Esercitazione 8
Algorithmi e Strutture Dati (Informatica)
A.A 2015/2016

Tong Liu

May 9, 2016

Elementi fondamentali

Programmazione dinamica
Progettare la tabella per problema con sotto-struttura ottima. Usare l'approccio iterativo bottom-up che risolve prima i problemi piccoli e usare tali risultati per risolvere problemi di dimensione più grandi.

Tecnica Greedy
Ordina array secondo un critiero, itera su array, aggiungi l'elemento se non supera il volume.

Esercizi

Esercizio 1: Change making problem

Data un insieme di valori di monete $V_1, ..., V_n$ e un valore S. Determinare il numero minimo di monete necessarie per pagare una somma S. Utilizzare la programmazione dinamica.

Soluzione

Si suppone $T(i,k)$ che indica il numero di monete necessarie per soddisfare somma k e cardinalità di monete i. Se k è uguale a 0, nessuna moneta usiamo. Se non abbiamo nessuna moneta, dobbiamo usare ∞ monete per raggiungere $S > 0$ cioè nessuna soluzione, altrimenti scegliamo il minimo fra 2 casi: il numero di monete includendo i-esimo moneta opppure usare solo le monete fra le prime $i-1$, le considerazioni possono essere riassunte dalle seguenti relazioni di ricorrenza:

$$T(i,k) = \begin{cases} 0 & \text{se } k = 0 \\ \infty & \text{se } i = 0, k > 0 \\ \min\left(T(i-1,k), T(i,k-V_i)+1\right) & \text{altrimenti} \end{cases} \tag{1.1}$$

L'algoritmo è mostrato in Pseudocodice 1

Esercizio 2: Coin Change Problem

Data un insieme di monete $V_1, ..., V_n$ e un valore S. Determinare il numero di combinazione possibile per pagare una somma S senza dare il resto. Utilizzare la programmazione dinamica.

Soluzione

Poniamo $T(i,j)$ il numero di combinazioni per raggiungere somma j considerando i primi i monete. Quando la somma è 0, c'è una sola combinazione che è non usare nessuna moneta. Quando la somma è maggiore di zero e non abbiamo nessuna moneta, c'è 0 combinazione possible. Per l'i-esima moneta, il numero di combinazione è uguale al quello escludendo quella moneta più quello che la include, quindi otteniamo la seguente relazione:

$$T(i,j) = \begin{cases} 1 & \text{se } j = 0 \\ 0 & \text{se } i = 0, j > 0 \\ T(i-1,j) + T(i,j-V_i) & \text{altrimenti} \end{cases} \tag{1.2}$$

Pseudocodice 2
(Per calcolare il x dello Pseudocodice, per esempio, avendo 2,3,4 per ottenere 5, c'è solo 1 combinazione, avendo 2,3,4 per ottenere 9, il numero della combinazione è uguale al quello della combinazione per ottenere 5 (9-4) quindi sempre 1. Per calcolare y, è uno spostamento nella riga precedente della tabella)

Algorithm 1 Pseudocodice per minimo numero di monete

```
1:  procedure MINCOIN(Array V, Integer n, Integer S)
2:      Array T ← Array()
3:      for all i = 0 to n do
4:          T[i, 0] ← 0
5:      end for
6:      for k = 1 to S do
7:          T[0, k] ← ∞
8:      end for
9:      for all i = 1 to n do
10:         for k = 1 to S do
11:             if V_i = k then
12:                 T[i, k] = 1
13:             else if V_i > k then
14:                 T[i, k] = T[i − 1, k]
15:             else
16:                 % miglior scelta tra includere V_i ed escludere V_i
17:                 T[i, k] = min (T[i − 1, k], T[i, k − V_i] + 1)
18:             end if
19:         end for
20:     end for
21:     return T[n, S]
22: end procedure
```

Algorithm 2 Pseudocodice per contare soluzioni

```
1:  procedure MINCOIN(Array V, Integer n, Integer S)
2:      Array T ← Array()
3:      for all i = 0 to n do
4:          T[i, 0] ← 1
5:      end for
6:      for all j = 1 to S do
7:          T[0, j] ← 0
8:      end for
9:      for all i = 1 to n do
10:         for j = 1 to S do
11:             if V_i > j then
12:                 x ← 0                          ▷ soluzione includendo V_i
13:             else
14:                 x ← T[i, j − V_i]
15:             end if
16:             y ← T[i − 1, j]                    ▷ soluzione senza V_i
17:             T[i, j] = x + y                    ▷ somma totale
18:         end for
19:     end for
20:     return T[n, S]
21: end procedure
```

Esercizio 3: Longest Common Subsequence Problem

[17/2/2015] Siano date due stringhe $P = p_1 p_2 ... p_m$ e $T = t_1 t_2 ... t_n$ di caratteri alfabetici. Si progetti un algoritmo di "programmazione dinamica" per individuare la più lunga sottosequenza comune tra P e T (per esempio, se $P = 9,15,3,6,4,2,5,10,3$ e $T = 8,15,6,7,9,2,11,3,1$ allora il risultato è: $15,6,2,3$).

Soluzione

Il problema in esercizio essenzialmente è il problema della sottosequenza comune massinale(LCS), dove abbiamo $D(i, j)$ per rappresentare la lunghezza della LCS di p_i e t_j quale ha una seguente relazione di ricorrenza:

$$D(i,j) = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ D(i-1, j-1) + 1 & \text{se } p_i = t_j \\ \max(D(i-1, j), D(i, j-1)) & \text{altrimenti} \end{cases} \tag{1.3}$$

Denotiamo $C(i, j)$ come gli elementi esatti della LCS, e usiamo operazione del vettore len() per contare il numero di elementi, quindi $D(i, j) = \text{len}(C(i, j))$. Con queste modifiche otterremo la soluzione in Pseudocodice 3. Il ciclo for annidato è la parte dominante dell'algoritmo che è $O(mn)$

Algorithm 3 LCS

```
1:  procedure LCSP(Array P, Array T, Integer m,Integer n)
2:      Array C ← Array(0...m, 0...n)
3:      for all i = 0 to m do
4:          C[i,0] ← 0
5:      end for
6:      for j = 1 to n do
7:          C[0, j] ← 0
8:      end for
9:      for i = 1 to m do
10:         for j = 1 to n do
11:             if P[i] = T[j] then
12:                 C[i, j] ← C[i − 1, j − 1].append(P[i])
13:             else
14:                 C[i, j] ←max (len(C[i, j − 1]), len(C[i − 1, j]))
15:             end if
16:         end for
17:     end for
18:     return C[m, n]
19: end procedure
```

Per la versione originale del LCS si riferisce alla pagina 263 del libro.

Esercizio 4

[20/9/2011] Un esame scritto contiene n esercizi, ognuno dei quali vale $p[i]$ punti e richiede $t[i]$, minuti per essere risolto, ed è superato raggiungendo la sufficienza S. Si vuole trovare il sottoinsieme di esercizi che permette di passare l'esame nel minor tempo possibile.
Posto $T(i,v)$ il numero minimo di minuti richiesti per ottenere almeno v punti risolvendo un qualunque sottoinsieme dei primi i esercizi, si scriva un'espressione ricorsiva per $T(i,v)$;
Si progetti un algoritmo basato sulla programmazione dinamica e se ne valuti la complessità.

Soluzione

Definiamo la funzione $T(i,v)$ come il seguente, dove $T(i,v)$ assume valore 0 quando il minimo voto richiesto è 0, ∞ quando non ci sono esercizi da svolgere. E per l'$i$esimo esercizio, si considerano 2 casi: risolverlo opppure risolvere i primi solo tra $i-1$.

$$T(i,v) = \begin{cases} 0 & \text{se } v = 0 \\ \infty & \text{se } i = 0, v > 0 \\ \min\left(T(i-1,v), T(i-1,v-p[i]) + t[i]\right) & \text{altrimenti} \end{cases} \tag{1.4}$$

L'algorithmo è descritto nello Pseudocodice 4 che ha una complessità $O(nS)$ dove n è il numero di esercizi e S è il voto richiesto per superare la prova.

Algorithm 4 minimo sforzo per superare l'esame

```
1:  procedure MINIMO-SFORZO(integer[] p, integer[] t, integer n, integer S)
2:      integer[][] T ← new integer[0...n][1...S]
3:      for i = 0 to n do
4:          T[i,0] ← 0
5:      end for
6:      for v = 1 to S do
7:          T[0,v] ← +∞
8:      end for
9:      for i = 1 to n do
10:         for v = 1 to S do
11:             Integer t' ← iif(p_i < v, T[i−1, v−p_i], 0)
12:             T[i,v] ← min(T[i−1,v], t' + t_i)
13:         end for
14:     end for
15:     return T[n,S]
16: end procedure
```

Esercizio 5

Risolvere il problema dell'esercizio precedente usando la tecnica Greedy.

Soluzione

Un approccio greedy può funzionare qui (cosi come è funzionava con lo zaino frazionario). È sufficiente ordinare i problemi per $v[i]/t[i]$ decrescente. A questo punto, iteriamo sulla lista ordinata, fino a quando S non è stato raggiunto. Pseudocodice 5

Algorithm 5 minimo sforzo per superare l'esame con Greedy

```
1:  procedure MINIMO-SFORZO-GREEDY(integer[] v, integer[] t, integer n, integer S)
2:      Array M ← [v_i/t_i, i] for v_i ∈ v, t_i ∈ t
3:      sort(M)
4:      Integer T ← 0
5:      for i = 1 to n do
6:          if S ≥ 0 then
7:              j ← M_i.getIndex
8:              T ← T + T_j
9:              S ← S − V_j
10:         end if
11:     end for
12:     return T
13: end procedure
```

Esercizio 6

[3/6/2014] Un testo composto da una sequenza di n parole, di lunghezze $w_1, ..., w_n$ (misurate in punti tipografici), va stampato in righe di L punti (assumendo $\max\{w_i\} < L$), senza cambiare l'ordine delle parole. La somma K delle lunghezze delle parole contenute in una riga può essere minore o uguale ad L, ma non maggiore. Una riga di lunghezza K ha penalità $L-K$. Scrivere un algoritmo greedy che stampi il testo cercando di minimizzare la sommatoria delle penalità delle righe così create, dimostrandone correttezza e complessità (l'ouput può essere prodotto stampando le parole (print w_i) e stampando il carattere di new line (println).

soluzione

È possibile risolvere il problema con un semplice algoritmo greedy: si inseriscono in ciascuna riga le parole, nell'ordine indicato, finchè non si supera la lunghezza massima consentita. Utilizziamo la variabile start per indicare l'indice della prima parola della riga corrente; Lres eÌĂ la lunghezza residua (ancora da riempire) dalla riga corrente. Si noti la stampa effettuata al termine del ciclo "for", senza la quale l'algoritmo non verrebbero stampate le parole dell'ultima riga. Il costo dell'algoritmo è $\theta(n)$. Pseudocodice 6

Algorithm 6 FORMATTA PARAGRAFO

```
1:  procedure FORMATTA-PARAGRAFO(array w_{1..n}, int S, int L)
2:      integer start ← 1;
3:      integer resto ← L − w_1;
4:      for i = 2 to n do
5:          if resto ≥ S + w_i then          ▷ aggiungiamo la parola i-esima alla riga corrente
6:              resto ← resto - S- w_i
7:              print w_i
8:          else                             ▷ iniziamo una nuova riga
9:              println w_i
10:             start ← i
11:             resto ← L − w_i
12:         end if
13:     end for
14: end procedure
```