PD per il problema dello zaino 0/1 (1/2)

Indice

Istanze e soluzioni ammissibili
- Codifica delle soluzioni ammissibili

Un primo algoritmo di PD
- Numero di passi
- Un'implementazione con liste
- Numero di passi con le liste

Obiettivi

Al termine di questo capitolo saremo in grado di:

- scrivere la ricorrenza $v(j, k)$ del KP, spiegare perché è corretta e calcolare a mano la tabella con l'approccio top-down, con l'approccio bottom-up e con un solo vettore;
- ricostruire una soluzione ottima dalla matrice s delle decisioni;
- contare i passi dell'algoritmo di PD e spiegare perché $\Theta(nc)$ è pseudopolinomiale e non polinomiale;
- descrivere gli stati (w, p), scartare quelli dominati ed eseguire l'algoritmo di PD su liste di stati;
- scambiare i ruoli di profitti e pesi e calcolare il valore ottimo dalla tabella $u(j, q)$.

Prerequisiti

- Da questo modulo, il capitolo «Principi della programmazione dinamica»: sottostruttura ottima, sottoproblemi sovrapposti, relazioni di ricorrenza, casi base e approcci top-down e bottom-up.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema dello zaino 0/1 (KP), la codifica binaria di una soluzione e i 2^n sottoinsiemi di oggetti.
- Dal modulo *Algoritmi*: come si legge lo pseudocodice e la dimensione dell'input di un algoritmo (serve a spiegare perché $\Theta(nc)$ è pseudopolinomiale).

Dove eravamo

- **Come problema**: il KP, le sue istanze e i 2^n sottoinsiemi (*Problemi di ottimizzazione combinatoria*, capitolo «Problemi di zaino»).
- **Come modello**: il modello PLI, il suo rilassamento lineare e i bound duali (*Problemi di ottimizzazione combinatoria*, capitolo «Formulazioni dei problemi di zaino»).
- **Ora, come tabella**: l'algoritmo di PD calcola esattamente il valore ottimo, senza modello e senza solver.

Istanze e soluzioni ammissibili

Introduzione: istanze e soluzioni ammissibili

- Il problema di questo capitolo è il problema dello zaino 0/1 (KP): la sua definizione e le sue ipotesi sono nella sua pagina
- Fissiamo una piccola istanza del KP, usata negli esempi del primo algoritmo di PD
- Ricordiamo come una soluzione ammissibile si codifica con un vettore binario
- Contiamo i sottoinsiemi di oggetti: enumerarli tutti diventa presto impraticabile, ed è per questo che serve un metodo migliore

Esempio: istanza del KP

Consideriamo uno zaino di capacità $c = 5$ e un insieme $\{1, 2, 3\}$ di 3 oggetti ($n = 3$), con i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

Chiamiamo questa istanza del KP la prima istanza del capitolo.

Il numero di sottoinsiemi

- Il numero di possibili sottoinsiemi di oggetti è 2^n poiché, per ogni oggetto, possiamo scegliere in modo indipendente se inserirlo o no nello zaino.
- Alcuni sottoinsiemi possono non rispettare la capacità dello zaino. Di conseguenza, il numero di soluzioni ammissibili è al più 2^n.
- Il numero di possibili soluzioni del KP cresce esponenzialmente con il numero n di oggetti.
- Pertanto, l'enumerazione di tutte le possibili soluzioni diventa presto impraticabile al crescere di n.

Codifica delle soluzioni ammissibili (1/2)

Una *soluzione ammissibile* del KP è un sottoinsieme di oggetti il cui peso totale non supera la capacità dello zaino.

Una soluzione ammissibile del KP si può descrivere con il vettore binario

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

dove ogni componente ha il seguente significato:

$$x_j = \begin{cases} 1 & \text{se l'oggetto } j \text{ è selezionato,} \\ 0 & \text{altrimenti,} \end{cases} \qquad \forall j \in \{1, 2, \ldots, n\}.$$

Codifica delle soluzioni ammissibili (2/2)

Poiché la soluzione è ammissibile, il peso totale degli oggetti selezionati non deve superare la capacità dello zaino:

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

Il profitto totale di una soluzione ammissibile è

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

Elenchiamo ora tutti i sottoinsiemi di oggetti della prima istanza,

prima con un albero binario, poi con un disegno e una tabella.

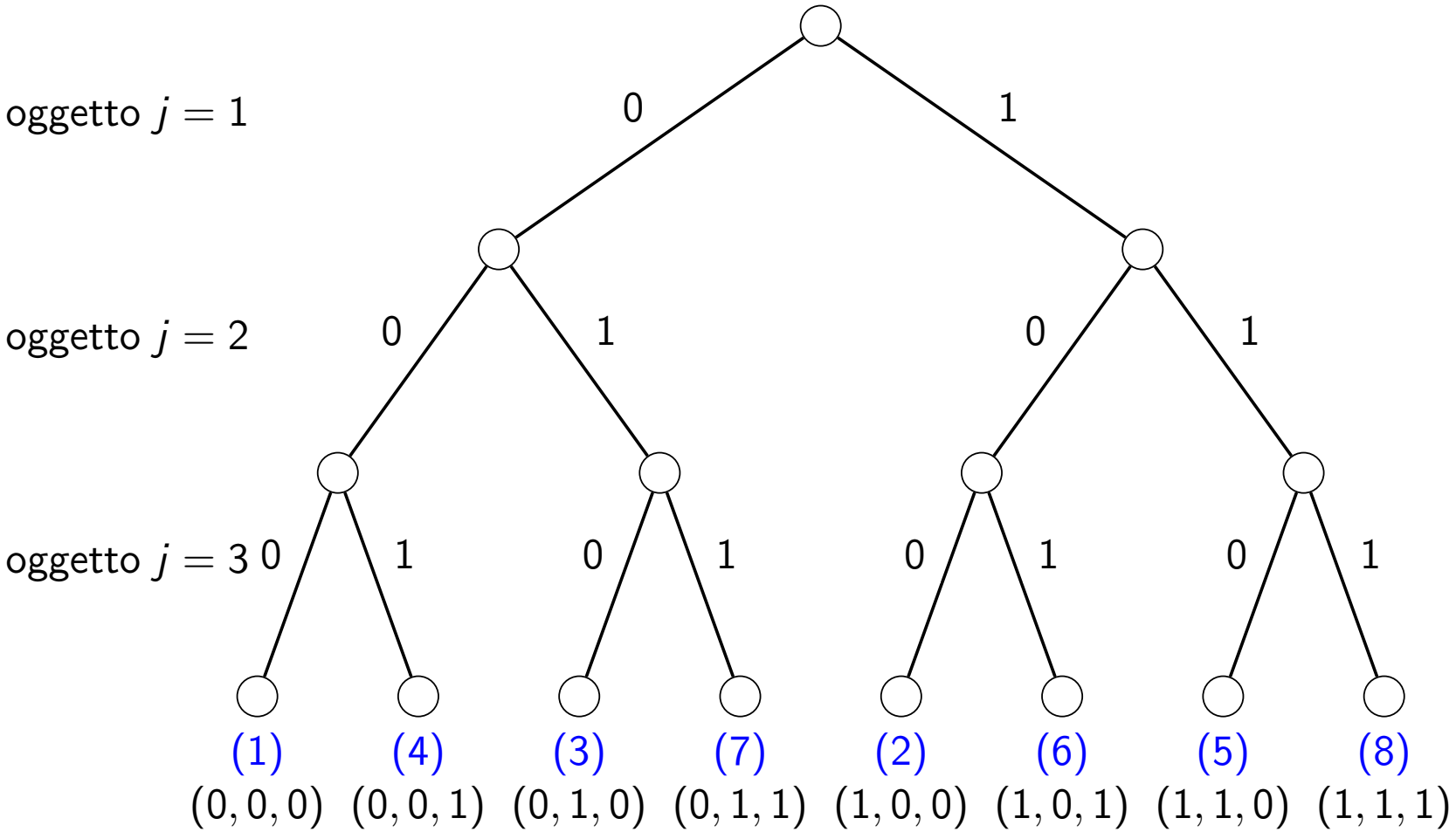
Esempio: tutte le possibili soluzioni del KP (1/6)

Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

- Ci sono $2^3 = 8$ possibili sottoinsiemi di oggetti, corrispondenti ai diversi modi di selezionare o non selezionare gli oggetti.
- Nell'albero binario della slide successiva ogni livello corrisponde a un oggetto: un arco con etichetta 1 indica che l'oggetto è selezionato, mentre un arco con etichetta 0 indica che non è selezionato.
- Ogni foglia rappresenta un sottoinsieme completo di oggetti e riporta il vettore corrispondente.

Esempio: tutte le possibili soluzioni del KP (2/6)

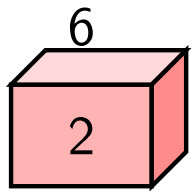


Esempio: tutte le possibili soluzioni del KP (3/6)

- Le due slide successive mostrano i sottoinsiemi associati alle foglie dell'albero, tramite le etichette $(1), (2), \ldots, (8)$.
- Per ogni oggetto selezionato, il numero scritto dentro il parallelepipedo è il suo peso, mentre il numero sopra è il suo profitto.
- La larghezza di ogni parallelepipedo è proporzionale al peso dell'oggetto.
- I sottoinsiemi il cui peso totale supera la capacità dello zaino sono indicati come non ammissibili.

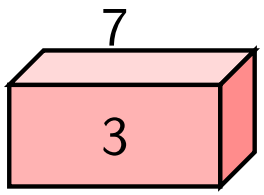
Esempio: tutte le possibili soluzioni del KP (4/6)

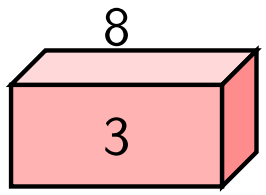
sottoinsieme vuoto



(1)
peso totale $= 0$
profitto totale $= 0$

(2)
peso totale $= 2$
profitto totale $= 6$

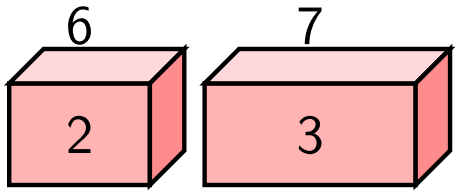




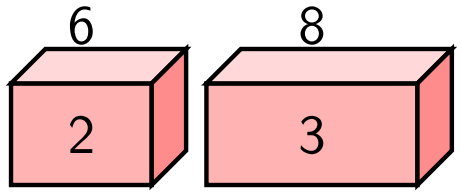
(3)
peso totale $= 3$
profitto totale $= 7$

(4)
peso totale $= 3$
profitto totale $= 8$

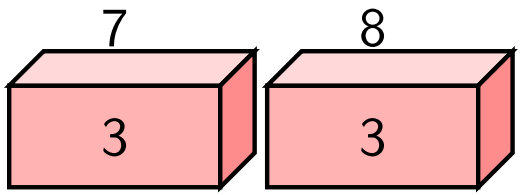
Esempio: tutte le possibili soluzioni del KP (5/6)



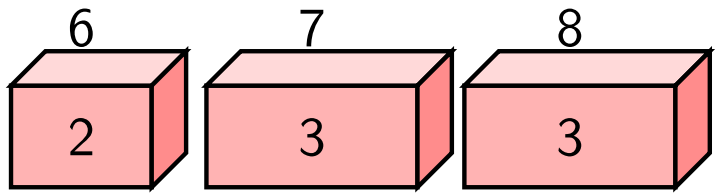
(5)
peso totale = 5
profitto totale = 13



(6)
peso totale = 5
profitto totale = 14



(7)
peso totale = 6
profitto totale = 15
non ammissibile



(8)
peso totale = 8
profitto totale = 21
non ammissibile

Esempio: tutte le possibili soluzioni del KP (6/6)

Etichetta	Vettore soluzione	Peso totale	Profitto totale	Ammissibile?
(1)	$(0,0,0)$	0	0	sì
(2)	$(1,0,0)$	2	6	sì
(3)	$(0,1,0)$	3	7	sì
(4)	$(0,0,1)$	3	8	sì
(5)	$(1,1,0)$	$2+3=5$	$6+7=13$	sì
(6)	$(1,0,1)$	$2+3=5$	$6+8=14$	sì
(7)	$(0,1,1)$	$3+3=6$	$7+8=15$	no
(8)	$(1,1,1)$	$2+3+3=8$	$6+7+8=21$	no

Riepilogo

La prima istanza:

- Uno zaino di capacità $c = 5$ e $n = 3$ oggetti, con profitti $6, 7, 8$ e pesi $2, 3, 3$
- È usata negli esempi del primo algoritmo di PD

Soluzioni ammissibili:

- Una soluzione ammissibile è un vettore binario $(x_1, x_2, \ldots, x_n)$ con $\sum_{j=1}^{n} w_j x_j \leq c$
- Ci sono 2^n sottoinsiemi di oggetti: nella prima istanza 6 degli 8 sottoinsiemi sono ammissibili
- L'enumerazione di tutti i sottoinsiemi diventa presto impraticabile al crescere di n

Un primo algoritmo di PD

Introduzione: un primo algoritmo di PD

- Definiamo i sottoproblemi $v(j, k)$: i primi j oggetti e uno zaino di capacità k
- Scriviamo la loro ricorrenza e verifichiamo la sottostruttura ottima e i sottoproblemi sovrapposti
- Calcoliamo il valore ottimo top-down, bottom-up e con un solo vettore, e poi una soluzione ottima
- Contiamo i passi dell'algoritmo di PD e vediamo una diversa implementazione con liste di stati
- Questo algoritmo di PD calcola esattamente il valore ottimo, senza modello e senza solver

Passo 1: calcolo del valore ottimo

I sottoproblemi $v(j, k)$

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

1. uno zaino di capacità k,
2. un insieme $\{1, 2, \ldots, j\}$ di oggetti disponibili,
3. per ogni oggetto $h \in \{1, 2, \ldots, j\}$, un profitto p_h e un peso w_h.

La ricorrenza di $v(j,k)$

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

$$v(j,k) = \begin{cases} 0 & \text{se } j = 0 \text{ o } k = 0, \\ v(j-1,k) & \text{se } 1 \leq j \leq n, \\ & 1 \leq k \leq c \text{ e } w_j > k, \\ \max\left\{ v(j-1,k), \right. & \text{se } 1 \leq j \leq n, \\ \left. \quad p_j + v(j-1,k-w_j) \right\} & 1 \leq k \leq c,\ w_j \leq k. \end{cases} \tag{1}$$

Lettura della ricorrenza (1/2)

- Il processo di decomposizione ricorsiva della formula (1) genera al più 2 **sottoproblemi indipendenti**, corrispondenti alla scelta di selezionare o no l'oggetto j per uno zaino di capacità k.
- Se $w_j > k$, l'oggetto j non può essere selezionato e quindi l'unica opzione ammissibile è non selezionarlo.
- Qui $v(0, k) = 0$ perché nessun oggetto è disponibile e $v(j, 0) = 0$ perché nessuna capacità è disponibile.

Lettura della ricorrenza (2/2)

- Se $w_j \leq k$, il valore ottimo $v(j,k)$ si ottiene provando le due opzioni possibili, selezionare o non selezionare l'oggetto j, e scegliendo quella che massimizza il profitto.
- I valori corrispondenti sono, rispettivamente, $p_j + v(j-1, k-w_j)$ e $v(j-1,k)$. Se $w_j > k$, l'oggetto j non può essere selezionato e quindi $v(j,k) = v(j-1,k)$.
- Quindi una soluzione ottima della sottoistanza con capacità k e oggetti disponibili $\{1,2,\ldots,j\}$ si costruisce combinando la decisione scelta sull'oggetto j con una soluzione ottima del corrispondente sottoproblema residuo.

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

Sottoproblemi sovrapposti

- Sequenze diverse di decisioni sugli oggetti, prese eventualmente per valori diversi di j e k, possono portare allo stesso sottoproblema.
- Quindi lo stesso valore $v(j, k)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (1).

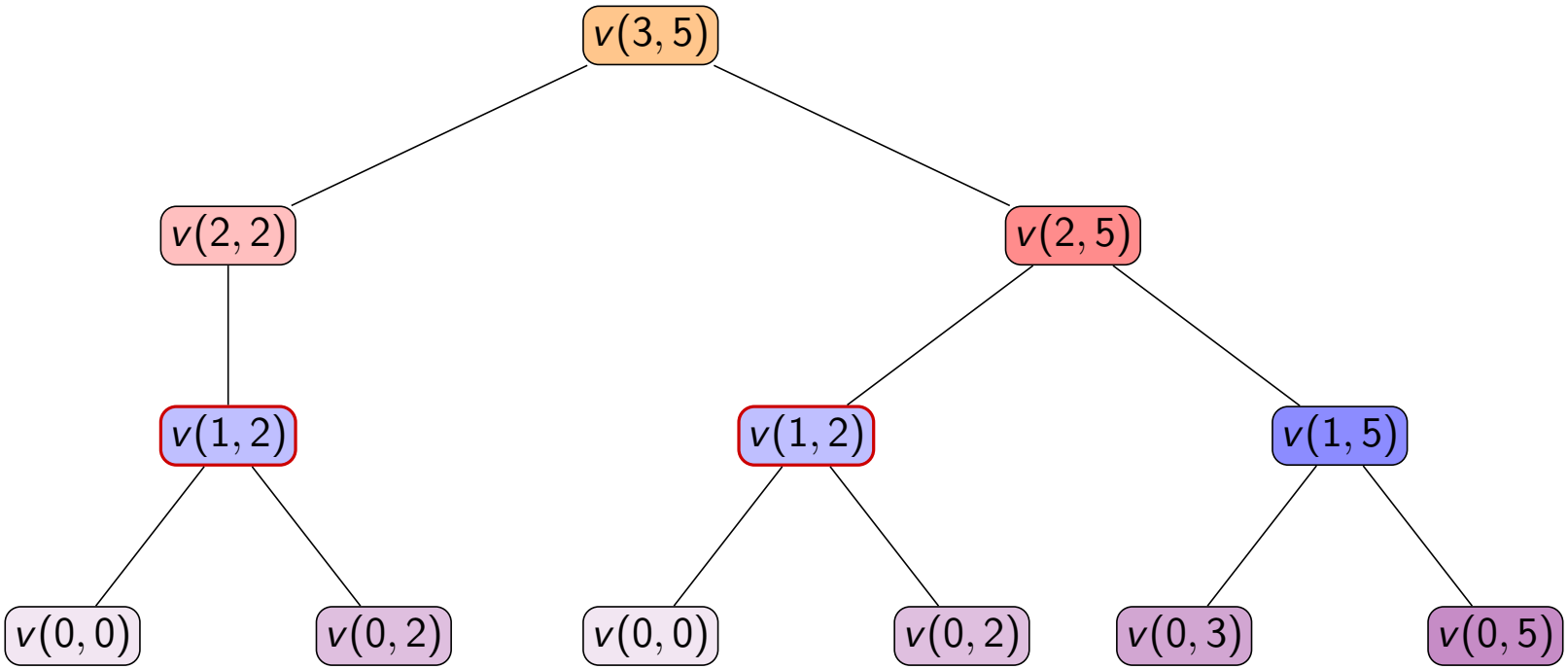
$\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

L'approccio top-down

- Un **primo metodo** per calcolare il valore ottimo $v(n, c)$ dell'istanza del KP è l'**approccio top-down**.
- Il metodo parte dalla chiamata con $j = n$ e $k = c$ e valuta ricorsivamente la formula (1): ogni chiamata considera l'oggetto j e non lo seleziona oppure, se l'oggetto entra nello zaino, considera anche di selezionarlo.
- La ricorsione si ferma quando non restano oggetti o quando la capacità residua è uguale a 0.
- 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

Consideriamo la prima istanza. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(3, 5)$:



La funzione viene chiamata 12 volte e lo stesso sottoproblema viene risolto più di una volta: $v(1, 2)$ viene chiamata 2 volte (con il bordo rosso), e così anche $v(0, 2)$ e $v(0, 0)$.

Ordine di calcolo

- Il valore $v(j, k)$ si può calcolare quando sono noti i valori richiesti dal membro destro della formula (1).
- In particolare, $v(j-1, k)$ serve quando l'oggetto j non è selezionato e, se $w_j \leq k$, $v(j-1, k-w_j)$ serve quando l'oggetto j è selezionato.
- Pertanto, per un valore fissato di j, tutti i valori $v(j, k)$ si possono calcolare considerando i valori di capacità k in ordine crescente.

Esempio: calcolo dei valori della funzione v (1/2)

Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

I valori della funzione v si calcolano come segue:

- I casi base danno $v(0, k) = 0$ per ogni $k \geq 0$.
- Calcolati i casi base, possiamo calcolare $v(1, 2)$:

$$v(1, 2) = \max\{v(0, 2),\ v(0, 0) + p_1\} = \max\{0,\ 0 + 6\} = 6$$

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

- Possiamo calcolare anche $v(1,5)$:

$$v(1,5) = \max\{v(0,5),\ v(0,3) + p_1\} = \max\{0,\ 0+6\} = 6$$

- Poiché $w_2 = 3 > k = 2$, l'oggetto 2 non entra nello zaino, quindi:

$$v(2,2) = v(1,2) = 6$$

- Calcolati $v(1,2)$ e $v(1,5)$, possiamo calcolare $v(2,5)$:

$$v(2,5) = \max\{v(1,5),\ v(1,2) + p_2\} = \max\{6,\ 6+7\} = 13$$

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

$$v(3,5) = \max\{v(2,5),\ v(2,2) + p_3\} = \max\{13,\ 6+8\} = 14$$

Lookup table e approccio bottom-up (1/2)

- Possiamo evitare di risolvere ripetutamente gli stessi sottoproblemi memorizzando il valore di ogni sottoproblema appena calcolato. I valori sono memorizzati in una **lookup table**.
- Poiché la soluzione di ogni sottoproblema dipende solo da sottoproblemi con un valore di j più piccolo, possiamo risolvere i sottoproblemi in ordine crescente di j e, per ogni valore fissato di j, in ordine crescente di capacità.
- Partiamo dai casi base $v(0, k) = 0$ e $v(j, 0) = 0$ fino ad arrivare a $v(n, c)$.
- Questa osservazione porta in modo naturale a un **metodo molto più efficiente** per calcolare il valore ottimo $v(n, c)$ dell'istanza del KP.

Lookup table e approccio bottom-up (2/2)

- Un **secondo metodo** per calcolare il valore ottimo $v(n, c)$ dell'istanza del KP è l'**approccio bottom-up**.
- Il metodo parte dai casi base e valuta la formula (1) per valori crescenti di $j \in \{1, 2, \ldots, n\}$ e $k \in \{0, 1, \ldots, c\}$.
- Ogni valore $v(j, k)$ si calcola con valori già calcolati, fino a ottenere il valore $v(n, c)$ dell'istanza del KP.

Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**:

Algorithm: DP

Input : Un'istanza del KP data dal numero n di oggetti, dai vettori $p[1..n]$ e $w[1..n]$ dei profitti e dei pesi degli oggetti e dalla capacità c dello zaino

Output : Il valore ottimo dell'istanza del KP

```
1 function DP(n, p, w, c)
2 |   v := new array[0..n][0..c]
3 |   for k := 0 to c do
4 |   |   v[0][k] ← 0
5 |   end
```

Algorithm: DP (continua)

```
6   |   for j := 1 to n do
7   |   |   for k := 0 to c do
8   |   |   |   if k < w[j] then
9   |   |   |   |   v[j][k] ← v[j − 1][k]
10  |   |   |   else
11  |   |   |   |   if v[j − 1][k − w[j]] + p[j] > v[j − 1][k] then
12  |   |   |   |   |   v[j][k] ← v[j − 1][k − w[j]] + p[j]
13  |   |   |   |   else
14  |   |   |   |   |   v[j][k] ← v[j − 1][k]
15  |   |   |   |   end
16  |   |   |   end
17  |   |   end
18  |   end
19  |   return v[n][c]
20  end
```

Come funziona l'algoritmo di PD (1/2)

- La matrice $v[0..n][0..c]$ è la **lookup table**: memorizza i valori ottimi di tutti i sottoproblemi.
- Il primo ciclo inizializza i valori $v[0][k] = 0$ per tutte le capacità $k \in \{0, 1, \ldots, c\}$, corrispondenti al caso in cui nessun oggetto è disponibile.
- Il ciclo esterno, sugli oggetti $j \in \{1, 2, \ldots, n\}$, calcola le righe della matrice in ordine crescente di j.
- Il ciclo interno, su $k \in \{0, 1, \ldots, c\}$, calcola il valore ottimo $v[j][k]$ per ogni capacità k.

Come funziona l'algoritmo di PD (2/2)

- Se l'oggetto j non entra nello zaino, allora $v[j][k] \leftarrow v[j-1][k]$.
- Altrimenti, l'algoritmo di PD confronta il valore ottenuto non selezionando l'oggetto j, cioè $v[j-1][k]$, con il valore ottenuto selezionando l'oggetto j, cioè $p[j] + v[j-1][k-w[j]]$.
- Infine, l'algoritmo restituisce il valore ottimo dell'istanza del KP, $v[n][c]$.

Esempio: esecuzione dell'algoritmo di PD (1/4)

Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di j e k, la cella $v[j][k]$ e l'operazione eseguita.

j	k	$v[j][k]$	Azione
0	0	0	$v[0][0] \leftarrow 0$
0	1	0	$v[0][1] \leftarrow 0$
0	2	0	$v[0][2] \leftarrow 0$
0	3	0	$v[0][3] \leftarrow 0$
0	4	0	$v[0][4] \leftarrow 0$
0	5	0	$v[0][5] \leftarrow 0$

Esempio: esecuzione dell'algoritmo di PD (2/4)

j	k	$v[j][k]$	Azione
1	0	0	$v[1][0] \leftarrow v[0][0] = 0$
1	1	0	$v[1][1] \leftarrow v[0][1] = 0$
1	2	6	$\max\{v[0][2], p[1] + v[0][0]\} = \max\{0, 6 + 0\} = 6$
1	3	6	$\max\{v[0][3], p[1] + v[0][1]\} = \max\{0, 6 + 0\} = 6$
1	4	6	$\max\{v[0][4], p[1] + v[0][2]\} = \max\{0, 6 + 0\} = 6$
1	5	6	$\max\{v[0][5], p[1] + v[0][3]\} = \max\{0, 6 + 0\} = 6$
2	0	0	$v[2][0] \leftarrow v[1][0] = 0$
2	1	0	$v[2][1] \leftarrow v[1][1] = 0$
2	2	6	$v[2][2] \leftarrow v[1][2] = 6$
2	3	7	$\max\{v[1][3], p[2] + v[1][0]\} = \max\{6, 7 + 0\} = 7$
2	4	7	$\max\{v[1][4], p[2] + v[1][1]\} = \max\{6, 7 + 0\} = 7$
2	5	13	$\max\{v[1][5], p[2] + v[1][2]\} = \max\{6, 7 + 6\} = 13$

Esempio: esecuzione dell'algoritmo di PD (3/4)

j	k	$v[j][k]$	Azione
3	0	0	$v[3][0] \leftarrow v[2][0] = 0$
3	1	0	$v[3][1] \leftarrow v[2][1] = 0$
3	2	6	$v[3][2] \leftarrow v[2][2] = 6$
3	3	8	$\max\{v[2][3], p[3] + v[2][0]\} = \max\{7, 8 + 0\} = 8$
3	4	8	$\max\{v[2][4], p[3] + v[2][1]\} = \max\{7, 8 + 0\} = 8$
3	5	14	$\max\{v[2][5], p[3] + v[2][2]\} = \max\{13, 8 + 6\} = 14$

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

Dopo l'esecuzione dell'algoritmo di PD, questa è la matrice v:

$v[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$j=0$	0	0	0	0	0	0
$j=1$	0	0	6	6	6	6
$j=2$	0	0	6	7	7	13
$j=3$	0	0	6	8	8	14

Dalla matrice a un solo vettore

- Dopo l'esecuzione dell'algoritmo di PD, la matrice v contiene il valore ottimo $v[j][k] = v(j, k)$ per ogni oggetto $j \in \{0, 1, \ldots, n\}$ e ogni valore di capacità $k \in \{0, 1, \ldots, c\}$.
- L'algoritmo di PD calcola la riga j della matrice v usando solo la riga $j - 1$.
- Pertanto, quando serve solo il valore ottimo dell'istanza del KP, non è necessario memorizzare tutta la matrice: basta un solo vettore $v[0..c]$, che usa $c + 1$ celle invece delle $(n + 1)(c + 1)$ celle della matrice.

Un solo vettore (1/2)

- Il vettore si inizializza con i casi base $v[k] = 0$ per ogni $k \in \{0, 1, \ldots, c\}$, che corrispondono alla riga $j = 0$ della matrice.
- Poi, per ogni oggetto $j \in \{1, 2, \ldots, n\}$, il vettore viene aggiornato in modo da contenere la riga j della matrice.
- L'ordine di aggiornamento delle celle è essenziale. Quando la cella $v[k]$ viene aggiornata per l'oggetto j, il valore $v[k - w_j]$ deve essere ancora quello della riga $j - 1$, cioè $v(j - 1, k - w_j)$.

Un solo vettore (2/2)

- Poiché $k - w_j < k$, questo è garantito se i valori di capacità sono considerati in ordine **decrescente** $k = c, c-1, \ldots, w_j$: quando la cella k viene aggiornata, la cella $k - w_j$ non è ancora stata aggiornata nella passata corrente.
- Se i valori di capacità fossero considerati in ordine crescente, il valore $v[k - w_j]$ sarebbe già quello della riga j, cioè $v(j, k - w_j)$, e l'oggetto j potrebbe essere selezionato più di una volta, cosa non consentita nel KP.
- Per ogni valore di capacità $k < w_j$, l'oggetto j non entra nello zaino e quindi $v[k]$ deve mantenere il valore $v(j-1, k)$: non si esegue alcun aggiornamento e il ciclo interno può fermarsi a $k = w_j$.

Questo è lo pseudocodice dell'**algoritmo di PD con un solo vettore**:

Algorithm: DP_Vector

Input : Un'istanza del KP data dal numero n di oggetti, dai vettori $p[1..n]$ e $w[1..n]$ dei profitti e dei pesi degli oggetti e dalla capacità c dello zaino

Output : Il valore ottimo dell'istanza del KP

```
1 function DP_Vector(n, p, w, c)
2 |   v := new array[0..c]
3 |   for k := 0 to c do
4 |   |   v[k] ← 0
5 |   end
```

Algorithm: DP_Vector (continua)

```
6   |   for j := 1 to n do
7   |   |   for k := c downto w[j] do
8   |   |   |   if p[j] + v[k − w[j]] > v[k] then
9   |   |   |   |   v[k] ← p[j] + v[k − w[j]]
10  |   |   |   end
11  |   |   end
12  |   end
13  |   return v[c]
14  end
```

Come funziona DP_Vector (1/2)

- Il vettore $v[0..c]$ è la **lookup table**: al termine della passata per l'oggetto j, memorizza la riga j della matrice v dell'algoritmo di PD.
- Il primo ciclo inizializza i valori $v[k] = 0$ per tutte le capacità $k \in \{0, 1, \ldots, c\}$, corrispondenti al caso in cui nessun oggetto è disponibile.
- Il ciclo esterno, sugli oggetti $j \in \{1, 2, \ldots, n\}$, esegue una passata del vettore per ogni oggetto.

Come funziona DP_Vector (2/2)

- Il ciclo interno, su $k \in \{w[j], w[j]+1, \ldots, c\}$, considera i valori di capacità in ordine decrescente, in modo che le celle che servono ancora con il valore precedente non siano ancora state modificate.
- La condizione confronta il valore ottenuto non selezionando l'oggetto j, cioè il valore corrente $v[k]$, con il valore ottenuto selezionando l'oggetto j, cioè $p[j] + v[k - w[j]]$. La cella viene aggiornata solo quando selezionare l'oggetto j è strettamente migliore.
- Infine, l'algoritmo restituisce il valore ottimo dell'istanza del KP, $v[c]$.

Esempio: esecuzione di DP con un solo vettore (1/3)

Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

Il primo ciclo inizializza $v[k] = 0$ per ogni $k \in \{0, 1, \ldots, 5\}$. Per ogni passo dei due cicli annidati, la tabella riporta i valori di j e k, la cella $v[k]$ dopo il passo e l'operazione eseguita.

j	k	$v[k]$	Azione
1	5	6	$p[1] + v[3] = 6 + 0 > v[5] = 0$: $v[5] \leftarrow 6$
1	4	6	$p[1] + v[2] = 6 + 0 > v[4] = 0$: $v[4] \leftarrow 6$
1	3	6	$p[1] + v[1] = 6 + 0 > v[3] = 0$: $v[3] \leftarrow 6$
1	2	6	$p[1] + v[0] = 6 + 0 > v[2] = 0$: $v[2] \leftarrow 6$

Esempio: esecuzione di DP con un solo vettore (2/3)

j	k	$v[k]$	Azione
2	5	13	$p[2]+v[2]=7+6>v[5]=6$: $v[5]\leftarrow 13$
2	4	7	$p[2]+v[1]=7+0>v[4]=6$: $v[4]\leftarrow 7$
2	3	7	$p[2]+v[0]=7+0>v[3]=6$: $v[3]\leftarrow 7$
3	5	14	$p[3]+v[2]=8+6>v[5]=13$: $v[5]\leftarrow 14$
3	4	8	$p[3]+v[1]=8+0>v[4]=7$: $v[4]\leftarrow 8$
3	3	8	$p[3]+v[0]=8+0>v[3]=7$: $v[3]\leftarrow 8$

Esempio: esecuzione di DP con un solo vettore (3/3)

Questo è il contenuto del vettore v dopo l'inizializzazione e dopo ogni passata del ciclo esterno:

$v[k]$	0	1	2	3	4	5
inizializzazione	0	0	0	0	0	0
passata $j = 1$	0	0	6	6	6	6
passata $j = 2$	0	0	6	7	7	13
passata $j = 3$	0	0	6	8	8	14

Ogni passata trasforma il vettore v dalla riga $j - 1$ nella riga j della matrice v calcolata dall'algoritmo di PD sulla stessa istanza. Il valore ottimo dell'istanza del KP è $v[5] = 14$.

Il vettore calcola solo il valore ottimo

- La versione con un vettore calcola solo il valore ottimo dell'istanza del KP.
- L'informazione necessaria per costruire una soluzione ottima, descritta nel Passo 2, registra la decisione presa per ogni oggetto j e ogni valore di capacità k, e quindi non si può memorizzare in un solo vettore.

Passo 2: calcolo di una soluzione ottima

La matrice s delle decisioni

- Per calcolare una soluzione ottima, cioè il vettore binario $(x_1, x_2, \ldots, x_n)$, è necessario memorizzare alcune informazioni aggiuntive.
- Per ogni oggetto j e valore di capacità k, oltre al valore ottimo $v[j][k]$, memorizziamo $s[j][k]$, che indica se l'oggetto j è selezionato in una soluzione ottima del sottoproblema ristretto agli oggetti $\{1, 2, \ldots, j\}$ e alla capacità k.
- Più precisamente, $s[j][k]$ è uguale a 1 se l'oggetto j è selezionato nel calcolo del valore $v[j][k]$, ed è uguale a 0 altrimenti.

Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, con la memorizzazione delle informazioni necessarie per costruire una soluzione ottima:

Algorithm: DP_InfoSol

Input : Un'istanza del KP data dal numero n di oggetti, dai vettori $p[1..n]$ e $w[1..n]$ dei profitti e dei pesi degli oggetti e dalla capacità c dello zaino

Output : Il valore ottimo dell'istanza del KP e la matrice delle informazioni binarie $s[0..n][0..c]$

```
1 function DP_InfoSol(n, p, w, c)
2     v := new array[0..n][0..c]
3     s := new array[0..n][0..c]
4     for k := 0 to c do
5         v[0][k] ← 0
6         s[0][k] ← 0
7     end
```

Algorithm: DP_InfoSol (continua)

```
8   for j := 1 to n do
9       for k := 0 to c do
10          if k < w[j] then
11              v[j][k] ← v[j − 1][k]
12              s[j][k] ← 0
13          else
14              if v[j − 1][k − w[j]] + p[j] > v[j − 1][k] then
15                  v[j][k] ← v[j − 1][k − w[j]] + p[j]
16                  s[j][k] ← 1
17              else
18                  v[j][k] ← v[j − 1][k]
19                  s[j][k] ← 0
20              end
21          end
22      end
23  end
```

Algorithm: DP_InfoSol (continua)

```
24      return v[n][c], s
25  end
```

- L'algoritmo di PD usa la matrice aggiuntiva s, che registra la decisione presa quando si calcola $v[j][k]$.
- Se l'oggetto j è selezionato, $s[j][k] \leftarrow 1$ memorizza questa decisione; altrimenti, $s[j][k] \leftarrow 0$ memorizza che l'oggetto j non è selezionato.

Esempio: calcolo delle informazioni aggiuntive (1/4)

Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di j e k, la cella $s[j][k]$ e l'operazione eseguita.

j	k	$s[j][k]$	Azione
0	0	0	$s[0][0] \leftarrow 0$
0	1	0	$s[0][1] \leftarrow 0$
0	2	0	$s[0][2] \leftarrow 0$
0	3	0	$s[0][3] \leftarrow 0$
0	4	0	$s[0][4] \leftarrow 0$
0	5	0	$s[0][5] \leftarrow 0$

Esempio: calcolo delle informazioni aggiuntive (2/4)

j	k	$s[j][k]$	Azione
1	0	0	$0 < 2$: $s[1][0] \leftarrow 0$
1	1	0	$1 < 2$: $s[1][1] \leftarrow 0$
1	2	1	$p[1] + v[0][0] = 6 + 0 > v[0][2] = 0$: $s[1][2] \leftarrow 1$
1	3	1	$p[1] + v[0][1] = 6 + 0 > v[0][3] = 0$: $s[1][3] \leftarrow 1$
1	4	1	$p[1] + v[0][2] = 6 + 0 > v[0][4] = 0$: $s[1][4] \leftarrow 1$
1	5	1	$p[1] + v[0][3] = 6 + 0 > v[0][5] = 0$: $s[1][5] \leftarrow 1$
2	0	0	$0 < 3$: $s[2][0] \leftarrow 0$
2	1	0	$1 < 3$: $s[2][1] \leftarrow 0$
2	2	0	$2 < 3$: $s[2][2] \leftarrow 0$
2	3	1	$p[2] + v[1][0] = 7 + 0 > v[1][3] = 6$: $s[2][3] \leftarrow 1$
2	4	1	$p[2] + v[1][1] = 7 + 0 > v[1][4] = 6$: $s[2][4] \leftarrow 1$
2	5	1	$p[2] + v[1][2] = 7 + 6 > v[1][5] = 6$: $s[2][5] \leftarrow 1$

Esempio: calcolo delle informazioni aggiuntive (3/4)

j	k	$s[j][k]$	Azione
3	0	0	$0 < 3$: $s[3][0] \leftarrow 0$
3	1	0	$1 < 3$: $s[3][1] \leftarrow 0$
3	2	0	$2 < 3$: $s[3][2] \leftarrow 0$
3	3	1	$p[3] + v[2][0] = 8 + 0 > v[2][3] = 7$: $s[3][3] \leftarrow 1$
3	4	1	$p[3] + v[2][1] = 8 + 0 > v[2][4] = 7$: $s[3][4] \leftarrow 1$
3	5	1	$p[3] + v[2][2] = 8 + 6 > v[2][5] = 13$: $s[3][5] \leftarrow 1$

Esempio: calcolo delle informazioni aggiuntive (4/4)

Questa è la matrice s dopo l'esecuzione dell'algoritmo di PD:

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$j=0$	0	0	0	0	0	0
$j=1$	0	0	1	1	1	1
$j=2$	0	0	0	1	1	1
$j=3$	0	0	0	1	1	1

Ricostruzione di una soluzione ottima

- Calcolata la matrice s, si può ricostruire una soluzione ottima scorrendo gli oggetti da $j = n$ fino a $j = 1$, partendo dalla capacità $k = c$ e da $x_j = 0$ per ogni $j \in \{1, 2, \ldots, n\}$:

$$\begin{aligned} x_j &\leftarrow 1, \quad k \leftarrow k - w_j \quad \text{se } s[j][k] = 1, \\ x_j &\leftarrow 0 \qquad\qquad\qquad\quad \text{se } s[j][k] = 0. \end{aligned}$$

- Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, dove x_j è uguale a 1 se l'oggetto j è selezionato in una soluzione ottima, e a 0 altrimenti.

Algorithm: DP_Reconstruct

Input : Il numero n di oggetti, la capacità c dello zaino, il vettore $w[1..n]$ dei pesi degli oggetti e la matrice $s[0..n][0..c]$

Output : Una soluzione ottima dell'istanza del KP

```
1 function DP_Reconstruct(n, c, w, s)
2 |   x := new array[1..n]
3 |   for j := 1 to n do
4 |   |   x[j] ← 0
5 |   end
6 |   k := c
```

Algorithm: DP_Reconstruct (continua)

```
7     for j := n downto 1 do
8         if s[j][k] = 1 then
9             x[j] ← 1
10            k ← k − w[j]
11        end
12    end
13    return x
14 end
```

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

Dopo l'esecuzione dell'algoritmo di PD sulla prima istanza, abbiamo la seguente matrice s:

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$j=0$	0	0	0	0	0	0
$j=1$	0	0	1	1	1	1
$j=2$	0	0	0	1	1	1
$j=3$	0	0	0	1	1	1

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

Possiamo ricostruire una soluzione ottima partendo da $j = 3$ e $k = 5$ come segue:

Iterazione	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	Azione
–	–	–	–	0	0	0	$k = 5$
1	3	5	1	0	0	1	$k \leftarrow 5 - 3 = 2$
2	2	2	0	0	0	1	nessun aggiornamento
3	1	2	1	1	0	1	$k \leftarrow 2 - 2 = 0$

Quindi il vettore

$$(1, 0, 1)$$

è una soluzione ottima. Il valore ottimo corrispondente è $6 + 8 = 14$.

Quanti passi esegue l'algoritmo di PD?

Contiamo quante volte viene eseguita ogni riga dello pseudocodice di `DP`.

Numero di passi di DP (1/3)

- Indichiamo con $S(n, c)$ il numero di passi eseguiti da `DP` nel caso peggiore, dove n è il numero di oggetti e c è la capacità dello zaino.
- Calcoliamo $S(n, c)$ contando quante volte vengono eseguite le righe dello pseudocodice di `DP`.
- Nel caso peggiore, la condizione della riga 11 viene verificata ogni volta che l'oggetto entra nello zaino. Poiché i pesi degli oggetti sono interi positivi, questo accade al più nc volte.

Numero di passi di DP (2/3)

Il numero di esecuzioni delle righe rilevanti è:

- La Riga 2 viene eseguita 1 volta.
- La Riga 3 viene eseguita $c + 1$ volte.
- La Riga 4 viene eseguita $c + 1$ volte.
- La Riga 6 viene eseguita n volte.
- La Riga 7 viene eseguita $n(c + 1)$ volte.
- La Riga 8 viene eseguita $n(c + 1)$ volte.
- La Riga 11 viene eseguita al più nc volte.
- una fra la Riga 9, la Riga 12 e la Riga 14 viene eseguita per ogni coppia (j, k), e quindi queste righe di assegnamento vengono eseguite in tutto $n(c + 1)$ volte.
- La Riga 19 viene eseguita 1 volta.

Numero di passi di DP (3/3)

Pertanto, il numero totale di passi nel caso peggiore è

$$1 + (c+1) + (c+1) + n + n(c+1) + n(c+1) + nc + n(c+1) + 1.$$

Quindi

$$S(n, c) = 4nc + 4n + 2c + 4.$$

Dunque

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

Infatti

$$\lim_{n,c\to\infty} \frac{4nc + 4n + 2c + 4}{nc} = \lim_{n,c\to\infty} \left(4 + \frac{4}{c} + \frac{2}{n} + \frac{4}{nc}\right)$$
$$= 4 \in \mathbb{R}_{>0}.$$

Il numero di passi delle tre versioni

L'algoritmo di PD esegue

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

passi nel caso peggiore.

- La versione `DP_InfoSol` memorizza la matrice aggiuntiva s ed esegue solo un numero costante di operazioni in più dentro gli stessi cicli. Pertanto, il suo numero di passi resta $\Theta(nc)$.
- La versione `DP_Vector` usa un solo vettore invece della matrice ed esegue gli stessi due cicli annidati, il cui ciclo interno esegue al più $c + 1$ iterazioni per ogni oggetto. Pertanto, il suo numero di passi resta $\Theta(nc)$, mentre il numero di celle della lookup table scende da $(n + 1)(c + 1)$ a $c + 1$.

Un'implementazione con liste

Passo 1: calcolo del valore ottimo

Perché le liste?

- L'algoritmo di PD di questa sezione memorizza un valore per ogni coppia formata da un oggetto e da un valore di capacità, cioè $(n+1)(c+1)$ valori in tutto.
- Ogni riga della matrice v è non decrescente nel valore di capacità e cambia solo in pochi valori di capacità: tutte le altre celle ripetono il valore della cella alla loro sinistra.
- Vediamo ora una **diversa implementazione dello stesso algoritmo di PD**. La decomposizione ricorsiva e la sottostruttura ottima sono quelle della formula (1): cambia la struttura dati.
- Invece di un'intera riga della matrice, memorizziamo solo le coppie in cui la riga cambia, cioè solo le coppie ottenute effettivamente da un sottoinsieme di oggetti. Queste coppie sono tenute in una lista, una lista per ogni oggetto.

Stati

Uno **stato** è una coppia (w, p), dove w è il peso totale e p è il profitto totale di un sottoinsieme di oggetti il cui peso totale non supera la capacità c dello zaino.

- Il sottoinsieme vuoto dà lo stato $(0, 0)$. Un sottoinsieme degli oggetti $\{1, 2, \ldots, j\}$ dà lo stato ottenuto sommando i pesi e i profitti dei suoi oggetti.
- Due sottoinsiemi diversi possono dare lo stesso stato. In questo caso i due sottoinsiemi sono intercambiabili: occupano lo stesso peso nello zaino e danno lo stesso profitto, quindi basta tenere un solo stato.
- Più in generale, uno stato si può scartare quando un altro stato è almeno altrettanto buono in entrambe le componenti.

Stati dominati

Dati due stati (w, p) e (w', p') con $(w, p) \neq (w', p')$, lo stato (w, p) **domina** lo stato (w', p') se

$$w \leq w' \qquad \text{e} \qquad p \geq p'.$$

Uno stato è **non dominato** se nessun altro stato lo domina.

- Uno stato dominato si può scartare. Infatti, supponiamo che (w, p) domini (w', p') e sia S' un sottoinsieme degli oggetti rimanenti che si può aggiungere a (w', p') senza superare la capacità.
- Poiché $w \leq w'$, lo stesso sottoinsieme S' si può aggiungere anche a (w, p) e, poiché $p \geq p'$, il profitto totale risultante non è minore. Pertanto nessuna soluzione ottima va persa.

Le liste L_j

Per ogni $j \in \{0, 1, \ldots, n\}$, sia L_j la lista degli stati non dominati della sottoistanza del KP data da:

1. uno zaino di capacità c,
2. un insieme $\{1, 2, \ldots, j\}$ di oggetti disponibili,
3. per ogni oggetto $h \in \{1, 2, \ldots, j\}$, un profitto p_h e un peso w_h.

Gli stati di L_j sono ordinati per peso strettamente crescente.

La ricorrenza di L_j

La formula ricorsiva per calcolare L_j è la seguente:

$$L_j = \begin{cases} \{(0,0)\} & \text{se } j = 0, \\ \mathrm{nd}\big(L_{j-1} \cup M_j\big) & \text{se } 1 \le j \le n, \end{cases} \tag{2}$$

dove $\mathrm{nd}(S)$ indica la lista degli stati non dominati di un insieme S di stati, e

$$M_j = \big\{\, (w + w_j,\ p + p_j) \ :\ (w,p) \in L_{j-1},\ w + w_j \le c \,\big\}.$$

Lettura della ricorrenza di L_j (1/2)

- Il processo di decomposizione ricorsiva della formula (2) genera al più 2 **sottoproblemi indipendenti** per ogni stato di L_{j-1}, corrispondenti alla scelta di selezionare o no l'oggetto j.
- Se l'oggetto j non è selezionato, lo stato non cambia e appartiene a L_{j-1}.
- Se l'oggetto j è selezionato, lo stato diventa $(w + w_j, p + p_j)$ e appartiene a M_j; questa opzione è disponibile solo quando $w + w_j \leq c$.
- Qui $L_0 = \{(0, 0)\}$ perché l'unico sottoinsieme di un insieme vuoto di oggetti è il sottoinsieme vuoto.

Lettura della ricorrenza di L_j (2/2)

- Quindi ogni stato non dominato raggiungibile con gli oggetti $\{1, 2, \ldots, j\}$ si costruisce combinando la decisione scelta sull'oggetto j con uno stato non dominato raggiungibile con gli oggetti $\{1, 2, \ldots, j-1\}$.

$$\rightsquigarrow \textbf{Sottostruttura ottima verificata!}$$

- Sottoinsiemi diversi degli oggetti $\{1, 2, \ldots, j\}$ possono dare lo stesso stato, e uno stato può anche essere dominato da un altro. In entrambi i casi in L_j si tiene un solo stato, e tutti i sottoinsiemi che portano a esso vengono trattati insieme nelle iterazioni successive.

$$\rightsquigarrow \textbf{Sottoproblemi sovrapposti verificati!}$$

Il valore ottimo dall'ultima lista

Il valore ottimo dell'istanza del KP è il profitto più grande memorizzato nell'ultima lista:

$$\max\left\{ p \ : \ (w,p) \in L_n \right\}. \tag{3}$$

- In una lista di stati non dominati ordinata per peso strettamente crescente, anche i profitti sono strettamente crescenti: se uno stato avesse un peso maggiore di un altro e un profitto non maggiore, sarebbe dominato.
- Pertanto il massimo della formula (3) è il profitto dell'ultimo stato di L_n.

Le liste e le righe della matrice v

La lista L_j contiene in forma compressa l'informazione della riga j della matrice v:

$$v[j][k] = \max\left\{ p \ : \ (w,p) \in L_j, \ w \leq k \right\} \qquad \forall k \in \{0,1,\ldots,c\}.$$

Ogni stato (w,p) di L_j indica, con il suo peso, un valore di capacità in cui la riga aumenta, e p è il nuovo valore.

Esempio: le liste di stati (1/3)

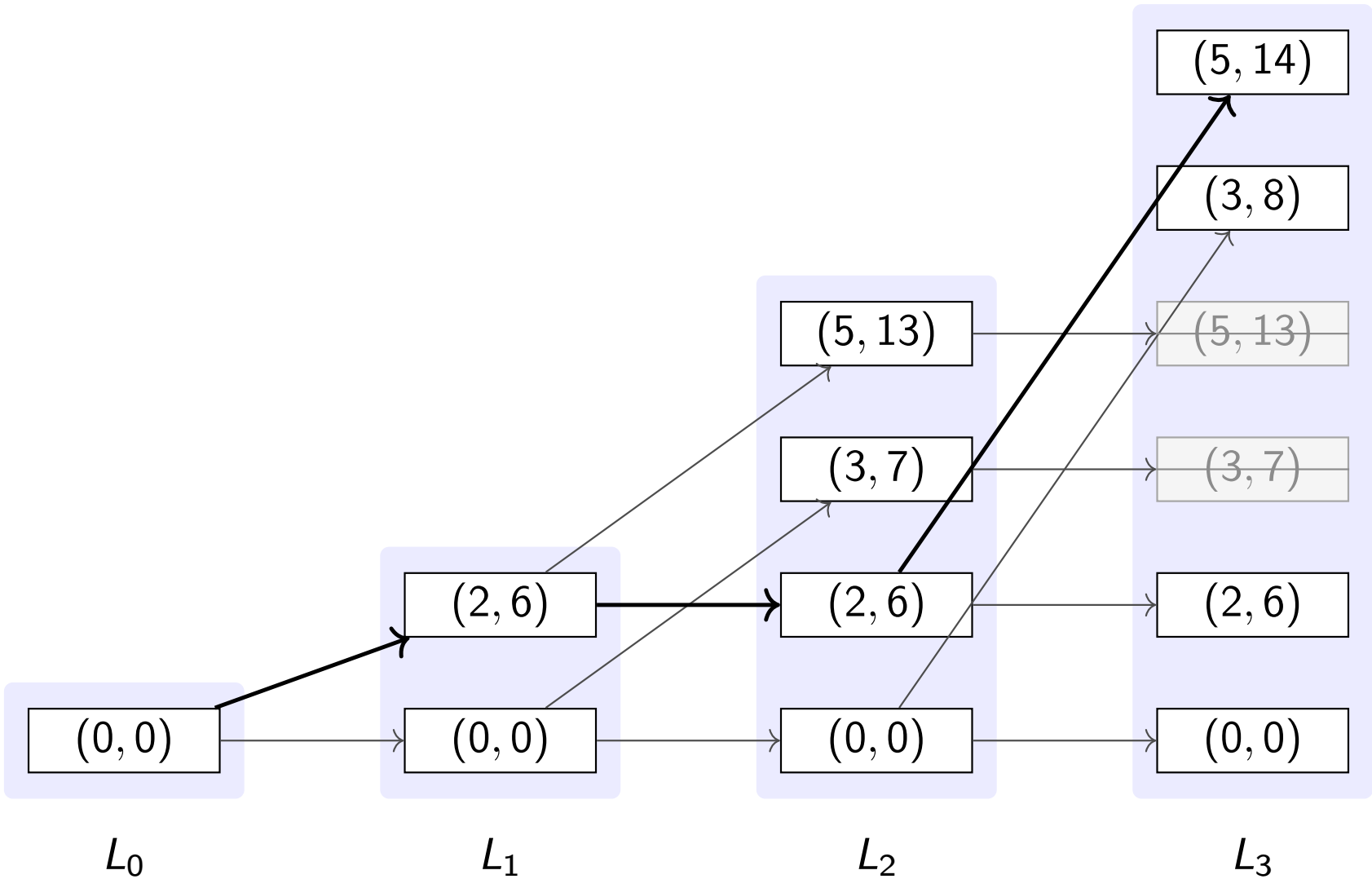
Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

Nella figura della slide successiva:

- ogni fascia ombreggiata raccoglie gli stati di una lista;
- un arco che entra nella fascia di L_j corrisponde alla decisione presa sull'oggetto j: un arco orizzontale indica che l'oggetto j non è selezionato, mentre un arco obliquo indica che l'oggetto j è selezionato;
- gli archi spessi danno gli stati di una soluzione ottima.

Esempio: le liste di stati (2/3)



Esempio: le liste di stati (3/3)

- I due stati barrati dell'ultima fascia sono dominati e quindi vengono scartati: $(3, 7)$ è dominato da $(3, 8)$ e $(5, 13)$ è dominato da $(5, 14)$.
- Selezionare l'oggetto 3 a partire dagli stati $(3, 7)$ e $(5, 13)$ di L_2 darebbe i pesi totali 6 e 8, che superano la capacità $c = 5$ dello zaino.
- Quindi da questi due stati non esce alcun arco verso gli stati di M_3.

Il calcolo delle liste

- Le liste si calcolano per valori crescenti di $j \in \{1, 2, \ldots, n\}$, partendo da $L_0 = \{(0, 0)\}$: ogni lista L_j si calcola dalla lista L_{j-1}, già calcolata, fino a ottenere la lista L_n.
- La lista L_{j-1} e la lista M_j sono entrambe ordinate per peso crescente, perché M_j si ottiene da L_{j-1} aggiungendo lo stesso peso w_j a ogni stato.
- Pertanto l'unione della formula (2) si calcola scorrendo contemporaneamente le due liste, come nella fusione di due liste ordinate.

L'ordine di scansione

Per descrivere l'ordine di scansione usiamo la seguente convenzione.

Lo stato (w, p) **precede** lo stato (w', p') se

$$w < w' \qquad \text{o} \qquad (w = w' \text{ e } p > p').$$

- Se gli stati vengono scanditi in quest'ordine, uno stato è non dominato se e solo se il suo profitto è strettamente maggiore del profitto più grande degli stati già scanditi.
- Infatti, ogni stato già scandito ha un peso non maggiore, e quindi domina lo stato corrente non appena il suo profitto non è minore.

Questo è lo pseudocodice dell'**algoritmo di PD con liste**:

Algorithm: DP_Lists

Input : Un'istanza del KP data dal numero n di oggetti, dai vettori $p[1..n]$ e $w[1..n]$ dei profitti e dei pesi degli oggetti e dalla capacità c dello zaino

Output : Il valore ottimo dell'istanza del KP

```
1 function DP_Lists(n, p, w, c)
2 |   L := new list containing the single state (0, 0)
```

Algorithm: DP_Lists (continua)

```
3     for j := 1 to n do
4         M := new empty list
5         for h := 1 to |L| do
6             if L[h].w + w[j] ≤ c then
7                 accodare (L[h].w + w[j], L[h].p + p[j]) a M
8             end
9         end
10        L ← Merge(L, M)
11    end
12    return L[|L|].p
13 end
```

Questo è lo pseudocodice della procedura che fonde due liste e scarta gli stati dominati:

Algorithm: Merge

Input : Due liste L e M di stati, entrambe ordinate per peso crescente

Output : La lista degli stati non dominati di $L \cup M$, ordinata per peso crescente

```
1 function Merge(L, M)
2     R := new empty list
3     h := 1
4     k := 1
5     π := −1
```

Algorithm: Merge (continua)

```
6     while h ≤ |L| o k ≤ |M| do
7         if k > |M| o (h ≤ |L| e L[h] precede M[k]) then
8             (a, b) ← L[h]
9             h ← h + 1
10        else
11            (a, b) ← M[k]
12            k ← k + 1
13        end
14        if b > π then
15            accodare (a, b) a R
16            π ← b
17        end
18    end
19    return R
20 end
```

Come funziona DP_Lists (1/2)

- La lista L è la **lookup table**: al termine dell'iterazione per l'oggetto j, memorizza la lista L_j.
- La lista L si inizializza con il solo stato $(0, 0)$, cioè con la lista L_0.
- Il ciclo esterno, sugli oggetti $j \in \{1, 2, \ldots, n\}$, calcola una lista per ogni oggetto.
- Il ciclo interno costruisce la lista M degli stati ottenuti selezionando l'oggetto j. Uno stato viene aggiunto a M solo quando il peso totale risultante non supera la capacità dello zaino.

Come funziona DP_Lists (2/2)

- La procedura `Merge` scorre contemporaneamente le due liste e tiene uno stato solo quando il suo profitto è strettamente maggiore del profitto più grande già accodato, memorizzato nella variabile ausiliaria π.
- Il valore iniziale $\pi = -1$ garantisce che il primo stato scandito venga sempre accodato.
- Poiché gli stati di L sono ordinati per peso strettamente crescente e profitto strettamente crescente, il profitto più grande è quello dell'ultimo stato. Pertanto l'algoritmo restituisce il valore ottimo dell'istanza del KP, $L[|L|].p$.
- Dopo l'esecuzione dell'algoritmo di PD, la lista L contiene gli stati non dominati L_n dell'istanza del KP, ordinati per peso strettamente crescente e profitto strettamente crescente.

Esempio: esecuzione dell'algoritmo di PD con liste (1/3)

Consideriamo la prima istanza, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

Per ogni oggetto j, la tabella riporta la lista M_j degli stati ottenuti selezionando l'oggetto j e la lista L_j risultante.

j	M_j	L_j
0	–	$\{(0,0)\}$
1	$\{(2,6)\}$	$\{(0,0),(2,6)\}$
2	$\{(3,7),(5,13)\}$	$\{(0,0),(2,6),(3,7),(5,13)\}$
3	$\{(3,8),(5,14)\}$	$\{(0,0),(2,6),(3,8),(5,14)\}$

Esempio: esecuzione dell'algoritmo di PD con liste (2/3)

Per ogni stato scandito da `Merge`, la tabella riporta il valore di j, lo stato, la lista da cui è preso e l'operazione eseguita.

j	stato	da	Azione
1	$(0,0)$	L_0	$0 > \pi = -1$: accodare
1	$(2,6)$	M_1	$6 > \pi = 0$: accodare
2	$(0,0)$	L_1	$0 > \pi = -1$: accodare
2	$(2,6)$	L_1	$6 > \pi = 0$: accodare
2	$(3,7)$	M_2	$7 > \pi = 6$: accodare
2	$(5,13)$	M_2	$13 > \pi = 7$: accodare

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

j	stato	da	Azione
3	$(0,0)$	L_2	$0 > \pi = -1$: accodare
3	$(2,6)$	L_2	$6 > \pi = 0$: accodare
3	$(3,8)$	M_3	$8 > \pi = 6$: accodare
3	$(3,7)$	L_2	$7 \leq \pi = 8$: scartare (dominato)
3	$(5,14)$	M_3	$14 > \pi = 8$: accodare
3	$(5,13)$	L_2	$13 \leq \pi = 14$: scartare (dominato)

L'ultimo stato di L_3 è $(5,14)$, e quindi il valore ottimo dell'istanza del KP è 14.

Un'implementazione con liste

Passo 2: calcolo di una soluzione ottima

Le informazioni aggiuntive per le liste (1/2)

- Per calcolare una soluzione ottima, cioè il vettore binario $(x_1, x_2, \ldots, x_n)$, è necessario memorizzare alcune informazioni aggiuntive.
- La sola lista L_n dà il valore ottimo, ma non dice quali oggetti producono lo stato che lo raggiunge.
- Per ogni oggetto j e ogni posizione $h \in \{1, 2, \ldots, |L_j|\}$, oltre allo stato $L_j[h]$, memorizziamo due valori:
  - $\sigma_j[h]$, che è uguale a 1 se l'oggetto j è selezionato nello stato $L_j[h]$, ed è uguale a 0 altrimenti;
  - $\rho_j[h]$, che è la posizione in L_{j-1} dello stato da cui si ottiene $L_j[h]$.

Le informazioni aggiuntive per le liste (2/2)

- Uno stato preso da L_{j-1} dà $\sigma_j[h] = 0$, mentre uno stato preso da M_j dà $\sigma_j[h] = 1$.
- In entrambi i casi $\rho_j[h]$ è la posizione dello stato di L_{j-1} che lo ha generato.
- L'algoritmo di PD deve anche conservare tutte le liste $L_0, L_1, \ldots, L_n$, e non solo l'ultima, perché la ricostruzione le ripercorre all'indietro.

Esempio: calcolo delle informazioni aggiuntive (1/2)

Consideriamo la prima istanza. Per ogni oggetto j e ogni posizione h, la tabella riporta lo stato $L_j[h]$ e i valori memorizzati $\sigma_j[h]$ e $\rho_j[h]$.

j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$
1	1	$(0,0)$	0	1
1	2	$(2,6)$	1	1
2	1	$(0,0)$	0	1
2	2	$(2,6)$	0	2
2	3	$(3,7)$	1	1
2	4	$(5,13)$	1	2

Esempio: calcolo delle informazioni aggiuntive (2/2)

j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$
3	1	$(0, 0)$	0	1
3	2	$(2, 6)$	0	2
3	3	$(3, 8)$	1	1
3	4	$(5, 14)$	1	2

Per esempio, lo stato $L_3[4] = (5, 14)$ ha $\sigma_3[4] = 1$ e $\rho_3[4] = 2$: si ottiene selezionando l'oggetto 3 a partire dallo stato $L_2[2] = (2, 6)$, poiché $2 + w_3 = 2 + 3 = 5$ e $6 + p_3 = 6 + 8 = 14$.

Ricostruzione con le liste

- Calcolati le liste e i valori memorizzati, si ricostruisce una soluzione ottima scorrendo gli oggetti da $j = n$ fino a $j = 1$, partendo dall'ultima posizione $h = |L_n|$, che contiene lo stato di profitto più grande:

$$x_j \leftarrow \sigma_j[h], \quad h \leftarrow \rho_j[h] \quad \text{per } j = n, n-1, \ldots, 1.$$

- Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, dove x_j è uguale a 1 se l'oggetto j è selezionato in una soluzione ottima, e a 0 altrimenti.

Algorithm: DP_Lists_Reconstruct

Input : Il numero n di oggetti, le liste $L_0, L_1, \ldots, L_n$ e i valori memorizzati σ e ρ

Output : Una soluzione ottima dell'istanza del KP

```
1 function DP_Lists_Reconstruct(n, L, σ, ρ)
2 |   x := new array[1..n]
3 |   for j := 1 to n do
4 |   |   x[j] ← 0
5 |   end
6 |   h := |L_n|
```

Algorithm: DP_Lists_Reconstruct (continua)

```
7     for j := n downto 1 do
8         x[j] ← σ_j[h]
9         h ← ρ_j[h]
10    end
11    return x
12 end
```

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

Dopo l'esecuzione dell'algoritmo di PD sulla prima istanza, l'ultima lista è

$$L_3 = \{(0,0),(2,6),(3,8),(5,14)\}$$

e il valore ottimo è il profitto del suo ultimo stato, cioè 14.

Esempio: calcolo di una soluzione ottima con le liste (2/2)

Possiamo ricostruire una soluzione ottima partendo da $j = 3$ e $h = |L_3| = 4$ come segue:

Iterazione	j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$	$x[1]$	$x[2]$	$x[3]$	Azione
–	–	–	–	–	–	0	0	0	$h = 4$
1	3	4	$(5, 14)$	1	2	0	0	1	$h \leftarrow 2$
2	2	2	$(2, 6)$	0	2	0	0	1	$h \leftarrow 2$
3	1	2	$(2, 6)$	1	1	1	0	1	$h \leftarrow 1$

Quindi il vettore

$$(1, 0, 1)$$

è una soluzione ottima. Il valore ottimo corrispondente è $6 + 8 = 14$.

Numero di passi di DP_Lists (1/2)

- Indichiamo con $S(n, c)$ il numero di passi eseguiti da `DP_Lists` nel caso peggiore, dove n è il numero di oggetti e c è la capacità dello zaino.
- Gli stati di una lista hanno pesi strettamente crescenti e ogni peso appartiene a $\{0, 1, \ldots, c\}$: pertanto $|L_j| \leq c + 1$ per ogni $j \in \{0, 1, \ldots, n\}$, e $|M_j| \leq |L_{j-1}|$.
- Per ogni oggetto, `DP_Lists` scorre una volta L_{j-1} per costruire M_j e poi chiama `Merge`, che scandisce $|L_{j-1}| + |M_j| \leq 2(c + 1)$ stati ed esegue un numero costante di operazioni per ciascuno di essi.
- Pertanto il numero di passi eseguiti per ogni oggetto è $\Theta(c)$ nel caso peggiore, e

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

Numero di passi di DP_Lists (2/2)

- Le tre implementazioni hanno lo stesso ordine di crescita, ma non memorizzano la stessa quantità di informazione.
- La matrice v ha sempre $(n+1)(c+1)$ celle, mentre le liste contengono solo gli stati effettivamente raggiungibili: una lista raggiunge il limite $c+1$ solo quando ogni valore di capacità è il peso totale di un sottoinsieme degli oggetti.
- Sulla prima istanza le liste contengono 1, 2, 4 e 4 stati, contro le 24 celle della matrice v.
- La lista L_j soddisfa anche $|L_j| \leq 2^j$, poiché gli oggetti $\{1, 2, \ldots, j\}$ hanno 2^j sottoinsiemi. Pertanto

$$|L_j| \leq \min\left\{2^j,\ c+1\right\},$$

e il numero di stati resta piccolo finché j è piccolo, qualunque sia la capacità dello zaino.

Riepilogo (1/2)

La ricorrenza $v(j, k)$:

- $v(j, k)$ è il valore ottimo con gli oggetti $\{1, 2, \ldots, j\}$ e la capacità k
- Sceglie fra lasciare fuori l'oggetto j e prenderlo; ha la sottostruttura ottima e i sottoproblemi sovrapposti

Il calcolo del valore ottimo:

- Top-down, gli stessi sottoproblemi vengono risolti più di una volta
- Bottom-up, la matrice v si riempie riga per riga in $\Theta(nc)$ passi
- Un solo vettore, scorso con capacità decrescenti, dà il valore ottimo con $c + 1$ celle

Riepilogo (2/2)

Una soluzione ottima:

- La matrice s memorizza le decisioni; si legge all'indietro a partire da $j = n$ e $k = c$

Liste di stati:

- Uno stato (w, p) è il peso totale e il profitto totale di un sottoinsieme; gli stati dominati si scartano
- La lista L_j è la riga j di v in forma compressa, calcolata con `Merge`
- Il numero di passi resta $\Theta(nc)$, ma $|L_j| \leq \min\{2^j, c+1\}$

Riepilogo della prima parte

- Una soluzione ammissibile del KP è un vettore binario; i 2^n sottoinsiemi non si possono enumerare quando n cresce.
- La ricorrenza $v(j, k)$ ha la sottostruttura ottima e i sottoproblemi sovrapposti; la matrice v si riempie bottom-up in $\Theta(nc)$ passi, e un solo vettore con $c + 1$ celle dà il valore ottimo.
- La matrice s delle decisioni dà una soluzione ottima, letta all'indietro a partire da (n, c).
- Le liste di stati non dominati memorizzano solo gli stati raggiungibili.
- Nella seconda parte: un secondo algoritmo di PD, che scambia i ruoli di profitti e pesi.