= m \cdot c'(s, k) && \text{se } m < k \\ c'(\langle\,\rangle) &= \langle\,\rangle \end{aligned}$$

La corretta definizione della codifica di una sequenza in $\mathrm{Perm}[[n]]$ tale che ad ogni sequenza s associa un numero naturale che rappresenta il posto che s occupa in $\mathrm{Perm}[[n]]$ nell'ordine lessicografico è la seguente:

$$\begin{aligned} c(k \cdot s, n) &= (k-1)(n-1)! + c(c'(s, k), n-1) \\ c(\langle\,\rangle) &= 1 \end{aligned}$$

[4] ponendo $c(\langle\,\rangle) = 0$ otteniamo un'altra codifica delle sequenze nell'insieme di naturali $\{0, 1, \ldots, n!-1\}$ che è coerente con la logica di cominciare a contare da 0, che qua e là può risultare conveniente.

L'inversa $d(m,n)$ che preso un naturale m codifica di una sequenza lunga n ($1 \leq m \leq n!$) si costruisce con lo stesso principio, ancora una volta affrontando la piccola difficoltà che abbiamo dovuto aggirare nella codifica. Sia $k = \left\lfloor \frac{m}{n!} \right\rfloor$ il risultato della divisione intera tra m e $n!$, allora possiamo scrivere d come segue

$$\begin{array}{rcl} d(m,n) & = & k \cdot d'(d(m - k(n-1)!, n-1), k) \\ d(m,0) & = & \langle\,\rangle \end{array}$$

dove, ovviamente, $d' : \mathrm{Perm}[[n-1]] \mapsto \mathrm{Perm}[[n] \setminus \{k\}]$ è l'inversa di c':

$$\begin{array}{rcll} d'(m \cdot s, k) & = & (m+1) \cdot d'(s,k) & \text{se } m \geq k \\ d'(m \cdot s, k) & = & m \cdot d'(s,k) & \text{se } m < k \\ d'(\langle\,\rangle) & = & \langle\,\rangle & \end{array}$$

3.2 Esercizi e Spunti di Riflessione

1. Provare a definire le usuali funzioni aritmetiche (somma, differenza, prodotto, test di minore o uguale etc.) sui numeri interi codificati come in Sezione 3. Sfruttare ove possibile le funzioni scritte per i numeri naturali in notazione unaria.
2. ★ Come è possibile codificare l'insieme dei numeri razionali con numeri naturali? (non preoccuparsi del fatto che un numero razionale può avere più rappresentazioni come coppia di numeri naturali)
3. Progettare una codifica bieetiva dell'insieme delle parti di un insieme X nel sottoinsieme dei numeri naturali $[2^{|X|}]$. Scrivere due programmi C che codificano una sequenza in un numero naturale e uno che decodifica un numero naturale producendo in output un sottoinsieme di X (anche qui conviene, per semplicità risolvere il problema quando $X = [n]$).
4. Progettare una codifica bieetiva dell'insieme delle partizioni di un numero naturale n. Scrivere due programmi C che codificano una partizione in un numero naturale e uno che decodifica un numero naturale producendo in output una partizione.
5. Una possibile codifica (o *rappresentazione* di un sottoinsieme di X è attraverso il suo vettore caratteristico. Se $X = \{x_1, x_2, \ldots, x_n\}$ ogni sottoinsieme $Y \subseteq X$ può essere codificato con n bits $y_1, y_2, \ldots, y_n$ tale che $y_i = 1$ se e solo se $x_i \in Y$. In che modo questa rappresentazione potrebbe essere sfruttata per risolvere il problema precedente? In che ordine vengono enumerati i sottoinsiemi di X da questa codifica?
6. Data la rappresentazione dei sottoinsiemi di un certo insieme X come vettori caratteristici, come si possono implementare le operazioni di unione, intersezione, complementazione ($\overline{Y} = X \setminus Y$), test di uguaglianza, inclusione etc.?