Esercitazione 10
Algorithmi e Strutture Dati (Informatica)
A.A 2015/2016

Tong Liu

May 10, 2016

Elementi fondamentali

Un algoritmo non deterministico è un algoritmo che posto di fronte alla necessità di effettuare una decisione, sceglie sempre la trada giusta. L'algoritmo pur di interesse teorico, non sono ovviamente utilizzabili in pratica.
Tre istruzioni elementari dell'algoritmo non deterministico, ciascuna delle quali richieda $O(1)$ tempo di computazione.
choice(c): sceglie arbitrariamente un elemento dell'insieme finito C
failure: blocca la computazione in uno stato di fallimento.
success: blocca la computazione in uno stato di successo.
La struttura generale di un algoritmo non deterministico si vede lo Pseudocodice 1
Interpretazione del choice(c):
Ci sono 2 interpretazioni del choice: 1) choice è un operatore magico, che porta in tempo $O(1)$ la miglior scelta. 2) choice replica il problema di dimensione n in n copie di scelte diverse in tempo $O(1)$, esegue tutto in parallelo e sicuramente non manca mai la scelta migliore.

Algorithm 1 Algoritmo non deterministico

```
1: procedure ALGORITMO-NON-DETERMINISTICO({ opportuni paramteri })
2:     Integer[] S ← new Integers[1...n]
3:     for i = 1 to n do
4:         SET C ← choices(S, n, i, ...)                ▷ prepara le alternative(options)
5:         if C = ∅ then
6:             failure
7:         else
8:             S[i] ← choice(C)                          ▷ choice magico
9:             if S[1...i] è una soluzione then
10:                success
11:            end if
12:        end if
13:    end for
14: end procedure
```

Esercizi Programmi non deterministici

Esercizio 1

[10/02/2016] Dato un insieme A di n interi, si vuole decidere se esistono tre sottoinsiemi non vuoti S, T e U di A tali che il prodotto degli elementi di S sia uguale alla differenza tra la somma degli elementi di T e la somma degli elementi di U. Si scriva lo pseudocodice di un algoritmo non deterministico di complessità polinomiale.

Soluzione

La soluzione è descritta nello Pseudocodice 2. In pratica, l'algoritmo scorre su tutti gli elementi di A, la funzione choice() decide per ciascun elemento se farne una copia nell'sottoinsieme (S, T, U) in modo indipendemente, se sì, aggiorna la variabile di verifica corripondente al sottoinsieme (tra s, t, u). In fine, controlla se le condizioni sono soddisfate. L'operazione choice() ha un costo $O(1)$ quindi il costo della procedura è $O(n)$.

Esercizio 2

[27/01/2015] Dato un insieme A di n interi distinti, si vuole decidere se esistono tre sottoinsiemi S, T e U disgiunti e non vuoti, tali che S abbia cardinalità uguale a 3 ed il prodotto degli elementi in T sia uguale al doppio della somma degli elementi in U. Scrivere lo pseudocodice di un algoritmo non deterministico che richieda tempo polinomiale.

Soluzione

Essendo che la choice $O(1)$ la procedura è $O(n)$, la choice decide il valore u scelto da A se è opportuno inserirlo in S o in T, o in U oppure non inserirlo proprio. Una volta creati S, T e

Algorithm 2 nd tre sottoinsiemi

```
1:  procedure ND-TRE-SOTTOINSIEMI(Set A)
2:      SET S, T, U ← SET()
3:      Boolean vuotoS,vuotoT,vuotoU ← true
4:      Integer s ← 1
5:      Integer t,u ← 0
6:      for i = 1 to n do
7:          S[i] ← choice{true, false}
8:          T[i] ← choice{true, false}
9:          U[i] ← choice{true, false}
10:         if S[i] = true then
11:             s ← s · A[i]
12:             vuotoS ← false
13:         end if
14:         if T[i] = true then
15:             t ← t + A[i]
16:             vuotoT ← false
17:         end if
18:         if U[i] = true then
19:             u ← u + A[i]
20:             vuotoU ← false
21:         end if
22:     end for
23:     if not vuotoS and not vuotoT and not vuotoU and s = t − u then
24:         success
25:     else
26:         failure
27:     end if
28: end procedure
```

U, il certificato verifica se la cardinalità di S, e la uguaglianza tra il prodotto degli elementi in T e il doppio della somma degli elementi in U. Se è vero, restituisce true, false altrimenti. Pseudocodice 3

Algorithm 3 nd insiemi disgiunti

```
1:  procedure ND-INSIEMI-DISGIUNTI(Set A)
2:      SET S, T, U ← SET()
3:      for all a ∈ A do
4:          Integer scelta ← choice({1,2,3,4})
5:          if scelta = 1 then
6:              S.insert(a)
7:          else if scelta = 2 then
8:              T.insert(a)
9:          else if scelta = 3 then
10:             U.insert(a)
11:         end if
12:     end for
13:     if certificato(S, T, U) then
14:         success
15:     else
16:         failure
17:     end if
18: end procedure
19: procedure CERTIFICATO(Set S, Set T, Set U)
20:     Integer t ← 1
21:     Integer u ← 0
22:     for all e ∈ T do
23:         t ← t · e
24:     end for
25:     for all e ∈ U do
26:         u ← u + e
27:     end for
28:     if t = 2 · u and |S| = 3 and |T| > 0 and |U| > 0 then        ▷ |S| è la cardinalià di S
29:         return true
30:     end if
31:     return false
32: end procedure
```

Esercizio 3

[25/06/2014] In un grafo non orientato, un insieme coprente è un sottoinsieme S di nodi tale che, per ogni arco del grafo, almeno uno dei due nodi estremi dell'arco è incluso in S. Si scriva un algoritmo non deterministico di complessità polinomiale che, dati in input un intero k ed

un grafo G, decide se esiste un insieme coprente di cardinalità minore o uguale a k.

Soluzione

Sia S un insieme di n elementi interi che identifica i nodi, con n uguale al numero di nodi del grafo $G = (V, E)$. Un algoritmo non deterministico per calcolare l'insieme coprente è illustrato nello Pseudocodice 4. Nel primo ciclio for, vengono effettuate le scelte dei nodi. Dopodichè viene effettuato il certificato per verificare se l'insieme S così generato soddisfi i requisiti.

Algorithm 4 Insieme Coprente

```
1:  procedure ND-INSIEME-COPRENTE(Graph G(V,E), Integer k)
2:      SET S ← Set()
3:      for all i = 1 ∈ G.V do
4:          if choice({true,false}) then
5:              S.insert(i)
6:          end if
7:      end for
8:      if certificatoCoprenteK(G,S,k) then
9:          success
10:     else
11:         failure
12:     end if
13: end procedure
14: procedure CERTIFICATOCOPRENTEK(G,S,k)
15:     if S.size() > k then
16:         return false
17:     end if
18:     for all (p,q) ∈ G.E do
19:         if p ∉ S and q ∉ S then
20:             return false
21:         end if
22:     end for
23:     return true
24: end procedure
```

Esercizi esami su Programmazione Dinamica

[19/6/2015] Per comprarvi l'ultimo gadget tecnologico, avete risparmiato inserendo monete in un salvadanaio. Purtroppo non avete tenuto i conti, e non sapete quanti soldi ci sono dentro. È facile ottenere il valore totale rompendo il salvadanaio, ma sarebbe un peccato romperlo senza essere sicuri che ci siano abbastanza soldi per il vostro gadget. Fortunatamente, avete a disposizione le seguenti informazioni: il peso totale T delle monete contenute nel salvadanaio, il vettore dei pesi $p[1...n]$ e quello dei valori $v[1...n]$, dove $p[i]$ è il peso in

grammi e $v[i]$ è il valore in centesimi dell'i-esimo tipo di moneta fra gli n tipi di monete prodotti nel vostro stato. Scrivere in pseudocodice un algoritmo che restituisca il minimo valore in centesimi che può essere contenuto nel salvadanaio, e valutarne la complessità. Per esempio, supponete che il peso totale sia 50 grammi, e le monete a disposizione siano quella da 200 centesimi, che pesa 50 grammi, e quella da 50 centesimi, che pesa 10 grammi. È possibile ottenere i 50 grammi del peso totale con una singola moneta da 200 centesimi, o con 5 da 50 centesimi per un totale di 250 centesimi. Quindi il valore da restituire è 200, che è il minimo fra i due totali.

Soluzione

Per risolvere il problema, utilizziamo memoization. Sia $M(i,t)$ il minimo numero di soldi utilizzando le primi i monete e dovendo pesare t grammi. $M(i,t)$ può essere calcolato ricorsivamente come segue:

$$M(i,t) = \begin{cases} 0 & t = 0 \\ +\infty & t < 0 \\ +\infty & i = 0 \wedge t > 0 \\ min\{M(i, t-p[i]) + v[i], M(i-1,t)\} & t > 0 \wedge i > 0 \end{cases} \tag{1.1}$$

I quattro casi corrispondono alle situazioni seguenti:

- Se $t = 0$, allora 0 grammi sono sufficienti.
- Se siamo nei casi particolari generati dal quarto caso in cui uno dei due parametri è minore di zero, ritorna $+\infty$ ad indicare che questo caso non può essere considerato.
- Se non abbiamo più monete ma abbiamo ancora grammi da pesare, ritorna $+\infty$ ad indicare che questo caso non può essere considerato.
- Altrimenti, considera due possibilità: scegliere l'i-esima moneta e continuare ad utilizzarla, o non sceglierla e smettere di utilizzarla.
  Nel primo caso il peso totale diminuisce di $p[i]$ e il valore ottenuto aumenta di $v[i]$; nel secondo caso diminuisce l'indice i.

Il costo della procedura è pari a $O(nT)$, dovuto all'inizializzazione del vettore M (al fuori della chiamata ricorsiva). Pseudocodice 5

Esercizio Batteria

[25/6/2014] State viaggiando con una modernissima auto elettrica su un'autostrada lunga K km; prima che la vostra batteria si esaurisca (dopo r km), dovete fermarvi in un'area di servizio e cambiarla con una nuova batteria. Siano $D[1,...,n]$ e $C[1,...,n]$ due vettori di interi, dove $D[i]$ è la distanza dell'area di servizio i-esima dall'inizio dell'autostrada, e $C[i]$ è il costo di una nuova batteria nell'area di servizio i-esima. Progettare un algoritmo basato su programmazione dinamica che minimizzi il costo totale del viaggio, dimostrandone la correttezza e calcolandone la complessità.

Algorithm 5 Pseudocodice salvadanaio

```
1: procedure SALVADANAIO(integer[] p, integer[] v, integer t, integer i, integer[][] M)
2:     if t < 0 then
3:         return +∞
4:     end if
5:     if i = 0 and t > 0 then
6:         return +∞
7:     end if
8:     if t = 0 then
9:         return 0
10:    end if
11:    if M[i, t] = nil then
12:        M[i, t] ← min(salvadanaio(p, v, t − p[i], i, M)+v[i], salvadanaio(p, v, t, i − 1, M))
13:    end if
14:    return M[i, t]
15: end procedure
```

Soluzione

La sottostruttura ottima è facile da dimostrare, facendo notare che se viene effettuata una ricarica alla stazione i, ci si riduce al problema $D[i \dots n]$ con una lunghezza della strada pari a $K - D[i]$; qualunque soluzione ottima per questo problema fa parte della soluzione ottima del problema originale.
È possibile definire ricorsivamente il problema come segue. Aggiungiamo le stazioni fittizie $D[0]$ e $D[n+1]$, al km 0 e K, con costo $C[0] = C[n+1] = 0$.
Il problema consiste nell'individuare il costo minimo $M[i]$ da pagare nel caso si acquisti la batteria alla stazione i e l'importo minimo si deve preparare per arrivare alle stazioni future(include anche la stazione $n+1$) può essere definito ricorsivamente come segue:

$$M(i) = \begin{cases} 0 & i = n+1 \\ C[i] + \min\{M[j] \mid j : j > i \wedge D[j] \leq D[i] + r\} & \text{altrimenti} \end{cases} \tag{1.2}$$

Inviduare l'indice j per cui $D[j] \leq D[i] + r$ può richiedere $O(n)$; quindi il costo pessimo di tale algoritmo è $O(n^2)$. Il codice è il seguente:

Algorithm 6 Pseudocodice Batteria

```
1:  procedure BATTERIA(integer[] D, integer[] C, integer n)
2:      integer[] M ←new integer[0...n+1]
3:      M[n+1] ← 0
4:      for i ← n downto 0 do
5:          Integer j ← i+1
6:          Integer M[i] = +∞
7:          while j ≤ n+1 and D[j] ≤ D[i]+r do
8:              M[i] ← min (M[i], M[j])
9:              j ← j+1
10:         end while
11:         M[i] ← M[i]+C[i]
12:     end for
13:     return M[0]
14: end procedure
```