Algoritmi ricorsivi e programmazione dinamica

Lezione 20 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

problema del righello

problema: dividere un righello in 2^n parti di eguale lunghezza, marcandolo con una tacca di altezza **n** al centro, quindi con una tacca di altezza **n-1** al centro di ciascuna delle sue due metà, una tacca di altezza **n-2** al centro di ciascuno dei suoi quattro quarti, etc., fino alle tacche di altezza 1 al centro di ciascuna delle sue 2^{n-1} parti di eguale lunghezza

ecco una soluzione ricorsiva:

da invocare inizialmente con l=0, r=2^n, h=**n**, rispetta l' **invariante:** $r - l = 2^h$

```
void righello(int l, int r, int h)
{ int m = (l+r)/2;
  if (h > 0)
  { righello(l, m, h-1); marca(m, h); righello(m, r, h-1); }
}
```

è evidente una certa analogia formale di questa soluzione con la soluzione ricorsiva del problema delle torri di Hanoi ...

questo ci aiuterà a trovare un algoritmo **non** ricorsivo per il problema delle torri di Hanoi!

torri di Hanoi e righello

osserviamo innanzitutto che l'altezza h della m-sima tacca prodotta dall'algoritmo del righello ha lo stesso valore di N (diametro del disco) nella corrispondente mossa prodotta dall'algoritmo ricorsivo delle torri di Hanoi:

per m dispari: h = 1 (disco più piccolo nelle torri di Hanoi)
per m pari: h > 1 (non il disco più piccolo nelle torri di Hanoi)

più precisamente, il valore di **h** nell'invocazione di `marca(m, h)` è uguale a
1 + il numero di bit 0 in coda alla rappresentazione binaria naturale del valore di m

un algoritmo iterativo per il problema delle torri di Hanoi, con una pila di **n** dischi da spostare nel piolo accanto in direzione **d**, consta nell'iterare la seguente coppia di mosse, fino al conseguimento dell'obiettivo (v. inoltre l'esercizio 1):

1. sposta il disco più piccolo in direzione **d** se **n** è dispari, opposta se **n** è pari
2. esegui l'unica mossa consentita che non sposti il disco più piccolo

la correttezza di questo algoritmo si può facilmente verificare dimostrando per induzione su **n** che, nell'algoritmo ricorsivo delle torri di Hanoi:

una mossa ogni due sposta il disco più piccolo
si inizia e si termina spostando il disco più piccolo
quest'ultimo è sempre spostato nella stessa direzione, per un dato **n**, la quale dipende solo dalla parità di **n**: **d** se **n** è dispari, **-d** se **n** è pari

programmazione dinamica

negli algoritmi ricorsivi visti finora, l'efficienza dell'approccio divide et impera è stata garantita dal fatto che l'algoritmo partiziona il problema in istanze più piccole e **indipendenti** dello stesso problema, vale a dire operanti su parti disgiunte dell'input o della struttura dati in gioco

algoritmi ricorsivi che non soddisfano tale condizione di indipendenza possono avere costi computazionali proibitivi

ad esempio, l'algoritmo per il calcolo dei **numeri di Fibonacci** derivato direttamente dalla loro definizione ricorsiva richiede tempo **esponenziale**:

```
int F(int i)
{ if (i < 1) return 0; if (i == 1) return 1; return F(i-1) + F(i-2); }
```

l'inefficienza dell'algoritmo è dovuta al ripetersi di invocazioni ricorsive con ugual valore del parametro: il numero totale di invocazioni ricorsive per il calcolo di F_N, per $N > 1$, è maggiore di F_{N+1} (v. esercizio 2), che è proporzionale a φ^N (v. lezione 4)

> le tecniche di **programmazione dinamica** prendono le mosse da tal sorta di algoritmi ricorsivi, dove le parti in cui si decompone il problema non sono indipendenti, e hanno il merito di **eliminare la reiterazione superflua** di calcoli già effettuati

vediamo come ...

programmazione dinamica bottom-up

per valutare **in tempo lineare** una funzione definita induttivamente sui numeri naturali si può calcolarla per valori crescenti dell'argomento, a partire dal più piccolo e fino a quello desiderato, memorizzandone i valori in un array

> questa tecnica prende il nome di **programmazione dinamica bottom-up**: con essa, il calcolo di una funzione ricorsiva viene effettuato mediante un algoritmo iterativo

ecco ad es. come si realizza il calcolo dei primi N numeri di Fibonacci con questa tecnica:

```
F[0] = 0; F[1] = 1; for (i = 2; i <= N; i++) F[i] = F[i-1] + F[i-2];
```

> per il calcolo di F_N non è necessario memorizzare tutti i numeri di Fibonacci precedenti: bastano gli ultimi due; in questo caso si può dunque far uso di due variabili invece di un array (v. esercizio 3)

la programmazione dinamica bottom-up è efficace nel prevenire la reiterazione superflua di calcoli già effettuati, tuttavia con essa:

> si calcolano (e memorizzano) i valori della funzione per **tutti** i valori dell'argomento più piccoli di quello desiderato: non sempre ciò è necessario
> si trasforma la ricorsione in iterazione: neanche questo è necessario al fine di eliminare la reiterazione superflua di invocazioni ricorsive

una diversa tecnica di programmazione dinamica è la seguente

programmazione dinamica top-down

anziché eliminare le invocazioni ricorsive, si trasforma una definizione ricorsiva di funzione così che:

- alla fine del calcolo, il valore calcolato venga memorizzato prima di essere restituito
- all'inizio dell'esecuzione si controlli se il calcolo sia stato già effettuato per il dato valore del parametro: in tal caso si restituisce il valore memorizzato (così arrestando la ricorsione)

questa tecnica prende il nome di **programmazione dinamica top-down**: con essa, una definizione ricorsiva di funzione per induzione naturale può venire "automaticamente" trasformata in una definizione in cui il numero di invocazioni ricorsive è al più pari al valore dell'argomento

- il costo computazionale è al più pari a quello dell'approccio bottom-up

ecco ad es. la trasformazione della funzione di calcolo dei numeri di Fibonacci secondo questa tecnica:

```
int F(int i)
{ static int notoF[maxN]; if (notoF[i] != 0) return notoF[i];
  int t = i;  if (i < 0) return 0; if (i > 1) t = F(i-1) + F(i-2); return notoF[i] = t; }
```

alla struttura dell'algoritmo ricorsivo (inefficiente) visto prima, quello trasformato aggiunge:

- la dichiarazione dell'array statico, i cui contenuti sono inizializzati a 0 in C++, nel quale si memorizzano i valori calcolati della funzione
- il controllo iniziale sulla presenza del valore richiesto nell'array, per l'arresto della ricorsione
- la memorizzazione finale nell'array del nuovo valore calcolato, prima della sua restituzione

problema dello zaino

problema di **ottimizzazione combinatoria**, in diverse varianti: eccone la più semplice

dati: uno zaino di capacità **M**
N tipi di oggetti, ciascuno caratterizzato dal **valore** e dal **volume** occupato da ogni oggetto di quel tipo

problema:

determinare il massimo valore totale di un multinsieme di oggetti il cui volume totale non superi la capacità dello zaino

N.B. si suppone disponibile una quantità **illimitata** di oggetti di ciascun tipo

ovvia **variante** del problema: determinare il multinsieme di oggetti che massimizza il valore totale, con lo stesso vincolo sul volume totale

N.B. essenzialmente lo stesso problema, ha la stessa complessità

varianti **più complesse:**

- disponibilità limitata degli oggetti di ciascun tipo
- più zaini, della stessa o di diverse capacità
- combinazioni delle precedenti

soluzione ricorsiva inefficiente

si assuma senza perdita di generalità che il volume di ogni oggetto non superi la capacità dello zaino

specifica di soluzione ricorsiva: per ogni tipo **i** di oggetti, con $0 \le i < N$, siano

- val_i : il valore di un oggetto del tipo **i**
- vol_i : il volume occupato da un oggetto del tipo **i**
- z_i : la soluzione dell'istanza del problema dello zaino con gli stessi tipi di oggetti ma capacità (residua) $M - vol_i$

allora la soluzione del problema è data da: $z = \max \{ val_i + z_i \mid 0 \le i < N \}$

la specifica si traduce nel seguente algoritmo ricorsivo in C++, dove **M** sarà il parametro attuale nella prima invocazione della funzione ricorsiva:

```
typedef struct { int vol; int val; } Tipo;
const int N = 10, M = 1000; Tipo t[N];
int zaino(int c)
{ int i, r, max, v;
  for (i = 0, max = 0, i < N; i++)
    if ((r = c - t[i].vol) >= 0)
      if ((v = t[i].val + zaino(r)) > max) max = v;
  return max;
}
```

la possibilità di multiple invocazioni ricorsive con lo stesso valore del parametro è fonte di inefficienza

soluzione con programmazione dinamica

la trasformazione della soluzione ricorsiva inefficiente del problema dello zaino in una ricorsiva efficiente è immediata con la tecnica della programmazione dinamica top-down:

```
typedef struct { int vol; int val; } Tipo;
const int N = 10, M = 1000; Tipo t[N]; static int notoZ[M];
int zaino(int c)
{ int i, r, max, v;
  if (notoZ[i] != 0) return notoZ[i];
  for (i = 0, max = 0, i < N; i++)
    if ((r = c - t[i].vol) >= 0)
      if ((v = t[i].val + zaino(r)) > max) max = v;
  return notoZ[c] = max;
}
```

diversamente dal problema del calcolo dei numeri di Fibonacci, la soluzione ricorsiva del problema dello zaino per un dato valore del parametro **non** richiede la soluzione dello stesso problema **per tutti** i precedenti valori del parametro, ma **solo per alcuni** di essi

la programmazione dinamica top-down è dunque **più efficiente** di quella bottom-up, per questo problema

programmazione dinamica e prestazioni

se è **costante** il costo computazionale dell'esecuzione di una invocazione ricorsiva, detratto quello delle invocazioni ricorsive che essa eventualmente effettua, allora il costo computazionale di una funzione ricorsiva con argomento intero naturale, definita con una tecnica di programmazione dinamica, è **lineare** rispetto all'argomento

inoltre, come mostrato dalla variante considerata del problema dello zaino, la programmazione dinamica top-down ha un costo computazionale inferiore a quello della programmazione dinamica bottom-up quando la ricorsione non coinvolge tutti i valori dell'argomento inferiori a quello dato

varianti più complesse del problema dello zaino rivelano un **altro vantaggio dell'approccio top-down** rispetto a quello bottom-up:

> in tali varianti, la definizione ricorsiva dipende da **più argomenti**, ad es. una **k**-pla di capacità residue nella variante con **k** zaini
>
> l'ordine lineare naturale che si ha nel caso di un solo argomento diventa un ordine **parziale** quando gli argomenti sono più d'uno, e può non essere ovvio stabilire un ordine appropriato di valutazione della funzione per i valori inferiori degli argomenti
>
> nella programmazione dinamica top-down, poiché si preserva la struttura ricorsiva naturale di una semplice (ma inefficiente) soluzione al problema, trasformandola **meccanicamente** in una efficiente, l'**appropriatezza dell'ordine di valutazione** è implicitamente garantita

esercizi

1. sviluppare un algoritmo iterativo per il problema delle torri di Hanoi, in cui si rappresenti lo stato del sistema con un array di 3 pile di interi (i dischi, ciascuno rappresentato dal suo diametro), indicizzato dai pioli, e lo spostamento di un disco da un piolo ad un altro venga effettuato mediante operazioni standard dell'ADT pila sulle corrispondenti pile di dischi
2. il testo (Sedgewick, 2003), a p. 223, sostiene che il numero totale di invocazioni ricorsive per il calcolo dell'N-simo numero di Fibonacci F_N è F_{N+1}: mostrare che, invece, tale numero è maggiore di F_{N+1} per $N > 1$, e in verità maggiore di F_{N+2} per $N > 3$
3. esercizio 5.37 del testo (Sedgewick, 2003), p. 230: definire una funzione C++ che calcoli F_N mod M usando una quantità di spazio costante, dove F_N è l'N-simo numero di Fibonacci
4. specificare una soluzione ricorsiva della variante del problema dello zaino in cui si richiede di determinare il multinsieme di oggetti che massimizza il valore del contenuto dello zaino, e implementarla con la tecnica della programmazione dinamica top-down

   > **suggerimento:** considerare diverse alternative per la struttura dati che rappresenta il multinsieme contenuto nello zaino, ad es. pila di oggetti, array int[N] di rappresentazione della molteplicità per ciascun tipo di oggetti, e scegliere quella che sembra permettere la più semplice formulazione della specifica della soluzione ricorsiva; sarà anche quella che produce l'implementazione più efficiente con la tecnica della programmazione dinamica top-down?
5. confrontare l'efficienza della soluzione sviluppata nell'esercizio precedente con quella della soluzione proposta dal testo (Sedgewick, 2003), a p. 229, Programma 5.13, anch'essa basata sulla programmazione dinamica top-down, ma con una diversa rappresentazione del contenuto dello zaino