Algoritmi ricorsivi

Lezione 19 di Programmazione 2

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2006-7

Indice

primi esempi

definizione ricorsiva di una funzione per il calcolo del **fattoriale:**

```
int fattoriale(int N)
{ if ( N == 0 ) return 1; return N*fattoriale(N-1); }
```

definizione ricorsiva, ma **non induttiva**, di una funzione per il **problema di Collatz:**

```
int collatz(int N)
{ if ( N == 1 ) return 1;
  if ( N%2 == 0 ) return collatz(N/2); return collatz((3*N+1)/2); }
```

N.B. si **congettura** che questa funzione restituisca 1 per qualsiasi valore dell'argomento ...

per la definizione precedente, definiamo una funzione che ne misuri la **profondità della ricorsione**:

```
int cd(int N, int& d)
{ if ( N == 1 ) return d;
  d++; if ( N%2 == 0 ) return cd(N/2, d); return cd((3*N+1)/2, d); }
int collatzd(int N) { int d = 1, return cd(N, d); }
```

algoritmo di Euclide per il calcolo del massimo comun divisore di due numeri:

```
int gcd(int m, int n) { if ( n == 0 ) return m; return gcd(n, m%n); }
```

ricorsione o iterazione?

è sempre possibile trasformare una definizione ricorsiva di funzione in una equivalente non ricorsiva, usando opportunamente l'iterazione (v. esercizi)

ad esempio, il calcolo del fattoriale può essere codificato come segue:

```
int fattoriale(int N)
{ for (int t = 1, i = 1; i <= N; i++) t *= i; return t; }
```

viceversa, è sempre possibile trasformare una definizione di funzione che abbia cicli in una equivalente ricorsiva priva di cicli

abbiamo dunque due **stili di programmazione:** quale preferire?

la ricorsione dà soluzioni più **eleganti**, di correttezza più facilmente dimostrabile (quando basata sull'induzione), ma potenzialmente meno efficienti e limitate in pratica dalla profondità della ricorsione

l'iterazione non ha questi limiti: può essere utilmente impiegata in seconda battuta, per trasformare algoritmi ricorsivi in algoritmi più efficienti (quando serve)

valutazione di espressioni prefisse

la definizione ricorsiva della funzione di valutazione fornisce un esempio di **analisi sintattica** (ingl. parsing) per **discesa ricorsiva**

il primo parametro della funzione è un puntatore all'intera espressione da valutare; il secondo parametro, passato per riferimento, è l'indice alla posizione iniziale della sua sottostringa da valutare in ciascuna invocazione ricorsiva

```
int eval(char* a, int& i)
{ int x = 0;
  while (a[i] == ' ') i++;
  if (a[i] == '+') { i++; return eval(a, i) + eval(a, i); }
  if (a[i] == '-') { i++; return eval(a, i) - eval(a, i); }
  if (a[i] == '*') { i++; return eval(a, i) * eval(a, i); }
  if (a[i] == '/') { i++; return eval(a, i) / eval(a, i); }
  if (a[i] == '%') { i++; return eval(a, i) % eval(a, i); }
  while ((a[i] >= '0') && (a[i] <= '9'))
    x = 10*x + (a[i++]-'0');
  return x;
}
int preval(char* a) { int i = 0; return eval(a, i); }
```

funzioni ricorsive per liste concatenate

si ha **ricorsione in coda** (ingl. tail recursion) quando un'invocazione ricorsiva è presente nell'ultima istruzione di una definizione ricorsiva di funzione

la ricorsione in coda è facilmente eliminabile mediante l'iterazione

tre delle seguenti definizioni sono ricorsive in coda, mentre non lo è la definizione di `traverseR`, per l'attraversamento in ordine inverso di una lista concatenata

```
int count(nlink x)
{ if (x == 0) return 0; return 1 + count(x->next); }
```

```
void traverse(nlink h, void visit(nlink))
{ if (h == 0) return; visit(h); traverse(h->next, visit); }
```

```
void traverseR(nlink h, void visit(nlink))
{ if (h == 0) return; traverseR(h->next, visit); visit(h); }
```

```
void remove(nlink& x, Elem e)
{ while (x != 0 && x->elem == e)
  { nlink t = x; x = x->next; delete t; }
  if (x != 0) remove(x->next, e); }
```

algoritmi *divide et impera*

si è già invocata la regola divide et impera, per caratterizzare una vasta classe di algoritmi, quando si sono introdotte le relazioni di ricorrenza

appare ora chiaro che gli algoritmi di questa classe hanno definizioni ricorsive basate sull'**induzione**

per questi algoritmi, la struttura ricorsiva del programma fornisce lo schema essenziale

- sia per una dimostrazione induttiva di correttezza
- sia per una valutazione analitica delle prestazioni

riguardo al secondo aspetto, si dimostrano spesso per induzione proprietà di sottoclassi di algoritmi divide et impera

eccone una, a titolo di esempio:

- una funzione ricorsiva che, quando l'input ha dimensione **N > 2**, lo divide in due parti indipendenti non vuote e si invoca ricorsivamente su ciascuna di esse, ha profondità di ricorsione minore di **N**

esempio: ricerca del massimo

l'approccio divide et impera fornisce spesso una soluzione **rapida** al problema, in termini di tempo necessario a trovarla e/o codificarla

ecco ad esempio una soluzione divide et impera al problema di trovare il massimo in un insieme di elementi

- gli elementi sono memorizzati in un array, e si assume che siano di un tipo di prima categoria su cui sia definito l'operatore booleano >

```
Elem max(Elem a[], int l, int r)
{ if (l == r) return a[l];
  int m = (l+r)/2;
  Elem u = max(a, l, m);
  Elem v = max(a, m+1, r);
  if (u > v) return u; else return v; }
```

per il risultato enunciato in precedenza, questo algoritmo ha profondità di ricorsione inferiore al numero **N** di elementi nell'array

l'analisi dell'algoritmo permette una valutazione più accurata della profondità di ricorsione, che è all'incirca **lg N**

problema delle torri di Hanoi

dati: 3 pioli e N dischi, tutti di diverso diametro, inizialmente posti tutti in un piolo, per diametri decrescenti dalla base alla sommità

problema: trasferire la pila di dischi su un altro piolo, spostando solo un disco alla volta e senza mai porre un disco su un altro di diametro minore

soluzione ricorsiva: (il segno di) `d` rappresenta la direzione "circolare" dello spostamento, la sequenza di invocazioni a `muovi` generata dall'algoritmo rappresenta la soluzione

```
void hanoi(int N, int d)
{ if (N == 0) return;
  hanoi(N-1, -d); muovi(N, d); hanoi(N-1, -d);
}
```

profondità della ricorsione: N, numero totale di invocazioni ricorsive: 2^N-1

è facile convincersi della correttezza dell'algoritmo ...

... non altrettanto facile è trovarne uno equivalente iterativo (non ricorsivo)!

esercizi

1. dare definizioni non ricorsive delle funzioni `collatz` e `gcd` equivalenti alle rispettive definizioni ricorsive qui proposte
2. v. esercizio 5.1 del testo (Sedgewick, 2003), a p. 208: dare una definizione ricorsiva della funzione `lg(N!)` che riesca a produrre il risultato anche per valori di `N` abbastanza grandi da rendere `N!` non rappresentabile
3. v. esercizio 5.2 del testo (Sedgewick, 2003), a p. 208: dare una definizione ricorsiva della funzione `N! mod M` che produca il risultato per qualsiasi valore rappresentabile di `N`, dunque anche per valori abbastanza grandi da rendere `N!` non rappresentabile
4. la Proprietà 5.1 nel testo (Sedgewick, 2003), a p. 212, recita: "una funzione ricorsiva che divide un problema di dimensione N in due parti indipendenti (non vuote) che risolve in modo ricorsivo, invoca se stessa per meno di N volte"; verificare che tale formulazione è erronea

   la dimostrazione per induzione fornita dal testo è basata sulla seguente ricorrenza, dove T_N è il numero totale di chiamate ricorsive su un input di dimensione N, e si ipotizza che le due parti abbiano rispettivamente dimensione k e N-k: trovare in questa ricorrenza la fonte dell'errore nel testo

   $T_N = T_k + T_{N-k} + 1$, per $N > 1$, dove $T_1 = 0$