7 Programmazione dinamica

Presentiamo alcuni esempi concreti che illustrano la tecnica della programmazione dinamica. Tale tecnica si puo' applicare ogni volta che un problema puo' essere riformulato mediante una definizione ricorsiva (stile divide-et-impera). La differenza essenziale tra un algoritmo di programmazione dinamica e quello divide-et-impera (indotti dalla stessa definizione ricorsiva del problema), consiste nell'evitare di ricalcolare la soluzione dei sottoproblemi gia' risolti. Quindi se un sottoproblema viene ricalcolato molte volte da un algoritmo divide-et-impera, si puo' ridurre drasticamente il tempo di calcolo (questo puo' pero' richiedere di pagare una penale in termini di spazio necessario per memorizzare le soluzioni ai sottoproblemi). Un modo semplice di analizzare i trade-offs tra le due tecniche (programmazione dinamica e divide-et-impera) e' di considerare il grafo orientato delle chiamate ricorsive: i nodi sono le istanze del problema (cioe' i possibili input $x \in I$) e gli archi vanno da un problema ai suoi sottoproblemi immediati, cioe' quelli menzionati espicitamente nella definizione ricorsiva (quindi i casi base non hanno archi uscenti). Se la definizione ricorsiva e' ben fondata, il grafo delle chiamate ricorsive e' aciclico.

- per risolvere il problema x con il divide-et-impera si ricalcola la soluzione di un sottoproblema x' tante volte quanti sono i cammini da x a x' nel grafo delle chiamate ricorsive
- per risolvere il problema x con la programmazione dinamica si calcola la soluzione di un sottoproblema x' al piu' una volta.

Quindi se il grafo e' un albero, la programmazione dinamica non fornisce alcun vantaggio rispetto al divide-et-impera. La successione di Fibonacci.

- f(0) = f(1) = 1
- f(i) = f(i-1) + f(i-2)

fornisce un semplice esempio per illustrare i vantaggi della programmazione dinamica rispetto al divide-et-impera:

- l'algoritmo divide-et-impera (corrispondente alla definizione ricorsiva di $f(i)$) per calcolare $f(n)$ ricalcolera' $f(n-i)$ volte il valore $f(i)$, e poiche' $f(2i) \geq 2^i$ il tempo di calcolo sara' esponenziale in n.
- l'algoritmo di programmazione dinamica invece richiede tempo e spazio lineari in n.
- facendo un'analisi piu' accurata del grafo delle chiamate ricorsive, si vede che non e' necessario mantenere tutti i valori di $f(i)$ per $i \leq n$, basta conservare gli ultimi due, quindi si richiede solo uno spazio costante e tempo lineare.

In sostanza si riottiene l'algoritmo iterativo efficiente, applicando le idee della programmazione dinamica ed ottimizzando l'uso dello spazio sulla base del grafo delle chiamate ricorsive.

7.1 Coefficienti binomiali

Definiamo $b(j,k)$ per $j,k \geq 0$ come il coefficiente binomiale $\binom{j+k}{k} \triangleq \frac{(j+k)!}{j! * k!}$. La definizione ricorsiva dei coefficienti binomiali puo' essere riscritta in termini di $b(j,k)$ come segue

- $b(0,k) = b(j,0) = 1$ per $j,k \geq 0$
- $b(j,k) = b(j-1,k) + b(j,k-1)$ per $j,k > 0$

Quindi per calcolarci il valore $b(j,k)$ si dovranno calcolare i coefficienti per le coppie (x,y) nel rettangolo $[0..j] \times [0..k]$. Diamo un algoritmo bottom-up che riempie una matrice di interi positivi compatibilmente con la definizione ricorsiva di b, cioe' quando si riempie la posizione (x,y), con $x,y > 0$, devono essere gia' state riempite le posizioni $(x-1,y)$ e $(x,y-1)$.

```
var B: array [0..j,0..k] of integer
*si riempie B per righe crescenti e
*   all'interno della stessa riga per colonna crescente
for y <- 0 to k do B[0,y] <- 1
for x <- 1 to j do
  B[x,0] <- 1
  for y <- 1 to k do
    B[x,y] <- B[x-1,y] + B[x,y-1]
```

Si poteva procedere anche per colonne crescenti, e all'interno della stessa colonna per riga crescente.

7.2 Il problema dello zaino

Il *problema decisionale* dello zaino puo' essere formulato come segue:

- input: k; $k_1, \ldots, k_n$ (numeri naturali)
- output: esiste un sottoinsieme $J \subseteq \{1, \ldots, n\}$ t.c. $k = \sum_{j \in J} k_j$

Per risolverlo con le tecniche della programmazione dinamica occorre riformularlo in termini di una definizione ricorsiva. Definiamo il predicato $P(j,h)$ su $[0..n] \times [0..k]$ come $P(j,h)$ sse il problema dello zaino con input $(h; k_1, ..., k_j)$ e' risolubile. Quindi il problema che vogliamo risolvere equivale al calcolo di $P(n,k)$.
A questo punto possiamo dare una definizione ricorsiva di $P(j,h)$:

- $P(j,0) = true$, per ogni $j \geq 0$
- $P(0,h) = false$, per ogni $h > 0$
- $P(j,h)$ sse $P(j-1,h) \vee (h \geq k_j \wedge P(j-1, h-k_j))$, per ogni $j,h > 0$

Diamo un algoritmo bottom-up che riempie una matrice di boolean compatibilmente con la definizione ricorsiva di P, quindi nel caso peggiore quando si riempie la posizione (j,h), con $j,h > 0$, devono essere gia' state riempite le posizioni $(j-1,h)$ e $(j-1,h-k_j)$.

```
var P: array [0..n,0..k] of boolean
*si riempie P per righe crescenti e
*   all'interno della stessa riga per colonna crescente
*   (ma si poteva procedere in qualsiasi ordine)
P[0,0] <- true
for y <- 1 to k do P[0,y] <- false
for x <- 1 to n do
  P[x,0] <- true
  for y <- 1 to k do
*si utilizza un if-then-else in modo da
*evitare chiamate ricorsive inutili
    if P[x-1,y] then P[x,y] <- true
    elsif y < k_x then P[x,y] <- false
    else P[x,y] <- P[x-1,y-k_x]
```

Con il precedente algoritmo si tende a fare del lavoro in piu' di quello strettamente necessario, p.e. per calcolare $P(n,k)$ non abbiamo bisogno di calcolarci $P(n,h)$ per $h < k$.
Un metodo generale per evitare del lavoro inutile, e' di utilizzare un algoritmo demand-driven, che ricorda l'implementazione ricorsiva della visita DFS, ovvero si usa un array per tenere traccia di quali sotto-problemi sono stati sinora risolti (in analogia con quello dei nodi sinora visitati), e se un sotto-problema non e' stato risolto, si effettuano le stesse chiamate di un algoritmo divide-et-impera.

```
var P: array [0..n,0..k] of boolean+{NIL}
{NIL indica che P[x,y] non e' stato calcolato}
*inizializzazione di P: nessun valore e' stato calcolato
for x <- 0 to n do
  for y <- 0 to k do
    P[x,y] <- NIL
P(n,k)
```

$P(x,y)$ e' una procedura ricorsiva che usa come variabili globali P ed i k_j. Tale procedura puo' essere definita nel modo seguente.

```
P(x,y) =
if P[x,y]=NIL then if y = 0 then P[x,y] <- true
elsif x = 0 then P[x,y] <- false
elsif P(x-1,y) then P[x,y] <- true
elsif y < k_x then P[x,y] <- false
else P[x,y] <- P(x-1,y-k_x)
return(P[x,y])
```

In questo modo si calcola solo i $P[x,y]$ strettamente necessari per il calcolare $P[n,k]$. Poiche' dobbiamo inizializzare comunque P, la complessita' tempo e' $O(n*k)$ come l'algoritmo bottom-up. Tuttavia, se si usa una diversa rappresentazione dei sottoinsiemi di $[0..n] \times [0..k]$ si potrebbe ottenere dei vantaggi a livello di complessita' tempo.

Lo stack per le chiamate ricorsive non puo' crescere piu' di $O(n)$, che in genere sara' trascurabile rispetto allo spazio occupato da P.

E' abbastanza facile modificare i precedenti algoritmi che risolvono il problema decisionale dello zaino per restituire anche la soluzione, cioe' risolvere il *problema di ricerca*

- input: k; $k_1, \ldots, k_n$ (numeri naturali)
- output: un sottoinsieme $J \subseteq \{1, \ldots, n\}$ t.c. $k = \sum_{j \in J} k_j$ (se J esiste)

Il grafo delle chiamate ricorsive resta invariato, tuttavia oltre all'informazione booleana $P[j,h]$ su l'esistenza o meno della soluzione si dovra' fornire anche un sottoinsieme soluzione $I[j,h]$ ($I[j,h]$ non e' significativo se $P[j,h]$ e' falso).

```
var P: array [0..n,0..k] of boolean
var I: array [0..n,0..k] of sottoinsiemi of 1..n
P[0,0] <- true; I[0,0] <- vuoto
for y <- 1 to k do P[0,y] <- false
for x <- 1 to n do
  P[x,0] <- true; I[x,0] <- vuoto
  for y <- 1 to k do
    if P[x-1,y] then P[x,y] <- true; I[x,y] <- I[x-1,y]
    elsif y < k_x then P[x,y] <- false
    elsif P[x-1,y-k_x] then P[x,y] <- true; I[x,y] <- I[x-1,y-k_x] + {x}
    else P[x,y] <- false
```

Una riduzione nella occupazione di memoria si puo' ottenere sostituendo la tabella I con una tabella Q di booleani t.c. $Q[j,h]$ sse $j > 0$ e $j \in I[j,h]$. Infatti e' possibile ricostruire I a partire da P e Q sfruttando la seguente definizione ricorsiva

- $I[j,h]$ indefinito, se $P[j,h] = false$
- $I[j,h] = \emptyset$, se $h = 0$
- $I[j,h] = I[j-1, h-k_j] \cup \{j\}$, se $Q[j,h] = true$ e $h \geq k_j$
- $I[j,h] = I[j-1,h]$, se $Q[j,h] = false$

Quindi date le tabelle P e Q possiamo calcolare $I[j,h]$ in $O(j)$.

7.3 Moltiplicazione ottimale di matrici

Il *problema del valore ottimale* di moltiplicazioni scalari per calcolare la moltiplicazione n matrici rettangolari e' formulato come segue

- input: $(m_0, \ldots, m_n)$ (numeri naturali)
- output: numero ottimale di moltiplicazioni scalari per moltiplicare n matrici $M_1 * \ldots * M_n$ con M_i di dimensione $m_{i-1} \times m_i$

Definiamo ricorsivamente la funzione $MO(i,j)$ con $0 \leq i < j \leq n$ che da' il numero ottimale di moltiplicazioni scalari per moltiplicare le matrici $M_i * \ldots * M_j$

- $MO(i, i+1) = 0$, se $0 \leq i < n$
- $M(i,j) = \min_{i<k<j} (MO(i,k) + MO(k,j) + m_i * m_k * m_j)$, se $0 < i+1 < j \leq n$

Si ha un notevole riuso della soluzione ad un sotto-problema. Infatti per risolvere il problema che ci interessa, ovvero $MO(0,n)$ dobbiamo risolvere $O(n^2)$ sotto-problemi. L'approccio bottom-up parte dalla diagonale $\{(i,i+1) | 0 \leq i < n\}$ e procede per diagonali $\{(i,j) | j - i = k \wedge 0 \leq i,j \leq n\}$ con k crescente.

7.4 Altri esempi e raffinamenti

L'algoritmo di Floyd per calcolare la chiusura transitiva di una matrice e quello per trovare un cammino di lunghezza minima tra ogni coppia di nodi si possono vedere come esempi di programmazione dinamica, dove il grafo delle chiamate ricorsive puo' essere strutturato come un array tridimensionale.

Negli algoritmi di programmazione dinamica considerati sinora non ci siamo preoccupati di minimizzare il consumo di memoria a parita' di complessita' tempo. Una analisi piu' accurata della struttura del grafo delle chiamate ricorsive mostra che e' possibile ridurre drasticamente l'uso della memoria negli algoritmi bottom-up considerati:

- per calcolare i coefficienti binomiali basta mantenere una riga $B[x,_]$ (oppure una colonna $B[_,y]$) della tabella.
- per il problema dello zaino l'unica possibilita' e' mantenere una riga $P[x,_]$ della tabella.
- per calcolare la chiusura transitiva basta mantenere una matrice $M'[k,\_,\_]$ ed effettuare gli aggiornamenti nel seguente ordine: prima $M'[x,y]$ con x,y diversi da v_k, poi $M'[x,v_k]$ e $M'[v_k,y]$ con x,y diversi da v_k, ed infine $M'[v_k,v_k]$

```
input:  M: array [V,V] of A
output: M^+: array [V,V] of A
var: M': array [V,V] of A
M' <- M
for k <- 1 to |V| do
  for x,y in V-{v_k} do
    M'[x,y] <-  M'[x,y]+(M'[x,v_k]*(M'[v_k,v_k]^*)*M'[v_k,y])
  for x in V-{v_k} do
    M'[x,v_k] <-  M'[x,v_k]*(M'[v_k,v_k]^*)
  for y in V-{v_k} do
    M'[v_k,y] <- (M'[v_k,v_k]^*)*M'[v_k,y]
  M'[v_k,v_k] <- M'[v_k,v_k]*(M'[v_k,v_k]^*)
M^+ <- M'
```

Per gli algoritmi demand-driven e' molto piu' difficile capire quando un elemento della tabella non serve piu', poiche' il modo di riempire la tabella procede in modo alquanto irregolare.