PD per il problema del cambio monete

Indice

Istanze e soluzioni ammissibili
    Codifica delle soluzioni ammissibili

Un algoritmo di PD
    Numero di passi

Errori tipici

In sintesi

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- spiegare perché il CCP chiede un importo esatto e perché un'istanza può essere inammissibile;
- scrivere la ricorrenza $v(k)$ con il valore $+\infty$ per gli importi che non si possono restituire;
- eseguire a mano l'algoritmo di PD, seguendo i confronti con il miglior valore corrente m;
- ricostruire una soluzione ottima come vettore $(x_1, \ldots, x_n)$ quando l'istanza è ammissibile;
- contare i passi dell'algoritmo di PD.

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino illimitato»: disponibilità illimitata di ogni tipo di oggetto e ricostruzione di una soluzione.
- Da questo modulo, il capitolo «Principi della programmazione dinamica»: sottostruttura ottima, sottoproblemi sovrapposti, casi base.
- Dal modulo *Algoritmi*: convenzioni dello pseudocodice e la dimensione dell'input.

Il problema di questo capitolo

Il problema di questo capitolo è il
problema del cambio monete (CCP):

la sua definizione è nella sua pagina.

Istanze e soluzioni ammissibili

Introduzione: istanze e soluzioni ammissibili

- Introduciamo l'istanza del CCP che usiamo in tutto il capitolo
- Vediamo le due differenze con il problema dello zaino 0/1: un importo esatto e una quantità illimitata di ogni taglio
- Descriviamo come si codifica una soluzione ammissibile, e perché enumerare tutte le soluzioni ammissibili diventa presto impraticabile
- È il punto di partenza dell'algoritmo di PD della sezione successiva

Esempio: istanza del CCP

Consideriamo un resto $c = 10$ da restituire e $n = 3$ tagli di moneta:

$$w = (2, 3, 5).$$

Chiamiamo questa istanza del CCP istanza di riferimento.

La differenza con il KP

- A differenza del problema dello zaino 0/1 (KP), dove il peso totale degli oggetti scelti non deve superare la capacità, il CCP richiede che il valore totale delle monete scelte sia *esattamente uguale* a c.

- Inoltre, poiché ogni taglio è disponibile in quantità illimitata, il numero di monete di un dato taglio usate in una soluzione non è limitato a priori.

Soluzioni ammissibili

Una *soluzione ammissibile* del CCP è una scelta del numero di monete di ogni taglio il cui valore totale sia esattamente uguale al resto c.

Codifica delle soluzioni ammissibili (1/2)

Una soluzione ammissibile del CCP si può descrivere con il vettore intero

$$(x_1, x_2, \ldots, x_n)$$

dove ogni componente ha il seguente significato:

$$x_j = \text{numero di monete del taglio } j \text{ usate}, \quad \forall j \in \{1, 2, \ldots, n\},$$

$$x_j \in \mathbb{Z}_{\geq 0}.$$

Codifica delle soluzioni ammissibili (2/2)

Poiché la soluzione è ammissibile, il valore totale delle monete scelte deve essere esattamente uguale a c:

$$\sum_{j=1}^{n} w_j x_j = c.$$

Il numero totale di monete di una soluzione ammissibile è

$$\sum_{j=1}^{n} x_j.$$

Quante soluzioni ammissibili?

- Poiché $\sum_{j=1}^{n} w_j x_j = c$, ogni componente x_j di una soluzione ammissibile soddisfa
$$x_j \leq \lfloor c/w_j \rfloor.$$
- Quindi il numero di soluzioni ammissibili è finito, ma cresce rapidamente con c e n.
- Enumerarle tutte per trovare quella con meno monete diventa presto impraticabile.

Riepilogo

Istanze:

- Un'istanza del CCP: un resto c e n tagli $w_1, w_2, \ldots, w_n$, ciascuno disponibile in quantità illimitata
- A differenza del KP, il valore totale delle monete scelte deve essere esattamente uguale a c

Soluzioni ammissibili:

- Un vettore intero $(x_1, x_2, \ldots, x_n)$ con $\sum_{j=1}^{n} w_j x_j = c$, che usa $\sum_{j=1}^{n} x_j$ monete
- $x_j \leq \lfloor c/w_j \rfloor$: un numero finito di soluzioni ammissibili, ma troppe da enumerare

Un algoritmo di PD

Introduzione: un algoritmo di PD

- Ogni taglio si può usare un numero qualsiasi di volte, quindi il valore ottimo dipende solo dal resto ancora da restituire
- Scriviamo una ricorrenza $v(k)$ sui resti, con il valore $+\infty$ per gli importi che non si possono restituire, e verifichiamo la sottostruttura ottima e i sottoproblemi sovrapposti
- Calcoliamo il valore ottimo top-down e bottom-up, e poi ricostruiamo una soluzione ottima quando l'istanza è ammissibile
- Infine contiamo i passi dell'algoritmo di PD

Passo 1: calcolo del valore ottimo

Per ogni $k \in \{0, 1, \ldots, c\}$, sia $v(k)$ il valore ottimo della sottoistanza del CCP data da:

1. un resto intero k da restituire,
2. lo stesso insieme $\{1, 2, \ldots, n\}$ di tagli, ciascuno disponibile in quantità illimitata.

La ricorrenza per $v(k)$

La formula ricorsiva per calcolare $v(k)$ è la seguente:

$$v(k) = \begin{cases} 0 & \text{se } k = 0, \\ +\infty & \text{se } k \geq 1 \text{ e } w_j > k \\ & \text{per ogni } j \in \{1, 2, \ldots, n\}, \\ \min\limits_{j\,:\,w_j \leq k} \{1 + v(k - w_j)\} & \text{se } k \geq 1 \text{ e } w_j \leq k \\ & \text{per qualche } j \in \{1, 2, \ldots, n\}. \end{cases}$$

Sottoproblemi indipendenti e caso base

- Il processo di decomposizione ricorsiva della ricorrenza per $v(k)$ genera al più n **sottoproblemi indipendenti**, corrispondenti alle possibili scelte di un taglio j, con $w_j \leq k$, usato come una delle monete che restituiscono il resto k.

- Il caso base è $v(0) = 0$, perché per restituire un resto nullo non serve alcuna moneta.

- Inoltre $v(k) = +\infty$ se nessun taglio è abbastanza piccolo da essere usato, perché in questo caso il resto k non si può restituire affatto.

Sottostruttura ottima

- Il valore ottimo $v(k)$ si ottiene provando ogni taglio j con $w_j \leq k$ e scegliendo quello che minimizza la somma $1 + v(k - w_j)$ della singola moneta del taglio j e del valore ottimo del sottoproblema residuo, che restituisce il resto rimanente $k - w_j$.
- Quindi una soluzione ottima per la sottoistanza con resto k si costruisce combinando la moneta scelta, del taglio j, con una soluzione ottima del sottoproblema residuo corrispondente, che restituisce il resto $k - w_j$.

$\rightsquigarrow$ **Sottostruttura ottima verificata!**

Sottoproblemi sovrapposti

- Scelte diverse del taglio usato come prima moneta, eventualmente fatte per valori diversi di k, possono portare allo stesso resto residuo.
- Quindi lo stesso valore $v(k)$ può dover essere calcolato più volte durante la valutazione ricorsiva della ricorrenza per $v(k)$.

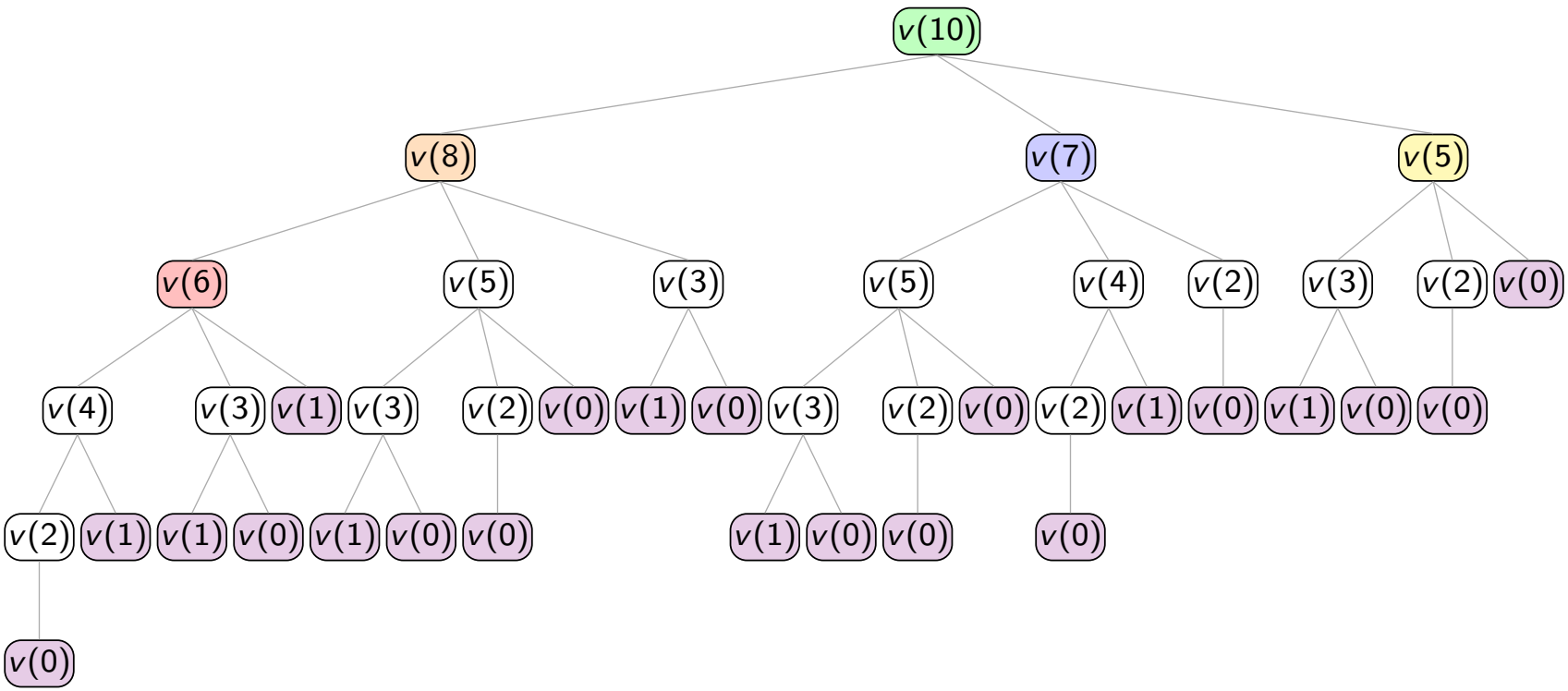
$\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

L'approccio top-down

- Un **primo metodo** per calcolare il valore ottimo $v(c)$ dell'istanza del CCP è l'**approccio top-down**.
- Il metodo parte dalla chiamata con $k = c$ e valuta ricorsivamente la ricorrenza per $v(k)$: ogni chiamata prova ogni taglio j con $w_j \leq k$, finché il resto vale 0 o nessun taglio è abbastanza piccolo da essere usato.
- L'**albero di ricorsione** della funzione v ha un nodo per ogni chiamata ricorsiva, etichettato con la chiamata che rappresenta. La radice rappresenta la chiamata iniziale. I figli di un nodo rappresentano le chiamate ricorsive fatte da quella chiamata. Le foglie rappresentano i casi base.

Esempio: albero di ricorsione (1/4)

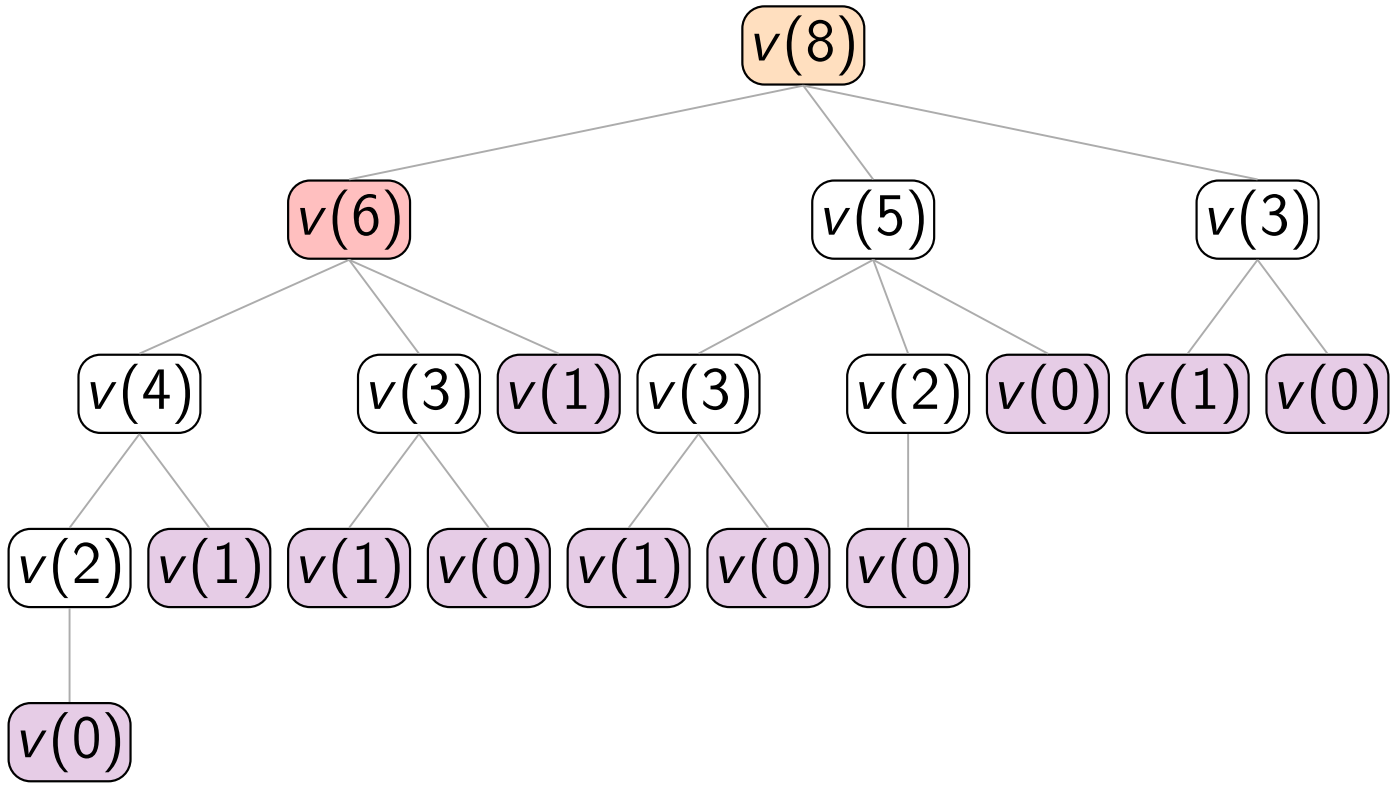
Consideriamo l'istanza di riferimento, con $c = 10$ e $w = (2, 3, 5)$. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(10)$:



Le due slide successive ingrandiscono i sottoalberi dei tre figli della radice.

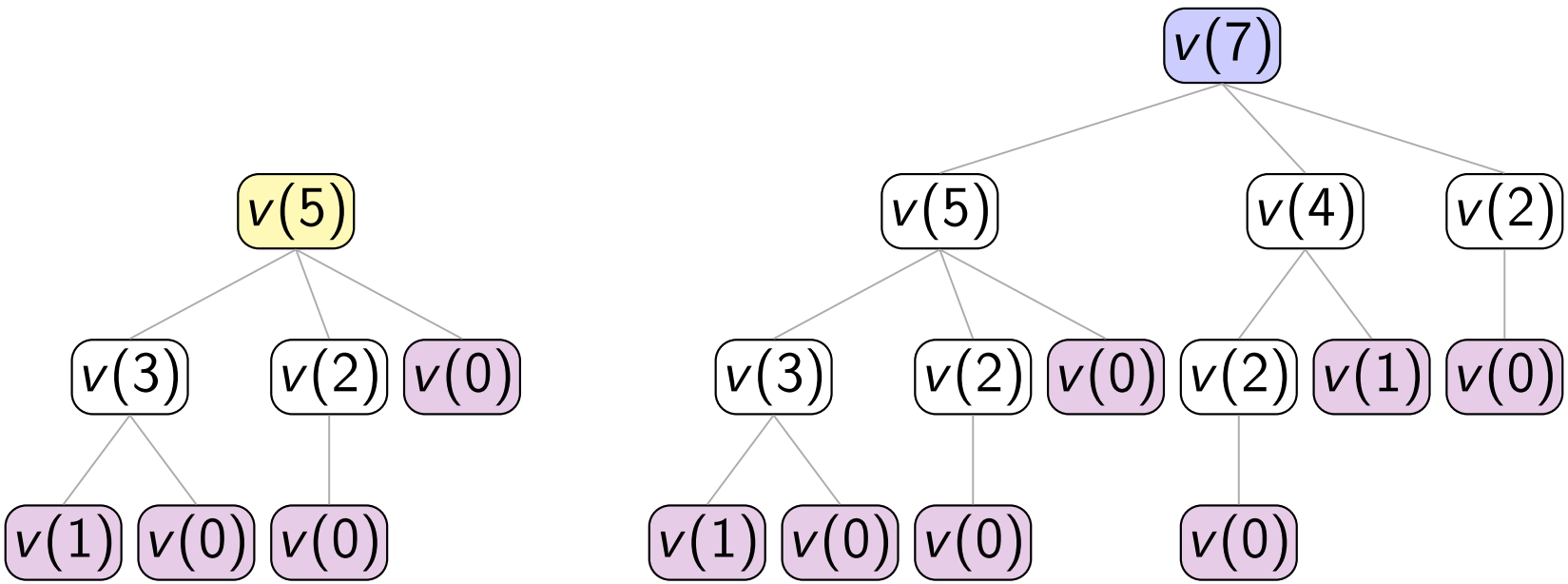
Esempio: albero di ricorsione (2/4)

Il sottoalbero della chiamata $v(8)$, figlio della radice $v(10)$:



Esempio: albero di ricorsione (3/4)

I sottoalberi delle chiamate $v(5)$ e $v(7)$, gli altri due figli della radice $v(10)$:



Esempio: albero di ricorsione (4/4)

La funzione v viene chiamata 42 volte, e gli stessi sottoproblemi vengono risolti più volte:

- $v(0)$ viene chiamata 14 volte,
- $v(1)$ viene chiamata 8 volte,
- $v(2)$ viene chiamata 6 volte,
- $v(3)$ viene chiamata 5 volte,
- $v(4)$ viene chiamata 2 volte,
- $v(5)$ viene chiamata 3 volte.

Tutte le altre chiamate vengono fatte una sola volta.

Calcolo dei valori in ordine crescente di resto

Il valore $v(k)$ si può calcolare una volta noti i valori $v(k')$ per ogni resto minore $k' < k$.

Quindi tutti i valori $v(k)$ si possono calcolare considerando il resto k in ordine crescente.

Esempio: calcolo dei valori della funzione v (1/4)

Consideriamo l'istanza di riferimento, con resto $c = 10$ da restituire e tagli $w = (2, 3, 5)$.

I valori della funzione v si calcolano come segue:

- Il caso base dà $v(0) = 0$.
- Calcolato $v(0)$, possiamo calcolare $v(1)$: poiché nessun taglio soddisfa $w_j \leq 1$, otteniamo

$$v(1) = +\infty.$$

- Calcolato $v(0)$, possiamo calcolare $v(2)$:

$$v(2) = 1 + v(0) = 1 + 0 = 1.$$

Esempio: calcolo dei valori della funzione v (2/4)

- Calcolati $v(0)$ e $v(1)$, possiamo calcolare $v(3)$:

$$v(3) = \min\{1 + v(1),\ 1 + v(0)\} = \min\{1 + \infty,\ 1 + 0\} = 1.$$

- Calcolati $v(1)$ e $v(2)$, possiamo calcolare $v(4)$, provando ogni taglio j con $w_j \leq 4$:

$$v(4) = \min\{1 + v(2),\ 1 + v(1)\} = \min\{1 + 1,\ 1 + \infty\} = 2.$$

- Calcolati $v(0)$, $v(2)$ e $v(3)$, possiamo calcolare $v(5)$:

$$\begin{aligned} v(5) &= \min\{1 + v(3),\ 1 + v(2),\ 1 + v(0)\} \\ &= \min\{1 + 1,\ 1 + 1,\ 1 + 0\} = 1. \end{aligned}$$

Esempio: calcolo dei valori della funzione v (3/4)

- Calcolati $v(1)$, $v(3)$ e $v(4)$, possiamo calcolare $v(6)$:

$$\begin{aligned} v(6) &= \min\{1 + v(4),\ 1 + v(3),\ 1 + v(1)\} \\ &= \min\{1 + 2,\ 1 + 1,\ 1 + \infty\} = 2. \end{aligned}$$

- Calcolati $v(2)$, $v(4)$ e $v(5)$, possiamo calcolare $v(7)$, provando ogni taglio j con $w_j \leq 7$:

$$\begin{aligned} v(7) &= \min\{1 + v(5),\ 1 + v(4),\ 1 + v(2)\} \\ &= \min\{1 + 1,\ 1 + 2,\ 1 + 1\} = 2. \end{aligned}$$

  Qui i tagli $w_1 = 2$ e $w_3 = 5$ sono in pareggio: entrambi danno valore 2. Poiché l'algoritmo bottom-up aggiorna il valore ottimo solo quando trova un valore strettamente minore, viene registrato il primo taglio che raggiunge il minimo, $w_1 = 2$.

Esempio: calcolo dei valori della funzione v (4/4)

- Calcolati $v(3)$, $v(5)$ e $v(6)$, possiamo calcolare $v(8)$:

$$\begin{aligned} v(8) &= \min\{1 + v(6), 1 + v(5), 1 + v(3)\} \\ &= \min\{1 + 2, 1 + 1, 1 + 1\} = 2. \end{aligned}$$

- Infine, calcolati $v(5)$, $v(7)$ e $v(8)$, possiamo calcolare $v(10)$, provando ogni taglio j con $w_j \leq 10$:

$$\begin{aligned} v(10) &= \min\{1 + v(8), 1 + v(7), 1 + v(5)\} \\ &= \min\{1 + 2, 1 + 2, 1 + 1\} = 2. \end{aligned}$$

La tabella di lookup

- Possiamo evitare di risolvere ripetutamente gli stessi sottoproblemi memorizzando il valore di ogni sottoproblema appena calcolato. I valori si memorizzano in una **tabella di lookup**.

- Poiché la soluzione di ogni sottoproblema dipende solo dai sottoproblemi con un resto minore, possiamo risolvere i sottoproblemi in ordine crescente di k, partendo dal caso base $v(0) = 0$ fino ad arrivare a $v(c)$.

- Questa osservazione porta in modo naturale a un **metodo molto più efficiente** per calcolare il valore ottimo $v(c)$ dell'istanza del CCP.

L'approccio bottom-up

- Un **secondo metodo** per calcolare il valore ottimo $v(c)$ dell'istanza del CCP è l'**approccio bottom-up**.
- Il metodo parte dal caso base e valuta la ricorrenza per $v(k)$ per valori crescenti di $k \in \{1, 2, \ldots, c\}$: ogni valore $v(k)$ si calcola con valori già calcolati, fino a ottenere il valore $v(c)$ dell'istanza del CCP.
- Vediamo ora lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**.

L'algoritmo di PD con l'approccio bottom-up (1/2)

Algorithm: CCP_DP (1/2)

Input : Un'istanza del CCP data da n, dall'array $w[1..n]$ dei tagli e dal resto c

Output: Il valore ottimo

```
1 function CCP_DP(n, w, c)
2 |   v := new array[0..c]
3 |   v[0] ← 0
```

L'algoritmo di PD con l'approccio bottom-up (2/2)

Algorithm: CCP_DP (2/2)

```
function CCP_DP(n, w, c) (continua)
4     for k := 1 to c do
5         m := +∞
6         for j := 1 to n do
7             if w[j] ≤ k e 1 + v[k − w[j]] < m then
8                 m ← 1 + v[k − w[j]]
9             end
10        end
11        v[k] ← m
12    end
13    return v[c]
14 end
```

Come funziona l'algoritmo di PD

- L'array $v[0..c]$ è la **tabella di lookup**: memorizza i valori ottimi di tutti i sottoproblemi.
- L'istruzione $v[0] \leftarrow 0$ inizializza il caso base, corrispondente a un resto nullo.
- Il ciclo esterno, su $k \in \{1, 2, \ldots, c\}$, calcola le componenti dell'array in ordine crescente di resto.
- Il ciclo interno, su $j \in \{1, 2, \ldots, n\}$, prova ogni taglio e tiene in m il valore minimo trovato finora; m è inizializzato a $+\infty$, così $v[k]$ resta $+\infty$ se non si trova alcun taglio j con $w[j] \leq k$.
- Alla fine del ciclo interno, $v[k] \leftarrow m$ memorizza il valore ottimo $v[k]$.
- Infine, l'algoritmo restituisce il valore ottimo dell'istanza del CCP, $v[c]$.

Esempio: esecuzione dell'algoritmo di PD (1/7)

Consideriamo l'istanza di riferimento, con resto $c = 10$ da restituire e tagli $w = (2, 3, 5)$.

- Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di k e j, la cella $v[k]$, il valore corrente di m e il confronto eseguito.
- La cella di m è evidenziata in rosso quando viene aggiornata; la cella di $v[k]$ è evidenziata in blu quando viene assegnata.

Esempio: esecuzione dell'algoritmo di PD (2/7)

k	j	$v[k]$	m	Azione
–	–	0	–	$v[0] \leftarrow 0$
	–	–	$+\infty$	$m := +\infty$
1	1	–	$+\infty$	$w[1] = 2 > 1$: si salta
	2	–	$+\infty$	$w[2] = 3 > 1$: si salta
	3	–	$+\infty$	$w[3] = 5 > 1$: si salta
	–	$+\infty$	$+\infty$	$v[1] \leftarrow m = +\infty$
	–	–	$+\infty$	$m := +\infty$
2	1	–	1	$1+v[0] = 1 < m$: $m \leftarrow 1$
	2	–	1	$w[2] = 3 > 2$: si salta
	3	–	1	$w[3] = 5 > 2$: si salta
	–	1	1	$v[2] \leftarrow m = 1$

Esempio: esecuzione dell'algoritmo di PD (3/7)

k	j	$v[k]$	m	Azione
	–	–	$+\infty$	$m := +\infty$
3	1	–	$+\infty$	$1+v[1] = +\infty \not< m = +\infty$: nessun aggiornamento
	2	–	1	$1+v[0] = 1 < m$: $m \leftarrow 1$
	3	–	1	$w[3] = 5 > 3$: si salta
	–	1	1	$v[3] \leftarrow m = 1$
	–	–	$+\infty$	$m := +\infty$
4	1	–	2	$1+v[2] = 2 < m$: $m \leftarrow 2$
	2	–	2	$1+v[1] = +\infty \not< m = 2$: nessun aggiornamento
	3	–	2	$w[3] = 5 > 4$: si salta
	–	2	2	$v[4] \leftarrow m = 2$

Esempio: esecuzione dell'algoritmo di PD (4/7)

k	j	$v[k]$	m	Azione
	–	–	$+\infty$	$m := +\infty$
5	1	–	2	$1+v[3] = 2 < m$: $m \leftarrow 2$
	2	–	2	$1+v[2] = 2 \not< m = 2$: nessun aggiornamento
	3	–	1	$1+v[0] = 1 < m$: $m \leftarrow 1$
	–	1	1	$v[5] \leftarrow m = 1$
	–	–	$+\infty$	$m := +\infty$
6	1	–	3	$1+v[4] = 3 < m$: $m \leftarrow 3$
	2	–	2	$1+v[3] = 2 < m$: $m \leftarrow 2$
	3	–	2	$1+v[1] = +\infty \not< m = 2$: nessun aggiornamento
	–	2	2	$v[6] \leftarrow m = 2$

Esempio: esecuzione dell'algoritmo di PD (5/7)

k	j	$v[k]$	m	Azione
	–	–	$+\infty$	$m := +\infty$
7	1	–	2	$1+v[5] = 2 < m$: $m \leftarrow 2$
	2	–	2	$1+v[4] = 3 \not< m = 2$: nessun aggiornamento
	3	–	2	$1+v[2] = 2 \not< m = 2$: nessun aggiornamento
	–	2	2	$v[7] \leftarrow m = 2$
	–	–	$+\infty$	$m := +\infty$
8	1	–	3	$1+v[6] = 3 < m$: $m \leftarrow 3$
	2	–	2	$1+v[5] = 2 < m$: $m \leftarrow 2$
	3	–	2	$1+v[3] = 2 \not< m = 2$: nessun aggiornamento
	–	2	2	$v[8] \leftarrow m = 2$

Esempio: esecuzione dell'algoritmo di PD (6/7)

k	j	$v[k]$	m	Azione
	–	–	$+\infty$	$m := +\infty$
9	1	–	3	$1+v[7] = 3 < m$: $m \leftarrow 3$
	2	–	3	$1+v[6] = 3 \not< m = 3$: nessun aggiornamento
	3	–	3	$1+v[4] = 3 \not< m = 3$: nessun aggiornamento
	–	3	3	$v[9] \leftarrow m = 3$
	–	–	$+\infty$	$m := +\infty$
10	1	–	3	$1+v[8] = 3 < m$: $m \leftarrow 3$
	2	–	3	$1+v[7] = 3 \not< m = 3$: nessun aggiornamento
	3	–	2	$1+v[5] = 2 < m$: $m \leftarrow 2$
	–	2	2	$v[10] \leftarrow m = 2$

Esempio: esecuzione dell'algoritmo di PD (7/7)

Questo è l'array v dopo l'esecuzione dell'algoritmo di PD:

k	0	1	2	3	4	5	6	7	8	9	10
$v[k]$	0	$+\infty$	1	1	2	1	2	2	2	3	2

- Dopo l'esecuzione dell'algoritmo di PD, l'array v contiene il valore ottimo $v[k]$ per ogni resto $k \in \{0, 1, \ldots, c\}$.
- $v[c]$ è il numero minimo di monete necessarie per restituire il resto c: qui $v[10] = 2$.

Passo 2: calcolo di una soluzione ottima

- Per calcolare una soluzione ottima, cioè il vettore intero $(x_1, x_2, \ldots, x_n)$, è necessario memorizzare alcune informazioni aggiuntive.
- Per ogni resto k, oltre al valore ottimo $v[k]$, memorizziamo $s[k]$, l'indice del taglio della moneta usata per ottenere il valore ottimo $v[k]$ nella ricorrenza per $v(k)$.
- Più precisamente, $s[k]$ è uguale all'indice j del taglio w_j che minimizza il membro destro della ricorrenza nel calcolo di $v[k]$.
- Vediamo ora lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, con la memorizzazione delle informazioni necessarie per costruire una soluzione ottima.

Le informazioni per una soluzione ottima (1/2)

Algorithm: CCP_DP_InfoSol (1/2)

Input : Un'istanza del CCP data da n, dall'array $w[1..n]$ dei tagli e dal resto c

Output: Il valore ottimo e l'array $s[0..c]$ degli indici dei tagli usati

```
1 function CCP_DP_InfoSol(n, w, c)
2 |   v := new array[0..c]; s := new array[0..c]
3 |   v[0] ← 0; s[0] ← 0
```

Le informazioni per una soluzione ottima (2/2)

Algorithm: CCP_DP_InfoSol (2/2)

```
function CCP_DP_InfoSol(n, w, c) (continua)
4     for k := 1 to c do
5         m := +∞; t := 0
6         for j := 1 to n do
7             if w[j] ≤ k e 1 + v[k − w[j]] < m then
8                 m ← 1 + v[k − w[j]]; t ← j
9             end
10        end
11        v[k] ← m; s[k] ← t
12    end
13    return v[c], s
14 end
```

Come si memorizzano le informazioni

- L'algoritmo di PD usa la variabile temporanea aggiuntiva t per tenere traccia della decisione presa quando m viene aggiornato.
- Se si usa il taglio $w[j]$, $t \leftarrow j$ ne memorizza l'indice.
- Alla fine del ciclo interno, $s[k] \leftarrow t$ memorizza l'indice del taglio della moneta usata, insieme al valore ottimo $v[k] \leftarrow m$.

Esempio: calcolo delle informazioni aggiuntive

Consideriamo l'istanza di riferimento, con resto $c = 10$ da restituire e tagli $w = (2, 3, 5)$.

Questo è l'array s dopo l'esecuzione dell'algoritmo di PD:

k	0	1	2	3	4	5	6	7	8	9	10
$s[k]$	0	0	1	2	1	3	2	1	2	1	3

- Per esempio, $s[5] = 3$ perché la ricorrenza sceglie il taglio $w_3 = 5$ nel calcolo di $v[5] = 1$, e $s[10] = 3$ perché il taglio $w_3 = 5$ viene scelto anche nel calcolo di $v[10] = 2$.
- La componente $s[1]$ resta indefinita (0, per convenzione), perché $v[1] = +\infty$ è inammissibile.

Calcolo di una soluzione ottima dall'array s

- Calcolato l'array s, e purché $v[c] < +\infty$, si può ricostruire una soluzione ottima usando ripetutamente una moneta del taglio $s[k]$ e passando al resto residuo $k - w_{s[k]}$, partendo da $k = c$ e da $x_j = 0$ per ogni $j \in \{1, 2, \ldots, n\}$, finché $k = 0$:

$$x_{s[k]} \leftarrow x_{s[k]} + 1, \qquad k \leftarrow k - w_{s[k]}.$$

- Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, dove ogni componente x_j è il numero di monete del taglio j in una soluzione ottima.

La condizione $v[c] < +\infty$

- La condizione $v[c] < +\infty$ è necessaria: se $v[c] = +\infty$ l'istanza è inammissibile, $s[c] = 0$ e non c'è alcuna soluzione da ricostruire.
- Se $v[c] < +\infty$, allora $s[k] \geq 1$ per ogni resto k visitato dalla procedura.
- Il seguente pseudocodice ricostruisce una soluzione ottima dall'array s.

Algorithm: CCP_Reconstruct

Input : Il numero n di tagli, l'array $w[1..n]$, il resto c con $v[c] < +\infty$ e l'array $s[0..c]$
Output : Una soluzione ottima

```
1  function CCP_Reconstruct(n, w, c, s)
2      x := new array[1..n]
3      for j := 1 to n do
4          x[j] ← 0
5      end
6      k := c
7      while k > 0 do
8          x[s[k]] ← x[s[k]] + 1
9          k ← k − w[s[k]]
10     end
11     return x
12 end
```

Esempio: calcolo di una soluzione ottima (1/2)

Dopo l'esecuzione dell'algoritmo di PD sull'istanza di riferimento, abbiamo il seguente array s:

k	0	1	2	3	4	5	6	7	8	9	10
$s[k]$	0	0	1	2	1	3	2	1	2	1	3

Possiamo ricostruire una soluzione ottima partendo da $k = 10$ come segue:

Iterazione	k	$s[k]$	$x[1]$	$x[2]$	$x[3]$	$k \leftarrow k - w_{s[k]}$
–	–	–	0	0	0	$k = 10$
1	10	3	0	0	1	$k = 10 - 5 = 5$
2	5	3	0	0	2	$k = 5 - 5 = 0$

Esempio: calcolo di una soluzione ottima (2/2)

Quindi il vettore

$$(0, 0, 2)$$

è una soluzione ottima.

- Usa due monete del taglio $w_3 = 5$.
- Il valore totale corrispondente è $2 \cdot 5 = 10 = c$.
- Il valore ottimo corrispondente è $v(10) = 2$.

Numero di passi

- Sia $S(n, c)$ il numero di passi eseguiti da `CCP_DP` nel caso peggiore, dove n è il numero di tagli e c è il resto.
- Calcoliamo $S(n, c)$ contando quante volte vengono eseguite le righe dello pseudocodice di `CCP_DP`.
- Nel caso peggiore, la condizione alla riga 7 è vera a ogni iterazione del ciclo interno, e quindi la riga 8 viene eseguita tante volte quanto la riga 7.

Richiamiamo lo pseudocodice di CCP_DP (1/2)

Algorithm: CCP_DP (1/2)

Input : Un'istanza del CCP data da n, dall'array $w[1..n]$ dei tagli e dal resto c

Output: Il valore ottimo

```
1 function CCP_DP(n, w, c)
2 |   v := new array[0..c]
3 |   v[0] ← 0
```

Richiamiamo lo pseudocodice di CCP_DP (2/2)

Algorithm: CCP_DP (2/2)

function *CCP_DP*(n, w, c) (continua)

```
4   |   for k := 1 to c do
5   |   |   m := +∞
6   |   |   for j := 1 to n do
7   |   |   |   if w[j] ≤ k e 1 + v[k − w[j]] < m then
8   |   |   |   |   m ← 1 + v[k − w[j]]
9   |   |   |   end
10  |   |   end
11  |   |   v[k] ← m
12  |   end
13  |   return v[c]
14  end
```

Numero di esecuzioni delle righe

Il numero di esecuzioni delle righe rilevanti è:

- Riga 2 viene eseguita 1 volta.
- Riga 3 viene eseguita 1 volta.
- Riga 4 viene eseguita c volte.
- Riga 5 viene eseguita c volte.
- Riga 6 viene eseguita cn volte.
- Riga 7 viene eseguita cn volte.
- Riga 8 viene eseguita cn volte.
- Riga 11 viene eseguita c volte.
- Riga 13 viene eseguita 1 volta.

Il numero totale di passi

Quindi il numero totale di passi nel caso peggiore è

$$1 + 1 + c + c + cn + cn + cn + c + 1.$$

Quindi

$$S(n, c) = 3 + 3c + 3cn.$$

Quindi

$$S(n, c) = \Theta(nc).$$

Infatti,

$$\lim_{n,c\to\infty} \frac{3cn + 3c + 3}{nc} = \lim_{n,c\to\infty} \left(3 + \frac{3}{n} + \frac{3}{nc}\right) = 3 \in \mathbb{R}_{>0}.$$

Il numero di passi dell'algoritmo di PD

L'algoritmo di PD esegue

$$S(n, c) = \Theta(nc)$$

passi nel caso peggiore.

- La versione `CCP_DP_InfoSol` memorizza l'array aggiuntivo s ed esegue solo un numero costante di operazioni in più dentro gli stessi cicli.
- Quindi il suo numero di passi resta $\Theta(nc)$.

Riepilogo (1/2)

La ricorrenza sui resti:

- $v(k)$ è il numero minimo di monete che restituiscono il resto k, con tutti gli n tagli
- $v(0) = 0$; $v(k) = +\infty$ se nessun taglio è abbastanza piccolo; altrimenti $v(k) = \min_{j:\, w_j \leq k}\{1 + v(k - w_j)\}$
- Al più n sottoproblemi indipendenti; sottostruttura ottima e sottoproblemi sovrapposti verificati

Top-down e bottom-up:

- L'albero di ricorsione per $v(10)$ ha 42 chiamate, con sottoproblemi ripetuti
- `CCP_DP` riempie la tabella di lookup $v[0..c]$ per k crescente; sull'istanza di riferimento $v[10] = 2$

Riepilogo (2/2)

Una soluzione ottima:

- $s[k]$ è l'indice del taglio della moneta scelta per il resto k
- Solo se $v[c] < +\infty$: da $k = c$, una moneta in più del taglio $s[k]$, poi $k \leftarrow k - w_{s[k]}$, finché $k = 0$
- Sull'istanza di riferimento: $(0, 0, 2)$, due monete del taglio $w_3 = 5$

Numero di passi:

- $S(n, c) = 3 + 3c + 3cn = \Theta(nc)$, anche con l'array s

Errori tipici

Introduzione: errori tipici

- Elenchiamo gli errori più frequenti con l'algoritmo di PD per il CCP
- Riguardano il valore $+\infty$, la ricostruzione di una soluzione, l'array s e la differenza con l'UKP
- Conoscerli aiuta a eseguire correttamente a mano l'algoritmo di PD

Osservazione 1: Errori tipici (1/2)

- ***Usare*** 0 ***invece di*** $+\infty$***.*** *Un importo che non si può restituire ha* $v(k) = +\infty$*; con il valore* 0 *l'algoritmo di PD affermerebbe che non serve alcuna moneta.*
- ***Ricostruire un'istanza inammissibile.*** *Se* $v[c] = +\infty$ *allora* $s[c] = 0$*, e la ricostruzione leggerebbe* $x[0]$ *e* $w[0]$*, fuori dagli array: bisogna prima controllare che* $v[c] < +\infty$*.*

Osservazione: Errori tipici (2/2)

- ***Memorizzare il valore della moneta invece del suo indice.*** *Il vettore x è indicizzato dai tagli: $s[k]$ memorizza l'indice j, e il valore della moneta è $w_{s[k]}$.*
- ***Confondere il CCP con l'UKP.*** *Il CCP chiede esattamente c e minimizza il numero di monete; l'UKP chiede al più c e massimizza il profitto.*

Riepilogo

Errori tipici:

- Un importo che non si può restituire ha valore $+\infty$, non 0
- Prima di ricostruire una soluzione, si controlla che $v[c] < +\infty$
- $s[k]$ memorizza l'indice j del taglio, non il valore della moneta $w_{s[k]}$
- Il CCP chiede esattamente c e minimizza; l'UKP chiede al più c e massimizza

In sintesi

Introduzione: in sintesi

- Raccogliamo le idee chiave del capitolo
- Chiudiamo con il quadro riassuntivo dell'istanza di riferimento

Riepilogo (1/2)

- Il CCP chiede il numero minimo di monete il cui valore totale sia esattamente c; l'istanza è inammissibile quando $v(c) = +\infty$.
- La ricorrenza $v(k) = \min_{j:\, w_j \leq k}\{1 + v(k - w_j)\}$ si calcola bottom-up in $\Theta(nc)$ passi, un numero pseudo-polinomiale.
- Una soluzione ottima si legge a ritroso a partire da $s[c]$, l'indice del taglio della prima moneta scelta per l'importo c, e solo se $v[c] < +\infty$.

Riepilogo (2/2)

Quadro riassuntivo dell'esempio del CCP. L'istanza di riferimento ($c = 10$, $w = (2, 3, 5)$):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	2	2	numero minimo di monete, soluzione $(0, 0, 2)$