Azzolini Riccardo 2019-05-21

Programmazione dinamica

1 Programmazione dinamica

Un uso poco accorto della ricorsione può portare alla ripetizione di calcoli già effettuati: adottando un approccio *top-down*, cioè partendo dal problema principale e passando di volta in volta al calcolo di problemi più piccoli, può capitare che uno stesso problema venga risolto un numero esponenziale di volte.

La soluzione è la **programmazione dinamica**, che si basa su un approccio *bottom-up*: si risolvono tutte le istanze richieste dal problema, partendo dalle più piccole e memorizzando i risultati, in modo da poterli riutilizzare per la risoluzione delle istanze di dimensioni superiori.

Con questa tecnica si ha spesso una drastica riduzione dei tempi di calcolo, a costo di un aumento (generalmente accettabile) della memoria utilizzata.

2 Esempio: numeri di Fibonacci

La successione di Fibonacci è definita dall'equazione di ricorrenza

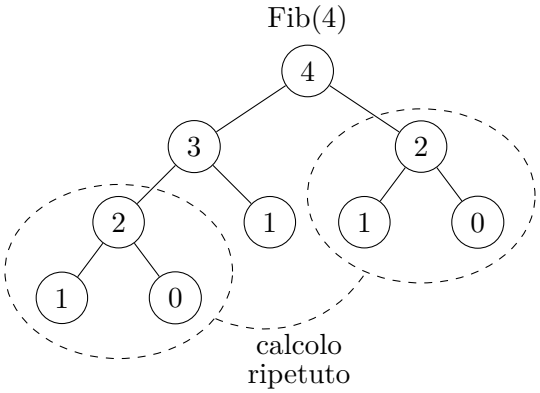
$$\text{Fib}(n) = \begin{cases} n & \text{se } n \leq 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{altrimenti} \end{cases}$$

2.1 Implementazione ricorsiva

L'implementazione ricorsiva di questa equazione è:

```
Fib(n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```

Le chiamate ricorsive necessarie per il calcolo di un numero di Fibonacci si possono rappresentare in un albero. Ad esempio, per Fib(4):



Siccome ogni chiamata esegue, oltre alle chiamate ricorsive, solo alcune operazioni a costo costante, il tempo di calcolo è proporzionale al numero di nodi dell'albero, e quest'ultimo è dato dall'equazione di ricorrenza

$$C(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 1 + C(n-1) + C(n-2) & \text{altrimenti} \end{cases}$$

Poiché $C(n) \geq \text{Fib}(n) \quad \forall n \in \mathbb{N}$ (si può dimostrare per induzione), e

$$\text{Fib}(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

si ha allora che

$$C(n) = \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

quindi questa soluzione richiede tempo di calcolo esponenziale.

2.2 Implementazione con la programmazione dinamica

Si calcolano i valori di Fib(k) per $k = 0, 1, \ldots, n$, salvando ciascun risultato in un vettore:

```
DFib(n) {
    V[0] = 0; V[1] = 1;
    for (k = 2; k <= n; k++)
        V[k] = V[k - 1] + V[k - 2];
```

```
    return V[n];
}
```

Questa soluzione richiede tempo $\Theta(n)$ e spazio $\Theta(n)$.

Lo spazio utilizzato si può ridurre osservando che $\text{Fib}(n)$ dipende solo dai due valori precedenti, quindi è sufficiente memorizzare questi:

```
DFib(n) {
    a = 0; b = 1;
    for (k = 2; k <= n; k++) {
        c = a + b;
        a = b;
        b = c;
    }
    return c;
}
```

In questo modo, il tempo di calcolo rimane $\Theta(n)$, ma la memoria occupata si riduce a $\Theta(1)$.

3 Metodo generale

Si ha un algoritmo ricorsivo descritto dalle procedure $P_1, \ldots, P_m$, dove P_1 è la procedura principale.

- $[P_k, x]$ indica la chiamata di P_k con input x.
- $[P_k, x]$ *dipende* da $[P_s, y]$ se l'esecuzione della procedura P_k con input x richiede almeno una volta la chiamata di P_s con input y, anche non direttamente (ad esempio, $[\text{Fib}, 4]$ dipende da $[\text{Fib}, 3]$, $[\text{Fib}, 2]$, $[\text{Fib}, 1]$ e $[\text{Fib}, 0]$). Le dipendenze formano quindi un grafo orientato e aciclico (se ci fossero cicli, l'algoritmo non terminerebbe).
- Il risultato dell'algoritmo sull'input z è calcolato dalla chiamata $[P_1, z]$.
- Si considera l'insieme di dipendenza

$$\mathbb{D}[P_1, z] = \{[P_s, y] \mid [P_1, z] \text{ dipende da } [P_s, y]\}$$

  Ad esempio:

$$\mathbb{D}[\text{Fib}, 4] = \{[\text{Fib}, 3], [\text{Fib}, 2], [\text{Fib}, 1], [\text{Fib}, 0]\}$$

- Si indica con $<$ l'ordine parziale su $\mathbb{D}[P_1, z]$ definito da

$$\forall [P_k, x], [P_s, y] \in \mathbb{D}[P_1, z], \quad [P_k, x] < [P_s, y] \iff [P_s, y] \text{ dipende da } [P_k, x]$$

- Si definisce un ordine totale $\prec$ compatibile con $<$.

Osservazioni:

- La programmazione dinamica *non* è una tecnica di progettazione di algoritmi: per sfruttarla, bisogna prima trovare un algoritmo ricorsivo, poi la si può applicare per realizzare un'implementazione efficiente.
- Le dipendenze, e quindi l'ordine parziale $<$, si ricavano dall'algoritmo ricorsivo, mentre l'ordine totale $\prec$ deve essere stabilito separatamente, determinando un (qualsiasi) modo per ordinare le chiamate indipendenti.

3.1 Struttura della procedura

1. Si definisce un ordine totale su $\mathbb{D}[P_1, z]$.
2. $i \coloneqq \min(\mathbb{D}[P_1, z])$
3. ripeti

   a) data $i = [P_j, x]$, esegui $P_j(x)$, interpretando

      - $b \coloneqq P_s(l)$ come $b \coloneqq V[P_s, l]$ (perché l'ordine totale garantisce che il risultato di $[P_s, l]$ sia già stato calcolato);
      - return E come $V[P_j, x] \coloneqq E$ per memorizzare il risultato calcolato (mentre esso verrebbe semplicemente restituito nell'algoritmo ricorsivo).

   b) $u \coloneqq i$; $i \coloneqq \text{Succ}(i)$
   (passa al prossimo problema nell'ordine totale, e memorizza in u qual è l'ultimo problema risolto);

   fino a quando $u = [P_1, z]$ (ciò indica infatti che è stata calcolata la soluzione del problema principale);
4. return $V[P_1, z]$

4 Esempio: moltiplicazione in cascata di matrici

- *Input*: m matrici $A_1, \ldots, A_m$, con A_i di ordine $r_{i-1} \times r_i$ (gli ordini possono essere rappresentati da un vettore $r = [r_0, r_1, \ldots, r_m]$).
- *Output*: il numero minimo di moltiplicazioni tra elementi necessarie per calcolare $A_1 \times A_2 \times \cdots \times A_m$ (le somme non si contano).

In generale, per moltiplicare due matrici di ordini $r \times s$ e $s \times t$ (ricavando una nuova matrice di ordine $r \times t$) servono $r \cdot s \cdot t$ moltiplicazioni tra elementi.

Si può sfruttare la proprietà associativa per ridurre il numero totale di moltiplicazioni. Ad esempio, per il calcolo di $A_1 \times A_2 \times A_3 \times A_4$, con $r = [2, 5, 3, 7, 4]$:

- $A_1 \times (A_2 \times (A_3 \times A_4))$ richiede $3 \cdot 7 \cdot 4 + 5 \cdot 3 \cdot 4 + 2 \cdot 5 \cdot 4 = 184$ moltiplicazioni;
- $((A_1 \times A_2) \times A_3) \times A_4$ richiede $2 \cdot 5 \cdot 3 + 2 \cdot 3 \cdot 7 + 2 \cdot 7 \cdot 4 = 128$ moltiplicazioni, che in questo caso è il numero minimo possibile.

4.1 Soluzione ricorsiva

Sia $M[k, s]$ il modo migliore di calcolare $A_k \times A_{k+1} \times \cdots \times A_{s-1} \times A_s$. Per ricavare la soluzione del problema, $M[1, m]$:

1. si seleziona la posizione dell'ultimo prodotto da eseguire (il più esterno, che rimane fuori da tutte le parentesi);
2. si trova (ricorsivamente) il modo migliore di moltiplicare le matrici a sinistra e a destra di tale prodotto;
3. si somma al risultato del punto 2 il numero di moltiplicazioni necessarie per l'ultimo prodotto.

L'equazione di ricorrenza corrispondente è:

$$M[k, s] = \begin{cases} 0 & \text{se } k = s \\ \min\limits_{k \le j < s} \{M[k, j] + M[j + 1, s] + r_{k-1} r_j r_s\} & \text{se } k \neq s \end{cases}$$

Un'implementazione diretta di questa soluzione non sarebbe di fatto utilizzabile, dato che richiederebbe tempo di calcolo esponenziale.

4.2 Implementazione con la programmazione dinamica

```
DCosto(m, r) {
    [k, s] = [1, 1];
    repeat {
        if (k == s) {
            V[k, s] = 0;
        } else {
            cmin = MAX_INT;
            for (j = k; j < s; j++) {
                C = V[k, j] + V[j + 1, s] + r[k - 1] * r[j] * r[s];
                if (C < cmin) cmin = C;
            }
            V[k, s] = cmin;
        }
        [k, s] = Succ(k, s);
    } until ([k, s] == [1, m + 1]);
    return V[1, m];
}
```

- Il codice `if (k == s) { ... } else { ... }` implementa i due casi dell'equazione di ricorrenza.
- La funzione `Succ` restituisce il prossimo problema nell'ordine totale.
- Il ciclo `repeat { ... } until ([k, s] == [1, m + 1])` termina quando il problema successivo è $[1, m+1]$ (che non esiste), cioè quando è appena stato risolto $[1, m]$.

4.2.1 Ordine totale

L'ordine parziale da rispettare è definito come

$$[k_1, s_1] < [k_2, s_2] \iff k_2 \leq k_1 \leq s_1 \leq s_2$$

(cioè $[k_1, s_1] < [k_2, s_2]$ se e solo se $[k_1, s_1]$ è compreso in $[k_2, s_2]$). Si può quindi definire, ad esempio, l'ordine totale

$$\begin{aligned}[k_1, s_1] \prec [k_2, s_2] \iff\ & s_1 - k_1 < s_2 - k_2 \\ & \vee\, (s_1 - k_1 = s_2 - k_2 \wedge s_1 < s_2)\end{aligned}$$

secondo il quale $[k_1, s_1] \prec [k_2, s_2]$ se $[k_1, s_1]$ è più piccolo (comprende meno matrici) di $[k_2, s_2]$ oppure, a parità di dimensione, se $s_1 < s_2$:

$$
\begin{aligned}
&[1,1], [2,2], \ldots, [m,m],\\
&[1,2], [2,3], \ldots, [m-1,m],\\
&[1,3], [2,4], \ldots, [m-2,m],\\
&\ldots, [1,m]
\end{aligned}
$$

La funzione `Succ` corrispondente è:

```
Succ(k, s) {
    if (s < m)
        return [k + 1, s + 1]; // Prossimo della stessa dimensione
    return [1, s - k + 2]; // Primo della dimensione successiva
}
```

4.2.2 Complessità

La complessità di quest'implementazione è:

- spazio $\Theta(m^2)$, per la matrice `V` dei risultati;
- tempo $\Theta(m^3)$, perché ci sono $\Theta(m^2)$ problemi (dato che corrispondono a coppie di numeri interi), e ciascuna iterazione risolve uno di questi impiegando tempo $O(n)$ (si può dimostrare che il costo complessivo risulta essere esattamente $\Theta(m^3)$, e non solo $O(m^3)$).