

SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione dinamica

Volume 1 — Programmazione dinamica

I principi della programmazione dinamica e gli algoritmi di PD per i problemi di zaino, il lot sizing, il cambio monete e il taglio delle aste: ricorrenze, pseudocodice, complessità e tabelle svolte.

Fabio Furini

fabiofurini.github.io

Dipartimento di Ingegneria informatica, automatica e gestionale — Sapienza Università di Roma

Versione online e materiale scaricabile:
fabiofurini.github.io/programmazione-dinamica/
github.com/fabiofurini/programmazione-dinamica

Anno accademico 2026–2027

Indice

Principi **4**

1 Principi della programmazione dinamica **5**
1.1 Dividere, conquistare, combinare 6
1.2 Proprietà del problema richieste dagli algoritmi di PD 7
1.3 Passi per progettare un algoritmo di PD 8
1.4 Gli algoritmi di PD di questo modulo 8
1.5 Errori tipici 9
1.6 In sintesi 9

Problemi di zaino **10**

2 PD per il problema dello zaino 0/1 **11**
2.1 Istanze e soluzioni ammissibili 12
2.2 Un primo algoritmo di PD 14
2.3 Un secondo algoritmo di PD 36
2.4 Errori tipici 62
2.5 In sintesi 63

3 PD per il problema dello zaino illimitato **64**
3.1 Istanze e soluzioni ammissibili 65
3.2 Un primo algoritmo di PD 66
3.3 Un secondo algoritmo di PD 77
3.4 Errori tipici 88
3.5 In sintesi 89

4 PD per il problema dello zaino limitato **90**
4.1 Istanze e soluzioni ammissibili 91
4.2 Un algoritmo di PD 92
4.3 Errori tipici 103
4.4 In sintesi 104

5 PD per il problema del subset sum **105**
5.1 Un algoritmo di PD . . . 106
5.2 Errori tipici . . . 112
5.3 In sintesi . . . 112

6 PD per il problema dello zaino a scelta multipla **113**
6.1 Un algoritmo di PD . . . 114
6.2 Errori tipici . . . 120
6.3 In sintesi . . . 120

7 PD per il problema dello zaino multidimensionale **121**
7.1 Un algoritmo di PD . . . 122
7.2 Errori tipici . . . 129
7.3 In sintesi . . . 129

8 PD per il problema dello zaino con setup **130**
8.1 Un algoritmo di PD . . . 131
8.2 Errori tipici . . . 138
8.3 In sintesi . . . 138

9 PD per il problema degli zaini multipli **139**
9.1 Un algoritmo di PD . . . 140
9.2 Errori tipici . . . 147
9.3 In sintesi . . . 147

Problemi di lot sizing **148**

10 PD per il problema del lot sizing senza capacità **149**
10.1 Notazione, modello e soluzioni ammissibili . . . 150
10.2 Un algoritmo di PD per l'ULS . . . 155
10.3 Errori tipici . . . 168
10.4 In sintesi . . . 168

11 PD per il problema del lot sizing capacitato **170**
11.1 Notazione, modello e soluzioni ammissibili . . . 171
11.2 Un algoritmo di PD per il CLS . . . 172
11.3 Errori tipici . . . 184
11.4 In sintesi . . . 185

Problemi misti **186**

12 PD per il problema del cambio monete **187**
12.1 Istanze e soluzioni ammissibili . . . 188

12.2 Un algoritmo di PD 189
12.3 Errori tipici 199
12.4 In sintesi 199

13 PD per il problema del taglio delle aste 200
13.1 Istanze e soluzioni ammissibili 201
13.2 Un primo algoritmo di PD 203
13.3 Un secondo algoritmo di PD 212
13.4 Errori tipici 224
13.5 In sintesi 224

Principi

Capitolo 1

Principi della programmazione dinamica

Prerequisiti

- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Ottimizzazione combinatoria: definizioni»: i problemi di ottimizzazione combinatoria, le loro soluzioni ammissibili e il fatto che il loro numero cresce in modo esponenziale.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema dello zaino 0/1 (KP), l'esempio guida degli algoritmi di PD.
- Dal modulo *Algoritmi*: pseudocodice (Capitolo 2), ricorsione (Capitolo 14) e la crescita del tempo di esecuzione di un algoritmo (Capitolo 10 «Crescita asintotica»).

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- spiegare perché l'enumerazione di tutte le soluzioni ammissibili non funziona, e che cosa fa invece un algoritmo di PD;
- enunciare i principi di divisione, conquista e combinazione, e la differenza fra sottoproblemi indipendenti e sottoproblemi sovrapposti;
- verificare su un problema la sottostruttura ottima e i sottoproblemi sovrapposti;
- elencare i passi di un algoritmo di PD: ricorrenza, casi base, tabella di memorizzazione e ricostruzione di una soluzione ottima.

1.1 Dividere, conquistare, combinare

Definizione 1.1: Algoritmo di PD

Un ***algoritmo di programmazione dinamica (PD)*** è un algoritmo esatto per trovare una soluzione ottima di una classe di problemi di ottimizzazione con proprietà specifiche (Sezione 1.2). Decompone il problema originale in sottoproblemi più piccoli dello stesso tipo e riusa l'informazione calcolata risolvendoli.

- Un algoritmo a forza bruta elenca tutte le soluzioni ammissibili, e il loro numero cresce in modo esponenziale (2^n sottoinsiemi di oggetti per il KP). Un algoritmo di PD evita questa enumerazione riusando i valori ottimi dei sottoproblemi.
- Un algoritmo di PD sfrutta il ***paradigma divide et impera*** (dividere, conquistare e combinare), i cui principi sono:
  1. **Principio di divisione** $\rightsquigarrow$ decomporre il problema in un certo numero di sottoproblemi, che sono problemi più piccoli dello stesso tipo;
  2. **Principio di conquista** $\rightsquigarrow$ risolvere i sottoproblemi in modo ricorsivo. Se le dimensioni dei sottoproblemi sono abbastanza piccole, risolverli direttamente;
  3. **Principio di combinazione** $\rightsquigarrow$ combinare soluzioni ottime dei sottoproblemi per ottenere una soluzione ottima del problema originale.
- Gli algoritmi di PD dividono il problema in ***sottoproblemi indipendenti***, li risolvono ricorsivamente e poi ne combinano le soluzioni per risolvere il problema originale. Qui indipendenti significa che ogni sottoproblema generato dalla divisione si può risolvere senza influire sugli altri.
- In un algoritmo di PD lo stesso sottoproblema può essere generato più volte durante il processo ricorsivo di soluzione. Per evitare calcoli ridondanti, il valore ottimo di ogni sottoproblema si memorizza in una ***tabella di memorizzazione***.
- Ogni volta che lo stesso sottoproblema viene generato di nuovo, il suo valore ottimo si legge dalla tabella di memorizzazione invece di essere ricalcolato.

1.2 Proprietà del problema richieste dagli algoritmi di PD

- Gli algoritmi di PD funzionano solo per problemi di ottimizzazione combinatoria che ammettono un processo di decomposizione ricorsiva con le seguenti proprietà:

  1. ***Sottostruttura ottima*** $\rightsquigarrow$ l'idea è che, quando un problema è decomposto in sottoproblemi più piccoli, una soluzione ottima del problema si può costruire a partire da soluzioni ottime dei sottoproblemi.

     In pratica, per verificare questa proprietà si controllano di solito i seguenti punti:

     - il problema si può decomporre in sottoproblemi più piccoli dello stesso tipo;
     - i sottoproblemi si possono risolvere in modo indipendente;
     - le soluzioni ottime dei sottoproblemi si possono combinare per ottenere una soluzione ottima del problema originale;
     - si può stabilire una **relazione di ricorrenza** per calcolare il valore ottimo del problema.

  2. ***Sottoproblemi sovrapposti*** $\rightsquigarrow$ l'idea è che, quando un problema è decomposto ricorsivamente in sottoproblemi più piccoli, gli stessi sottoproblemi possono essere generati più volte.

     In pratica, per verificare questa proprietà si controllano di solito i seguenti punti:

     - il processo di decomposizione ricorsiva genera lo stesso sottoproblema più di una volta;
     - è possibile memorizzare i valori ottimi dei sottoproblemi in una tabella di memorizzazione; allora, ogni volta che lo stesso sottoproblema serve di nuovo, il suo valore ottimo si legge dalla tabella invece di essere ricalcolato.

Gli algoritmi di PD si possono usare quando un problema ha sia la sottostruttura ottima sia i sottoproblemi sovrapposti.

1.3 Passi per progettare un algoritmo di PD

- Identificare il ***processo di decomposizione ricorsiva*** che, a partire da un problema dato, genera i suoi sottoproblemi, che sono problemi più piccoli dello stesso tipo, e verificare le due proprietà seguenti:
  1. la proprietà di sottostruttura ottima, controllando che una soluzione ottima del problema si possa ottenere combinando le soluzioni ottime dei suoi sottoproblemi;
  2. la proprietà dei sottoproblemi sovrapposti, controllando se il processo di decomposizione ricorsiva genera più volte gli stessi sottoproblemi;
- La costruzione di un algoritmo di PD segue di solito due passi principali:
  1. calcolo del valore ottimo
     - definire una ***relazione di ricorrenza*** che esprime il valore ottimo di un problema in funzione dei valori ottimi dei suoi sottoproblemi;
     - specificare i ***casi base***, ossia i sottoproblemi la cui soluzione ottima è immediata (il valore ottimo ne segue direttamente);
     - determinare un algoritmo di PD che calcoli i valori ottimi di tutti i sottoproblemi rilevanti, di solito ***dal basso verso l'alto*** (bottom-up), memorizzando ogni valore in una tabella di memorizzazione, in modo che sia calcolato una sola volta e letto ogni volta che serve di nuovo;
  2. costruzione di una soluzione ottima
     - specificare l'informazione aggiuntiva da memorizzare per ricostruire una soluzione ottima del problema.
     - identificare il modo di costruire una soluzione ottima dall'informazione memorizzata.

1.4 Gli algoritmi di PD di questo modulo

- Ogni algoritmo di PD di questo modulo segue i due passi precedenti: una ricorrenza con i suoi casi base dà il valore ottimo, una tabella aggiuntiva dà una soluzione ottima.

Problema	Stato	Ricorrenza su	Passi
KP	$v(j,k)$	oggetti e capacità	$\Theta(nc)$
UKP	$v(k)$	capacità	$\Theta(nc)$
BKP	$v(j,k)$	oggetti e capacità	$\Theta(nc\,b_{\max})$
SSP	$r(j,k)$	oggetti e valori da raggiungere	$\Theta(nc)$
MCKP	$v(h,k)$	classi e capacità	$\Theta(nc)$
MDKP	$v(j,k_1,\ldots,k_m)$	oggetti e vettori di capacità	$\Theta(m\,n\prod_d(c_d+1))$
KPS	$v(h,k)$, $u(h,i,k)$	classi, oggetti e capacità	$\Theta(nc)$
MKP	$v(j,k_1,\ldots,k_m)$	oggetti e capacità residue	$\Theta(m\,n\prod_i(c_i+1))$
RCP	$v(\ell)$	lunghezze dell'asta	$\Theta(n^2)$
CCP	$v(k)$	importi del resto	$\Theta(nc)$
ULS	$v(t)$	periodi	$\Theta(m^2)$
CLS	$v(t,k)$	periodi e scorte	$O(mDC)$

- In tutti gli esempi numerici i dati sono interi: il numero di stati è finito, e ogni stato è una cella della tabella. I tempi di esecuzione $\Theta(nc)$, $\Theta(nc\,b_{\max})$ e $O(mDC)$ contengono numeri dell'input (una capacità, un bound, una domanda), non solo la sua dimensione: sono **pseudo-polinomiali**.
- Con un vettore di capacità (MDKP, MKP) la tabella ha una dimensione per ogni capacità: il numero di passi è pseudo-polinomiale solo per un numero fissato m di dimensioni o di zaini.

1.5 Errori tipici

Attenzione

- **Usare la PD senza la sottostruttura ottima.** Se una soluzione ottima del problema non si può costruire da soluzioni ottime dei sottoproblemi, la ricorrenza è sbagliata.
- **Usare la PD senza sottoproblemi sovrapposti.** Se ogni sottoproblema compare una sola volta, memorizzare i valori in una tabella non porta alcun vantaggio: l'algoritmo di PD è una semplice ricorsione.
- **Dimenticare i casi base.** La ricorrenza ha bisogno dei sottoproblemi il cui valore ottimo è immediato, altrimenti la ricorsione non termina.
- **Confondere il valore ottimo con una soluzione ottima.** La tabella dei valori ottimi non dà la soluzione: serve una tabella aggiuntiva delle decisioni.

1.6 In sintesi

In sintesi

- Un algoritmo di PD è un algoritmo esatto che divide un problema in sottoproblemi dello stesso tipo, li risolve ricorsivamente e combina le soluzioni ottime.
- Funziona quando il problema ha la sottostruttura ottima e sottoproblemi sovrapposti; i valori ottimi si memorizzano in una tabella di memorizzazione.
- La progettazione ha due passi: il valore ottimo (ricorrenza, casi base, tabella bottom-up) e una soluzione ottima (decisioni memorizzate e lette a ritroso).
- La forza bruta richiede tempo esponenziale; la PD richiede un tempo polinomiale nel numero di stati, e pseudo-polinomiale quando un numero dell'input è una dimensione della tabella.

Problemi di zaino

Capitolo 2

PD per il problema dello zaino 0/1

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 (Capitolo 2) e la dimensione dell'input di un algoritmo (Capitolo 11) (serve a spiegare perché $\Theta(nc)$ è pseudopolinomiale).

Dove eravamo

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

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 $u(j, q)$.

Il problema di questo capitolo è il problema dello zaino 0/1 (KP): la sua definizione e le sue ipotesi sono nella sua pagina.

2.1 Istanze e soluzioni ammissibili

Esempio 2.1: 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 Esempio 2.1.

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.

2.1.1 Codifica delle soluzioni ammissibili

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:

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

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

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

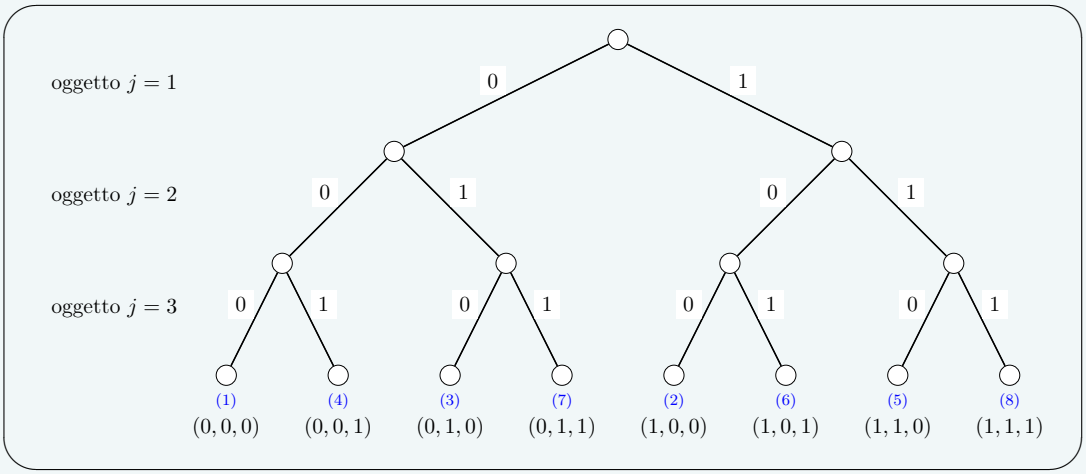
Esempio 2.2: Enumerazione di tutte le possibili soluzioni del KP

Consideriamo l'istanza del KP dell'Esempio 2.1, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

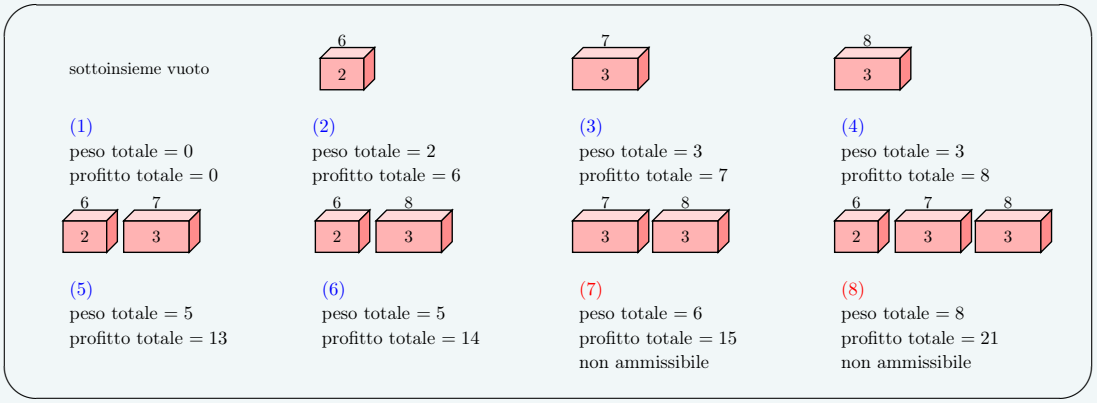
	$j=1$	$j=2$	$j=3$
p_j	6	7	8
w_j	2	3	3

Ci sono $2^3 = 8$ possibili sottoinsiemi di oggetti, corrispondenti ai diversi modi di selezionare o non selezionare gli oggetti.

Nel seguente albero binario 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.



La figura seguente mostra 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.



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

2.2 Un primo algoritmo di PD

- Passo 1: calcolo del valore ottimo

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

$$v(j,k) = \begin{cases} 0 & \text{se } j=0 \text{ o } k=0, \\ v(j-1,k) & \text{se } 1 \le j \le n,\ 1 \le k \le c \text{ e } w_j > k, \\ \max\left\{ v(j-1,k),\ p_j + v(j-1,k-w_j) \right\} & \text{se } 1 \le j \le n,\ 1 \le k \le c,\ w_j \le k. \end{cases} \tag{2.1}$$

- Il processo di decomposizione ricorsiva della formula (2.1) 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.

- 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!**

- 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 (2.1).

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

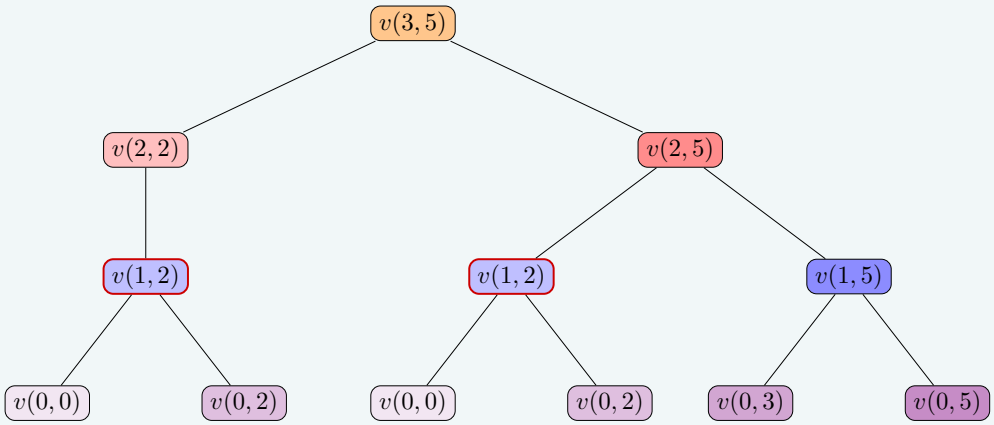
- Un **primo metodo** per calcolare il valore ottimo $v(n, c)$ dell'istanza del KP è l'**approccio top-down**.

- Il metodo parte dalla chiamata con $j = n$ e $k = c$ e valuta ricorsivamente la formula (2.1): 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 2.3: Albero di ricorsione

Consideriamo l'istanza del KP dell'Esempio 2.1. 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 $v(0,0)$.

– Il valore $v(j,k)$ si può calcolare quando sono noti i valori richiesti dal membro destro della formula (2.1). 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 2.4: Calcolo dei valori della funzione v

Consideriamo l'istanza del KP dell'Esempio 2.1, 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 $v(1,2)$:

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

* 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 $v(3,5)$:

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

– 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, partendo 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.

– 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 (2.1) 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**:

Algoritmo

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
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]
```

* 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.
* 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 2.5: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del KP dell'Esempio 2.1, 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	$v[0][5] \leftarrow 0$
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	$\max\{v[1][5], p[2] + v[1][2]\} = \max\{6, 7 + 6\} = 13$
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	$\max\{v[2][5], p[3] + v[2][2]\} = \max\{13, 8 + 6\} = 14$

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

- 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.

- 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 $v(j-1,k-w_j)$.

- 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**:

Algoritmo

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
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]
```

* 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.
* 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 2.6: Esecuzione dell'algoritmo di PD con un solo vettore

Consideriamo l'istanza del KP dell'Esempio 2.1, 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	$p[1]+v[0]=6+0>v[2]=0$: $v[2]\leftarrow 6$
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	$p[3]+v[0]=8+0>v[3]=7$: $v[3]\leftarrow 8$

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 riportata nell'Esempio 2.5. Il valore ottimo dell'istanza del KP è $v[5]=14$.

  - 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
  - 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:

Algoritmo

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

∗ 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 2.7: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza del KP dell'Esempio 2.1, 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	$s[0][5] \leftarrow 0$
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	$p[2] + v[1][2] = 7 + 6 > v[1][5] = 6$: $s[2][5] \leftarrow 1$
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	$p[3] + v[2][2] = 8 + 6 > v[2][5] = 13$: $s[3][5] \leftarrow 1$

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

– 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 $j \in$

$\{1, 2, \ldots, n\}$:

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

– 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.

Algoritmo

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
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
```

Esempio 2.8: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 2.1, 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

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 $6 + 8 = 14$.

2.2.1 Numero di passi

- 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. 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.

  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

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

  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 $c+1$.

2.2.2 Un'implementazione con liste

- Passo 1: calcolo del valore ottimo
  - 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.
  - Questa sottosezione descrive una **diversa implementazione dello stesso algoritmo di PD**. La decomposizione ricorsiva e la sottostruttura ottima sono quelle della formula (2.1): 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.

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.

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.

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 formula ricorsiva per calcolare L_j è la seguente:

$$L_j = \begin{cases} \{(0,0)\} & \text{se } j = 0, \\ \mathrm{nd}\big(L_{j-1} \cup M_j\big) & \text{se } 1 \le j \le n, \end{cases} \tag{2.2}$$

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

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

– Il processo di decomposizione ricorsiva della formula (2.2) 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 $w + w_j \le c$.

– Qui $L_0 = \{(0,0)\}$ perché l'unico sottoinsieme di un insieme vuoto di oggetti è il sottoinsieme vuoto.

– 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 $\{1, 2, \ldots, j-1\}$.

  $\rightsquigarrow$ **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.

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

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

Il valore ottimo dell'istanza del KP è

$$\max \{\, p \;:\; (w,p) \in L_n \,\}. \tag{2.3}$$

– 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 (2.3) è il profitto dell'ultimo stato di L_n.

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

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

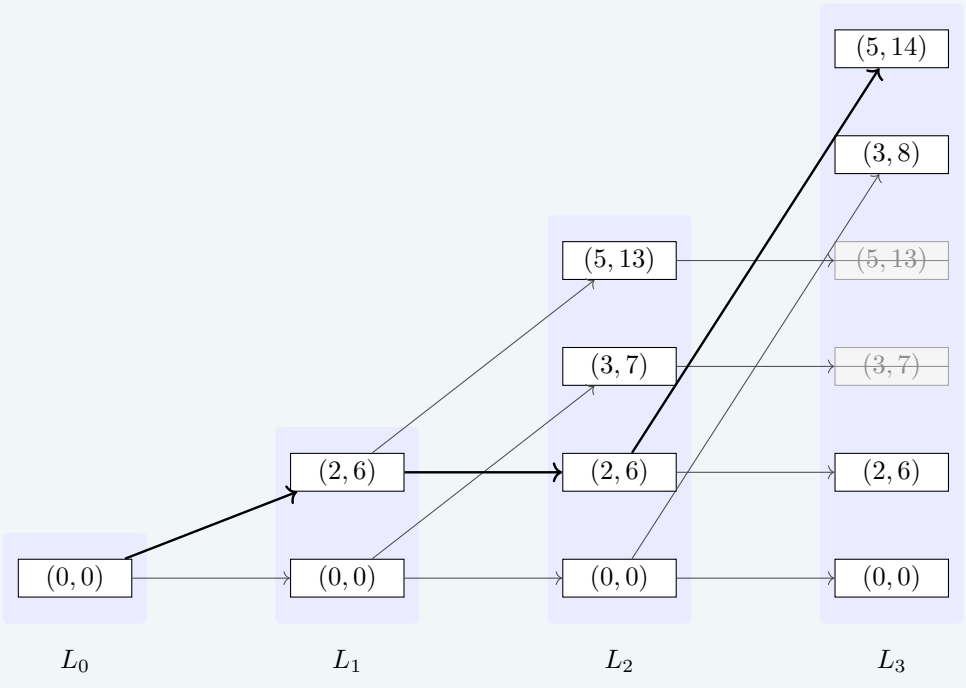
Ogni stato (w,p) di L_j indica, con il suo peso, un valore di capacità in cui la riga aumenta, e p è il nuovo valore.

Esempio 2.9: Le liste di stati

Consideriamo l'istanza del KP dell'Esempio 2.1, 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

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.



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.

– 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 (2.2) si calcola scorrendo contemporaneamente le due liste, come nella fusione di due liste ordinate.

– 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**:

Algoritmo

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)
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
```

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

Algoritmo

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
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
```

* 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.
* 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, $L[|L|].p$.

Esempio 2.10: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del KP dell'Esempio 2.1, 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	$\{(3,8),(5,14)\}$	$\{(0,0),(2,6),(3,8),(5,14)\}$

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
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.

  - 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.

- Passo 2: calcolo di una soluzione ottima

  - 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 $L_j[h]$.
- 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 2.11: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza del KP dell'Esempio 2.1. 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
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 $6 + p_3 = 6 + 8 = 14$.

- 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.

Algoritmo

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

Esempio 2.12: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 2.1, 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.

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 $6 + 8 = 14$.

2.2.3 Numero di passi

- 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
$$S(n, c) = \Theta(nc).$$
- 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.
- Sull'istanza del KP dell'Esempio 2.1 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.

2.3 Un secondo algoritmo di PD

- Passo 1: calcolo del valore ottimo
  - Nell'algoritmo di PD della sezione precedente, un sottoproblema è identificato da un valore di capacità, e il suo valore ottimo è il massimo profitto totale ottenibile senza superare quella capacità.
  - Nell'algoritmo di PD di questa sezione i ruoli dei profitti e dei pesi sono scambiati: un sottoproblema è identificato da un valore di profitto, e il suo valore ottimo è il minimo peso totale necessario per ottenere esattamente quel profitto.
  - Poiché ogni profitto p_j è un intero positivo e ogni oggetto si può selezionare al più una volta, il profitto totale di una soluzione ammissibile è un intero compreso fra 0 e il profitto totale di tutti gli oggetti
$$P = \sum_{j=1}^{n} p_j.$$

– Pertanto basta considerare i valori di profitto $q \in \{0, 1, \ldots, P\}$.

– Il numero di sottoproblemi di questo algoritmo di PD cresce con il profitto totale P. Per tenere piccolo il numero di sottoproblemi, illustriamo l'algoritmo di PD sulla seguente istanza del KP, che ha gli stessi pesi degli oggetti e la stessa capacità dello zaino dell'istanza dell'Esempio 2.1, e profitti più piccoli.

Esempio 2.13: 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	3	3	4
w_j	2	3	3

Chiamiamo questa istanza del KP Esempio 2.13.

Per ogni $j \in \{0, 1, \ldots, n\}$ e $q \in \{0, 1, \ldots, P\}$, sia $u(j, q)$ il valore ottimo della sottoistanza di peso minimo data da:

1. un profitto obiettivo q,
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,

che chiede di selezionare un sottoinsieme degli oggetti disponibili il cui profitto totale sia uguale a q, in modo da minimizzare il peso totale degli oggetti selezionati.

La formula ricorsiva per calcolare $u(j, q)$ è la seguente:

$$u(j,q) = \begin{cases} 0 & \text{se } q = 0, \\ +\infty & \text{se } j = 0 \text{ e } 1 \le q \le P, \\ u(j-1,q) & \text{se } 1 \le j \le n,\ 1 \le q \le P \text{ e } p_j > q, \\ \min\left\{u(j-1,q),\ w_j + u(j-1,q-p_j)\right\} & \text{se } 1 \le j \le n,\ 1 \le q \le P,\ p_j \le q. \end{cases} \tag{2.4}$$

– Il processo di decomposizione ricorsiva della formula (2.4) genera al più 2 **sottoproblemi indipendenti**, corrispondenti alla scelta di selezionare o no l'oggetto j per

raggiungere il profitto obiettivo q. Se $p_j > q$, selezionare l'oggetto j supererebbe il profitto obiettivo e quindi l'unica opzione ammissibile è non selezionarlo.

- Qui $u(j, 0) = 0$ perché il sottoinsieme vuoto ha profitto totale 0 e peso totale 0. Inoltre, $u(0, q) = +\infty$ per $q \geq 1$: in questo caso nessun oggetto è disponibile, quindi nessun sottoinsieme ha profitto totale uguale a q. Il valore $+\infty$ serve a escludere questo caso inammissibile.

- Se $p_j \leq q$, il valore ottimo $u(j, q)$ si ottiene provando le due opzioni possibili, selezionare o non selezionare l'oggetto j, e scegliendo quella che minimizza il peso totale. I valori corrispondenti sono, rispettivamente, $w_j + u(j-1, q-p_j)$ e $u(j-1, q)$. Se $p_j > q$, l'oggetto j non può essere selezionato e quindi $u(j, q) = u(j-1, q)$.

- Quindi un sottoinsieme di peso minimo con profitto totale q e oggetti disponibili $\{1, 2, \ldots, j\}$ si costruisce combinando la decisione scelta sull'oggetto j con un sottoinsieme di peso minimo del corrispondente sottoproblema residuo.

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

- Sequenze diverse di decisioni sugli oggetti, prese eventualmente per valori diversi di j e q, possono portare allo stesso sottoproblema; quindi lo stesso valore $u(j, q)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (2.4).

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

- I valori $u(n, q)$ non danno direttamente il valore ottimo dell'istanza del KP: danno, per ogni profitto obiettivo q, il minimo peso totale necessario per ottenerlo. Un sottoinsieme con profitto totale q entra nello zaino se e solo se il suo minimo peso totale non supera la capacità, cioè se e solo se $u(n, q) \leq c$.

  Il valore ottimo dell'istanza del KP è

  $$\max \Big\{ q \in \{0, 1, \ldots, P\} \ : \ u(n, q) \leq c \Big\}. \tag{2.5}$$

  * L'insieme della formula (2.5) non è mai vuoto, poiché $u(n, 0) = 0 \leq c$, e quindi il massimo è sempre ben definito.

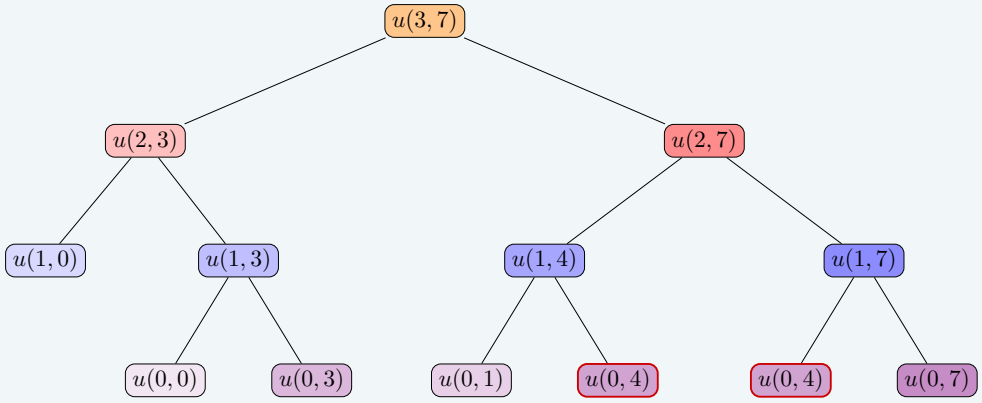
- Un **primo metodo** per calcolare il valore ottimo dell'istanza del KP è l'**approccio top-down**.

- Il metodo valuta ricorsivamente la formula (2.4), partendo dalla chiamata con $j = n$, per ogni profitto obiettivo $q \in \{0, 1, \ldots, P\}$: ogni chiamata considera l'oggetto j e non lo seleziona oppure, se il suo profitto non supera il profitto obiettivo, considera anche di selezionarlo. La ricorsione si ferma quando non restano oggetti o quando il profitto obiettivo è uguale a 0. Infine, si applica la formula (2.5) ai valori calcolati.

– L'**albero di ricorsione** della funzione u 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 2.14: Albero di ricorsione

Consideriamo l'istanza del KP dell'Esempio 2.13. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $u(3, 7)$:



La funzione viene chiamata 13 volte e lo stesso sottoproblema viene risolto più di una volta: $u(0, 4)$ viene chiamata 2 volte (con il bordo rosso). Inoltre, l'approccio top-down va applicato a ogni profitto obiettivo $q \in \{0, 1, \ldots, 10\}$, e lo stesso sottoproblema viene generato da chiamate iniziali diverse: per esempio, la chiamata iniziale $u(3, 3)$ genera le chiamate $u(2, 3)$, $u(1, 3)$, $u(1, 0)$, $u(0, 3)$ e $u(0, 0)$, già generate dalla chiamata iniziale $u(3, 7)$ riportata sopra.

– Il valore $u(j, q)$ si può calcolare quando sono noti i valori richiesti dal membro destro della formula (2.4). In particolare, $u(j - 1, q)$ serve quando l'oggetto j non è selezionato e, se $p_j \leq q$, $u(j - 1, q - p_j)$ serve quando l'oggetto j è selezionato.

– Pertanto, per un valore fissato di j, tutti i valori $u(j, q)$ si possono calcolare considerando i valori di profitto q in ordine crescente.

Esempio 2.15: Calcolo dei valori della funzione u

Consideriamo l'istanza del KP dell'Esempio 2.13, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	3	3	4
w_j	2	3	3

Il profitto totale di tutti gli oggetti è $P = 3 + 3 + 4 = 10$. I valori della funzione u si calcolano come segue:

* I casi base danno $u(0,0) = 0$ e $u(0,q) = +\infty$ per ogni $q \geq 1$.
* Il primo caso della formula (2.4) ($q = 0$) dà il caso base:

$$u(1,0) = 0$$

* Calcolati i casi base, possiamo calcolare $u(1,3)$:

$$u(1,3) = \min\{u(0,3),\ w_1 + u(0,0)\} = \min\{+\infty,\ 2+0\} = 2$$

* Possiamo calcolare anche $u(1,4)$ e $u(1,7)$:

$$u(1,4) = \min\{u(0,4),\ w_1 + u(0,1)\} = \min\{+\infty,\ +\infty\} = +\infty$$

$$u(1,7) = \min\{u(0,7),\ w_1 + u(0,4)\} = \min\{+\infty,\ +\infty\} = +\infty$$

* Calcolati $u(1,3)$ e $u(1,0)$, possiamo calcolare $u(2,3)$:

$$u(2,3) = \min\{u(1,3),\ w_2 + u(1,0)\} = \min\{2,\ 3+0\} = 2$$

* Calcolati $u(1,7)$ e $u(1,4)$, possiamo calcolare $u(2,7)$:

$$u(2,7) = \min\{u(1,7),\ w_2 + u(1,4)\} = \min\{+\infty,\ +\infty\} = +\infty$$

* Calcolati $u(2,7)$ e $u(2,3)$, possiamo calcolare $u(3,7)$:

$$u(3,7) = \min\{u(2,7),\ w_3 + u(2,3)\} = \min\{+\infty,\ 3+2\} = 5$$

* Pertanto un profitto totale di 7 si può ottenere con un peso totale di 5, che non supera la capacità $c = 5$ dello zaino.

– 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 profitto obiettivo, partendo dai casi base $u(0,0) = 0$, $u(0,q) = +\infty$ per ogni $q \in \{1,2,\ldots,P\}$ e $u(j,0) = 0$, fino ad arrivare ai valori $u(n,q)$ per ogni $q \in \{0,1,\ldots,P\}$.

– Questa osservazione porta in modo naturale a un **metodo molto più efficiente** per calcolare il valore ottimo dell'istanza del KP.

– Un **secondo metodo** per calcolare il valore ottimo dell'istanza del KP è l'**approccio bottom-up**.

– Il metodo parte dai casi base e valuta la formula (2.4) per valori crescenti di $j \in \{1, 2, \ldots, n\}$ e $q \in \{0, 1, \ldots, P\}$: ogni valore $u(j, q)$ si calcola con valori già calcolati, fino a ottenere tutti i valori $u(n, q)$. Il valore ottimo dell'istanza del KP si ottiene poi applicando la formula (2.5).

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

Algoritmo

Algorithm: DP2

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 DP2(n, p, w, c)
2      P := 0
3      for j := 1 to n do
4          P ← P + p[j]
5      end
6      u := new array[0..n][0..P]
7      u[0][0] ← 0
8      for q := 1 to P do
9          u[0][q] ← +∞
10     end
11     for j := 1 to n do
12         for q := 0 to P do
13             if q < p[j] then
14                 u[j][q] ← u[j − 1][q]
15             else
16                 if w[j] + u[j − 1][q − p[j]] < u[j − 1][q] then
17                     u[j][q] ← w[j] + u[j − 1][q − p[j]]
18                 else
19                     u[j][q] ← u[j − 1][q]
20                 end
21             end
22         end
23     end
24     z := 0
25     for q := 0 to P do
26         if u[n][q] ≤ c then
27             z ← q
28         end
29     end
30     return z
```

* Il primo ciclo calcola il profitto totale P di tutti gli oggetti, che è il più grande profitto obiettivo da considerare.
* La matrice $u[0..n][0..P]$ è la **lookup table**: memorizza i valori ottimi di tutti i sottoproblemi.
* Il secondo ciclo inizializza i valori $u[0][0] = 0$ e $u[0][q] = +\infty$ per tutti i profitti obiettivo $q \in \{1, 2, \ldots, P\}$, 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 $q \in \{0, 1, \ldots, P\}$, calcola il valore ottimo $u[j][q]$ per ogni profitto obiettivo q.
* Se il profitto dell'oggetto j supera il profitto obiettivo, allora $u[j][q] \leftarrow u[j-1][q]$.
* Altrimenti, l'algoritmo di PD confronta il valore ottenuto non selezionando l'oggetto j, cioè $u[j-1][q]$, con il valore ottenuto selezionando l'oggetto j, cioè $w[j] + u[j-1][q-p[j]]$.
* La variabile ausiliaria z memorizza il più grande profitto obiettivo il cui minimo peso totale non supera la capacità dello zaino. Si inizializza a 0, che è sempre un profitto obiettivo ammissibile poiché $u[n][0] = 0 \leq c$.
* L'ultimo ciclo scorre i profitti obiettivo in ordine crescente e aggiorna z ogni volta che $u[n][q] \leq c$, in modo che alla fine z sia il massimo della formula (2.5).
* Infine, l'algoritmo restituisce il valore ottimo dell'istanza del KP, z.

Esempio 2.16: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del KP dell'Esempio 2.13, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$
p_j	3	3	4
w_j	2	3	3

Il profitto totale di tutti gli oggetti è $P = 3+3+4 = 10$. Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di j e q, la cella $u[j][q]$ e l'operazione eseguita.

j	q	$u[j][q]$	Azione
0	0	0	$u[0][0] \leftarrow 0$
0	1	$+\infty$	$u[0][1] \leftarrow +\infty$
0	2	$+\infty$	$u[0][2] \leftarrow +\infty$
0	3	$+\infty$	$u[0][3] \leftarrow +\infty$

0	4	$+\infty$	$u[0][4] \leftarrow +\infty$
0	5	$+\infty$	$u[0][5] \leftarrow +\infty$
0	6	$+\infty$	$u[0][6] \leftarrow +\infty$
0	7	$+\infty$	$u[0][7] \leftarrow +\infty$
0	8	$+\infty$	$u[0][8] \leftarrow +\infty$
0	9	$+\infty$	$u[0][9] \leftarrow +\infty$
0	10	$+\infty$	$u[0][10] \leftarrow +\infty$
1	0	0	$u[1][0] \leftarrow u[0][0] = 0$
1	1	$+\infty$	$u[1][1] \leftarrow u[0][1] = +\infty$
1	2	$+\infty$	$u[1][2] \leftarrow u[0][2] = +\infty$
1	3	2	$\min\{u[0][3], w[1] + u[0][0]\} = \min\{+\infty, 2 + 0\} = 2$
1	4	$+\infty$	$\min\{u[0][4], w[1] + u[0][1]\} = \min\{+\infty, +\infty\} = +\infty$
1	5	$+\infty$	$\min\{u[0][5], w[1] + u[0][2]\} = \min\{+\infty, +\infty\} = +\infty$
1	6	$+\infty$	$\min\{u[0][6], w[1] + u[0][3]\} = \min\{+\infty, +\infty\} = +\infty$
1	7	$+\infty$	$\min\{u[0][7], w[1] + u[0][4]\} = \min\{+\infty, +\infty\} = +\infty$
1	8	$+\infty$	$\min\{u[0][8], w[1] + u[0][5]\} = \min\{+\infty, +\infty\} = +\infty$
1	9	$+\infty$	$\min\{u[0][9], w[1] + u[0][6]\} = \min\{+\infty, +\infty\} = +\infty$
1	10	$+\infty$	$\min\{u[0][10], w[1] + u[0][7]\} = \min\{+\infty, +\infty\} = +\infty$
2	0	0	$u[2][0] \leftarrow u[1][0] = 0$
2	1	$+\infty$	$u[2][1] \leftarrow u[1][1] = +\infty$
2	2	$+\infty$	$u[2][2] \leftarrow u[1][2] = +\infty$
2	3	2	$\min\{u[1][3], w[2] + u[1][0]\} = \min\{2, 3 + 0\} = 2$
2	4	$+\infty$	$\min\{u[1][4], w[2] + u[1][1]\} = \min\{+\infty, +\infty\} = +\infty$
2	5	$+\infty$	$\min\{u[1][5], w[2] + u[1][2]\} = \min\{+\infty, +\infty\} = +\infty$
2	6	5	$\min\{u[1][6], w[2] + u[1][3]\} = \min\{+\infty, 3 + 2\} = 5$
2	7	$+\infty$	$\min\{u[1][7], w[2] + u[1][4]\} = \min\{+\infty, +\infty\} = +\infty$
2	8	$+\infty$	$\min\{u[1][8], w[2] + u[1][5]\} = \min\{+\infty, +\infty\} = +\infty$
2	9	$+\infty$	$\min\{u[1][9], w[2] + u[1][6]\} = \min\{+\infty, +\infty\} = +\infty$
2	10	$+\infty$	$\min\{u[1][10], w[2] + u[1][7]\} = \min\{+\infty, +\infty\} = +\infty$
3	0	0	$u[3][0] \leftarrow u[2][0] = 0$
3	1	$+\infty$	$u[3][1] \leftarrow u[2][1] = +\infty$
3	2	$+\infty$	$u[3][2] \leftarrow u[2][2] = +\infty$
3	3	2	$u[3][3] \leftarrow u[2][3] = 2$
3	4	3	$\min\{u[2][4], w[3] + u[2][0]\} = \min\{+\infty, 3 + 0\} = 3$
3	5	$+\infty$	$\min\{u[2][5], w[3] + u[2][1]\} = \min\{+\infty, +\infty\} = +\infty$
3	6	5	$\min\{u[2][6], w[3] + u[2][2]\} = \min\{5, +\infty\} = 5$
3	7	5	$\min\{u[2][7], w[3] + u[2][3]\} = \min\{+\infty, 3 + 2\} = 5$
3	8	$+\infty$	$\min\{u[2][8], w[3] + u[2][4]\} = \min\{+\infty, +\infty\} = +\infty$
3	9	$+\infty$	$\min\{u[2][9], w[3] + u[2][5]\} = \min\{+\infty, +\infty\} = +\infty$
3	10	8	$\min\{u[2][10], w[3] + u[2][6]\} = \min\{+\infty, 3 + 5\} = 8$

Dopo l'esecuzione dei due cicli annidati dell'algoritmo di PD, questa è la matrice u:

$u[j][q]$	0	1	2	3	4	5	6	7	8	9	10
$j=0$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$j=1$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$j=2$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$j=3$	0	$+\infty$	$+\infty$	2	3	$+\infty$	5	5	$+\infty$	$+\infty$	8

L'ultimo ciclo dell'algoritmo di PD scorre l'ultima riga della matrice u e tiene il più grande profitto obiettivo il cui minimo peso totale non supera la capacità $c=5$ dello zaino:

q	$u[3][q]$	Azione
0	0	$u[3][0]=0\leq 5$: $z\leftarrow 0$
1	$+\infty$	$u[3][1]=+\infty>5$: nessun aggiornamento
2	$+\infty$	$u[3][2]=+\infty>5$: nessun aggiornamento
3	2	$u[3][3]=2\leq 5$: $z\leftarrow 3$
4	3	$u[3][4]=3\leq 5$: $z\leftarrow 4$
5	$+\infty$	$u[3][5]=+\infty>5$: nessun aggiornamento
6	5	$u[3][6]=5\leq 5$: $z\leftarrow 6$
7	5	$u[3][7]=5\leq 5$: $z\leftarrow 7$
8	$+\infty$	$u[3][8]=+\infty>5$: nessun aggiornamento
9	$+\infty$	$u[3][9]=+\infty>5$: nessun aggiornamento
10	8	$u[3][10]=8>5$: nessun aggiornamento

Quindi il valore ottimo dell'istanza del KP è $z=7$.

- Dopo l'esecuzione dell'algoritmo di PD, la matrice u contiene il valore ottimo $u[j][q]=u(j,q)$ per ogni oggetto $j\in\{0,1,\ldots,n\}$ e ogni profitto obiettivo $q\in\{0,1,\ldots,P\}$.

- L'algoritmo di PD calcola la riga j della matrice u 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 $u[0..P]$, che usa $P+1$ celle invece delle $(n+1)(P+1)$ celle della matrice.

- Il vettore si inizializza con i casi base $u[0]=0$ e $u[q]=+\infty$ per ogni $q\in\{1,2,\ldots,P\}$, 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 $u[q]$ viene aggiornata per l'oggetto j, il valore $u[q-p_j]$ deve essere ancora quello della riga $j-1$, cioè $u(j-1,q-p_j)$.

– Poiché $q - p_j < q$, questo è garantito se i profitti obiettivo sono considerati in ordine **decrescente** $q = P, P-1, \ldots, p_j$: quando la cella q viene aggiornata, la cella $q - p_j$ non è ancora stata aggiornata nella passata corrente.

– Se i profitti obiettivo fossero considerati in ordine crescente, il valore $u[q-p_j]$ sarebbe già quello della riga j, cioè $u(j, q - p_j)$, e l'oggetto j potrebbe essere selezionato più di una volta, cosa non consentita nel KP.

– Per ogni profitto obiettivo $q < p_j$, il profitto dell'oggetto j supera il profitto obiettivo e quindi $u[q]$ deve mantenere il valore $u(j-1, q)$: non si esegue alcun aggiornamento e il ciclo interno può fermarsi a $q = p_j$.

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

Algoritmo

Algorithm: DP2_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 DP2_Vector(n, p, w, c)
2      P := 0
3      for j := 1 to n do
4          P ← P + p[j]
5      end
6      u := new array[0..P]
7      u[0] ← 0
8      for q := 1 to P do
9          u[q] ← +∞
10     end
11     for j := 1 to n do
12         for q := P downto p[j] do
13             if w[j] + u[q − p[j]] < u[q] then
14                 u[q] ← w[j] + u[q − p[j]]
15             end
16         end
17     end
18     z := 0
19     for q := 0 to P do
20         if u[q] ≤ c then
21             z ← q
22         end
23     end
24     return z
```

* Il vettore $u[0..P]$ è la **lookup table**: al termine della passata per l'oggetto j, memorizza la riga j della matrice u dell'algoritmo di PD.
* Il secondo ciclo inizializza i valori $u[0] = 0$ e $u[q] = +\infty$ per tutti i profitti obiettivo $q \in \{1, 2, \ldots, P\}$, 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.
* Il ciclo interno, su $q \in \{p[j], p[j] + 1, \ldots, P\}$, considera i profitti obiettivo 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 $u[q]$, con il valore ottenuto selezionando l'oggetto j, cioè $w[j]+u[q-p[j]]$. La cella viene aggiornata solo quando selezionare l'oggetto j è strettamente migliore.
* L'ultimo ciclo scorre il vettore e restituisce il più grande profitto obiettivo il cui minimo peso totale non supera la capacità dello zaino, come nella formula (2.5).

Esempio 2.17: Esecuzione dell'algoritmo di PD con un solo vettore

Consideriamo l'istanza del KP dell'Esempio 2.13, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	3	3	4
w_j	2	3	3

Il profitto totale di tutti gli oggetti è $P = 3 + 3 + 4 = 10$ e il secondo ciclo inizializza $u[0] = 0$ e $u[q] = +\infty$ per ogni $q \in \{1, 2, \ldots, 10\}$. Per ogni passo dei due cicli annidati, la tabella riporta i valori di j e q, la cella $u[q]$ dopo il passo e l'operazione eseguita.

j	q	$u[q]$	Azione
1	10	$+\infty$	$w[1]+u[7] = +\infty \geq u[10] = +\infty$: nessun aggiornamento
1	9	$+\infty$	$w[1] + u[6] = +\infty \geq u[9] = +\infty$: nessun aggiornamento
1	8	$+\infty$	$w[1] + u[5] = +\infty \geq u[8] = +\infty$: nessun aggiornamento
1	7	$+\infty$	$w[1] + u[4] = +\infty \geq u[7] = +\infty$: nessun aggiornamento
1	6	$+\infty$	$w[1] + u[3] = +\infty \geq u[6] = +\infty$: nessun aggiornamento
1	5	$+\infty$	$w[1] + u[2] = +\infty \geq u[5] = +\infty$: nessun aggiornamento
1	4	$+\infty$	$w[1] + u[1] = +\infty \geq u[4] = +\infty$: nessun aggiornamento
1	3	2	$w[1] + u[0] = 2 < u[3] = +\infty$: $u[3] \leftarrow 2$
2	10	$+\infty$	$w[2]+u[7] = +\infty \geq u[10] = +\infty$: nessun aggiornamento
2	9	$+\infty$	$w[2] + u[6] = +\infty \geq u[9] = +\infty$: nessun aggiornamento
2	8	$+\infty$	$w[2] + u[5] = +\infty \geq u[8] = +\infty$: nessun aggiornamento

2	7	$+\infty$	$w[2]+u[4]=+\infty \geq u[7]=+\infty$: nessun aggiornamento
2	6	5	$w[2]+u[3]=5<u[6]=+\infty$: $u[6]\leftarrow 5$
2	5	$+\infty$	$w[2]+u[2]=+\infty \geq u[5]=+\infty$: nessun aggiornamento
2	4	$+\infty$	$w[2]+u[1]=+\infty \geq u[4]=+\infty$: nessun aggiornamento
2	3	2	$w[2]+u[0]=3 \geq u[3]=2$: nessun aggiornamento
3	10	8	$w[3]+u[6]=8<u[10]=+\infty$: $u[10]\leftarrow 8$
3	9	$+\infty$	$w[3]+u[5]=+\infty \geq u[9]=+\infty$: nessun aggiornamento
3	8	$+\infty$	$w[3]+u[4]=+\infty \geq u[8]=+\infty$: nessun aggiornamento
3	7	5	$w[3]+u[3]=5<u[7]=+\infty$: $u[7]\leftarrow 5$
3	6	5	$w[3]+u[2]=+\infty \geq u[6]=5$: nessun aggiornamento
3	5	$+\infty$	$w[3]+u[1]=+\infty \geq u[5]=+\infty$: nessun aggiornamento
3	4	3	$w[3]+u[0]=3<u[4]=+\infty$: $u[4]\leftarrow 3$

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

$u[q]$	0	1	2	3	4	5	6	7	8	9	10
inizializzazione	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
passata $j=1$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
passata $j=2$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$
passata $j=3$	0	$+\infty$	$+\infty$	2	3	$+\infty$	5	5	$+\infty$	$+\infty$	8

Ogni passata trasforma il vettore u dalla riga $j-1$ nella riga j della matrice u riportata nell'Esempio 2.16. L'ultimo ciclo scorre poi il vettore e dà il valore ottimo $z=7$ dell'istanza del KP, come nell'Esempio 2.16.

- 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 profitto obiettivo q, e quindi non si può memorizzare in un solo vettore.

- Passo 2: calcolo di una soluzione ottima
  - 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 profitto obiettivo q, oltre al valore ottimo $u[j][q]$, memorizziamo $s[j][q]$, che indica se l'oggetto j è selezionato in un sottoinsieme di peso minimo degli oggetti $\{1, 2, \ldots, j\}$ con profitto totale uguale a q.
  - Più precisamente, $s[j][q]$ è uguale a 1 se l'oggetto j è selezionato nel calcolo del valore $u[j][q]$, 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:

Algoritmo

Algorithm: DP2_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 $s[0..n][0..P]$

```
1  Function DP2_InfoSol(n, p, w, c)
2      P := 0
3      for j := 1 to n do
4          P ← P + p[j]
5      end
6      u := new array[0..n][0..P]
7      s := new array[0..n][0..P]
8      u[0][0] ← 0
9      s[0][0] ← 0
10     for q := 1 to P do
11         u[0][q] ← +∞
12         s[0][q] ← 0
13     end
14     for j := 1 to n do
15         for q := 0 to P do
16             if q < p[j] then
17                 u[j][q] ← u[j − 1][q]
18                 s[j][q] ← 0
19             else
20                 if w[j] + u[j − 1][q − p[j]] < u[j − 1][q] then
21                     u[j][q] ← w[j] + u[j − 1][q − p[j]]
22                     s[j][q] ← 1
23                 else
24                     u[j][q] ← u[j − 1][q]
25                     s[j][q] ← 0
26                 end
27             end
28         end
29     end
30     z := 0
31     for q := 0 to P do
32         if u[n][q] ≤ c then
33             z ← q
34         end
35     end
36     return z, s
```

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

Esempio 2.18: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza del KP dell'Esempio 2.13, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$
p_j	3	3	4
w_j	2	3	3

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

j	q	$s[j][q]$	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	$s[0][5] \leftarrow 0$
0	6	0	$s[0][6] \leftarrow 0$
0	7	0	$s[0][7] \leftarrow 0$
0	8	0	$s[0][8] \leftarrow 0$
0	9	0	$s[0][9] \leftarrow 0$
0	10	0	$s[0][10] \leftarrow 0$
1	0	0	$0 < 3$: $s[1][0] \leftarrow 0$
1	1	0	$1 < 3$: $s[1][1] \leftarrow 0$
1	2	0	$2 < 3$: $s[1][2] \leftarrow 0$
1	3	1	$w[1] + u[0][0] = 2 + 0 < u[0][3] = +\infty$: $s[1][3] \leftarrow 1$
1	4	0	$w[1] + u[0][1] = +\infty \geq u[0][4] = +\infty$: $s[1][4] \leftarrow 0$
1	5	0	$w[1] + u[0][2] = +\infty \geq u[0][5] = +\infty$: $s[1][5] \leftarrow 0$
1	6	0	$w[1] + u[0][3] = +\infty \geq u[0][6] = +\infty$: $s[1][6] \leftarrow 0$
1	7	0	$w[1] + u[0][4] = +\infty \geq u[0][7] = +\infty$: $s[1][7] \leftarrow 0$
1	8	0	$w[1] + u[0][5] = +\infty \geq u[0][8] = +\infty$: $s[1][8] \leftarrow 0$
1	9	0	$w[1] + u[0][6] = +\infty \geq u[0][9] = +\infty$: $s[1][9] \leftarrow 0$
1	10	0	$w[1] + u[0][7] = +\infty \geq u[0][10] = +\infty$: $s[1][10] \leftarrow 0$
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	0	$w[2] + u[1][0] = 3 + 0 \geq u[1][3] = 2$: $s[2][3] \leftarrow 0$
2	4	0	$w[2] + u[1][1] = +\infty \geq u[1][4] = +\infty$: $s[2][4] \leftarrow 0$
2	5	0	$w[2] + u[1][2] = +\infty \geq u[1][5] = +\infty$: $s[2][5] \leftarrow 0$
2	6	1	$w[2] + u[1][3] = 3 + 2 < u[1][6] = +\infty$: $s[2][6] \leftarrow 1$
2	7	0	$w[2] + u[1][4] = +\infty \geq u[1][7] = +\infty$: $s[2][7] \leftarrow 0$

2	8	0	$w[2]+u[1][5]=+\infty \geq u[1][8]=+\infty$: $s[2][8] \leftarrow 0$
2	9	0	$w[2]+u[1][6]=+\infty \geq u[1][9]=+\infty$: $s[2][9] \leftarrow 0$
2	10	0	$w[2]+u[1][7]=+\infty \geq u[1][10]=+\infty$: $s[2][10] \leftarrow 0$
3	0	0	$0<4$: $s[3][0] \leftarrow 0$
3	1	0	$1<4$: $s[3][1] \leftarrow 0$
3	2	0	$2<4$: $s[3][2] \leftarrow 0$
3	3	0	$3<4$: $s[3][3] \leftarrow 0$
3	4	1	$w[3]+u[2][0]=3+0<u[2][4]=+\infty$: $s[3][4] \leftarrow 1$
3	5	0	$w[3]+u[2][1]=+\infty \geq u[2][5]=+\infty$: $s[3][5] \leftarrow 0$
3	6	0	$w[3]+u[2][2]=+\infty \geq u[2][6]=5$: $s[3][6] \leftarrow 0$
3	7	1	$w[3]+u[2][3]=3+2<u[2][7]=+\infty$: $s[3][7] \leftarrow 1$
3	8	0	$w[3]+u[2][4]=+\infty \geq u[2][8]=+\infty$: $s[3][8] \leftarrow 0$
3	9	0	$w[3]+u[2][5]=+\infty \geq u[2][9]=+\infty$: $s[3][9] \leftarrow 0$
3	10	1	$w[3]+u[2][6]=3+5<u[2][10]=+\infty$: $s[3][10] \leftarrow 1$

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

$s[j][q]$	0	1	2	3	4	5	6	7	8	9	10
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	0	1	0	0	0	0	0	0	0
$j=2$	0	0	0	0	0	0	1	0	0	0	0
$j=3$	0	0	0	0	1	0	0	1	0	0	1

– Calcolata la matrice s, si può ricostruire una soluzione ottima scorrendo gli oggetti da $j=n$ fino a $j=1$, partendo dal profitto obiettivo $q=z$ restituito dall'algoritmo di PD e da $x_j=0$ per ogni $j \in \{1,2,\ldots,n\}$:

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

– 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.

Algoritmo

Algorithm: DP2_Reconstruct

Input: Il numero n di oggetti, il valore ottimo z dell'istanza del KP, il vettore $p[1..n]$ dei profitti degli oggetti e la matrice $s[0..n][0..P]$

Output: Una soluzione ottima dell'istanza del KP

```
1  Function DP2_Reconstruct(n, z, p, s)
2      x := new array[1..n]
3      for j := 1 to n do
4          x[j] ← 0
5      end
6      q := z
7      for j := n downto 1 do
8          if s[j][q] = 1 then
9              x[j] ← 1
10             q ← q − p[j]
11         end
12     end
13     return x
```

Esempio 2.19: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 2.13, il valore ottimo è $z = 7$ e abbiamo la seguente matrice s:

$s[j][q]$	0	1	2	3	4	5	6	7	8	9	10
$j = 0$	0	0	0	0	0	0	0	0	0	0	0
$j = 1$	0	0	0	1	0	0	0	0	0	0	0
$j = 2$	0	0	0	0	0	0	1	0	0	0	0
$j = 3$	0	0	0	0	1	0	0	1	0	0	1

Possiamo ricostruire una soluzione ottima partendo da $j = 3$ e $q = z = 7$ come segue:

Iterazione	j	q	$s[j][q]$	$x[1]$	$x[2]$	$x[3]$	Azione
–	–	–	–	0	0	0	$q = 7$
1	3	7	1	0	0	1	$q \leftarrow 7 - 4 = 3$
2	2	3	0	0	0	1	nessun aggiornamento
3	1	3	1	1	0	1	$q \leftarrow 3 - 3 = 0$

Quindi il vettore

$$(1, 0, 1)$$

è una soluzione ottima. Il valore ottimo corrispondente è $3 + 4 = 7$.

2.3.1 Numero di passi

- Indichiamo con $S(n, P)$ il numero di passi eseguiti da `DP2` nel caso peggiore, dove n è il numero di oggetti e $P = \sum_{j=1}^{n} p_j$ è il profitto totale di tutti gli oggetti. Calcoliamo $S(n, P)$ contando quante volte vengono eseguite le righe dello pseudocodice di `DP2`.

- La condizione della riga 16 viene verificata solo quando $q \geq p[j]$, cioè per $P + 1 - p_j$ profitti obiettivo per ogni oggetto j, e quindi $n(P + 1) - P$ volte in tutto. Il numero di esecuzioni delle righe rilevanti è:

  - La Riga 2 viene eseguita 1 volta.
  - La Riga 3 viene eseguita n volte.
  - La Riga 4 viene eseguita n volte.
  - La Riga 6 viene eseguita 1 volta.
  - La Riga 7 viene eseguita 1 volta.
  - La Riga 8 viene eseguita P volte.
  - La Riga 9 viene eseguita P volte.
  - La Riga 11 viene eseguita n volte.
  - La Riga 12 viene eseguita $n(P + 1)$ volte.
  - La Riga 13 viene eseguita $n(P + 1)$ volte.
  - La Riga 16 viene eseguita $n(P + 1) - P$ volte.
  - una fra la Riga 14, la Riga 17 e la Riga 19 viene eseguita per ogni coppia (j, q), e quindi queste righe di assegnamento vengono eseguite in tutto $n(P + 1)$ volte.
  - La Riga 24 viene eseguita 1 volta.
  - La Riga 25 viene eseguita $P + 1$ volte.
  - La Riga 26 viene eseguita $P + 1$ volte.
  - La Riga 27 viene eseguita al più $P + 1$ volte.
  - La Riga 30 viene eseguita 1 volta.

  Pertanto, il numero totale di passi nel caso peggiore è

$$\begin{aligned} &1 + n + n + 1 + 1 + P + P + n + n(P + 1) + n(P + 1) \\ &+ \big(n(P + 1) - P\big) + n(P + 1) + 1 + (P + 1) + (P + 1) + (P + 1) + 1. \end{aligned}$$

  Quindi

$$S(n, P) = 4nP + 7n + 4P + 8.$$

  Dunque

$$S(n, P) = \Theta(nP).$$

  Infatti

$$\lim_{n,P\to\infty} \frac{4nP + 7n + 4P + 8}{nP} = \lim_{n,P\to\infty} \left(4 + \frac{7}{P} + \frac{4}{n} + \frac{8}{nP}\right) = 4 \in \mathbb{R}_{>0}.$$

L'algoritmo di PD esegue
$$S(n, P) = \Theta(nP)$$
passi nel caso peggiore, dove $P = \sum_{j=1}^{n} p_j$.

- La versione `DP2_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 $\Theta(nP)$.
- La versione `DP2_Vector` usa un solo vettore invece della matrice ed esegue gli stessi due cicli annidati, il cui ciclo interno esegue al più $P+1$ iterazioni per ogni oggetto. Pertanto, il suo numero di passi resta $\Theta(nP)$, mentre il numero di celle della lookup table scende da $(n+1)(P+1)$ a $P+1$.

2.3.2 Un'implementazione con liste

- Passo 1: calcolo del valore ottimo
  - L'algoritmo di PD di questa sezione memorizza un valore per ogni coppia formata da un oggetto e da un profitto obiettivo, cioè $(n+1)(P+1)$ valori in tutto. Molti di questi valori sono uguali a $+\infty$, perché nessun sottoinsieme degli oggetti disponibili ha quel profitto totale.
  - Questa sottosezione descrive una **diversa implementazione dello stesso algoritmo di PD**. La decomposizione ricorsiva e la sottostruttura ottima sono quelle della formula (2.4): cambia la struttura dati.
  - Invece di un'intera riga della matrice, memorizziamo solo le coppie ottenute effettivamente da un sottoinsieme di oggetti e, fra queste, solo quelle che possono portare a una soluzione ottima. Queste coppie sono tenute in una lista, una lista per ogni oggetto.

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 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, e in questo caso basta tenerne uno. Più in generale, uno stato si può scartare quando un altro stato è almeno altrettanto buono in entrambe le componenti.

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.

Per ogni $j \in \{0, 1, \ldots, n\}$, sia L_j la lista degli stati non dominati che si possono ottenere selezionando un sottoinsieme degli oggetti $\{1, 2, \ldots, j\}$, ordinata per peso strettamente crescente.

La formula ricorsiva per calcolare L_j è la seguente:

$$L_j = \begin{cases} \{(0,0)\} & \text{se } j = 0, \\ \mathrm{nd}\big(L_{j-1} \cup M_j\big) & \text{se } 1 \leq j \leq n, \end{cases} \tag{2.6}$$

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

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

– Il processo di decomposizione ricorsiva della formula (2.6) genera 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.

– Qui $L_0 = \{(0, 0)\}$ perché l'unico sottoinsieme di un insieme vuoto di oggetti è il sottoinsieme vuoto.

– Come nella formula (2.4), la capacità dello zaino non compare nella ricorsione: la lista L_j raccoglie gli stati di *tutti* i sottoinsiemi degli oggetti $\{1, 2, \ldots, j\}$, compresi quelli il cui peso totale supera la capacità.

– 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 $\{1, 2, \ldots, j-1\}$.

⇝ **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.

⇝ **Sottoproblemi sovrapposti verificati!**

– La capacità dello zaino si usa solo alla fine, esattamente come nella formula (2.5): fra gli stati di L_n, solo quelli il cui peso totale non supera la capacità corrispondono a una soluzione ammissibile.

Il valore ottimo dell'istanza del KP è

$$\max\{\, p \;:\; (w,p) \in L_n,\ w \le c \,\}. \tag{2.7}$$

– L'insieme della formula (2.7) non è mai vuoto, poiché $(0,0) \in L_n$ e $0 \le c$, e quindi il massimo è sempre ben definito.

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

$$\min\{\, w \;:\; (w,p) \in L_j,\ p \ge q \,\} = \min_{q' \ge q} u[j][q'] \qquad \forall q \in \{0,1,\ldots,P\}.$$

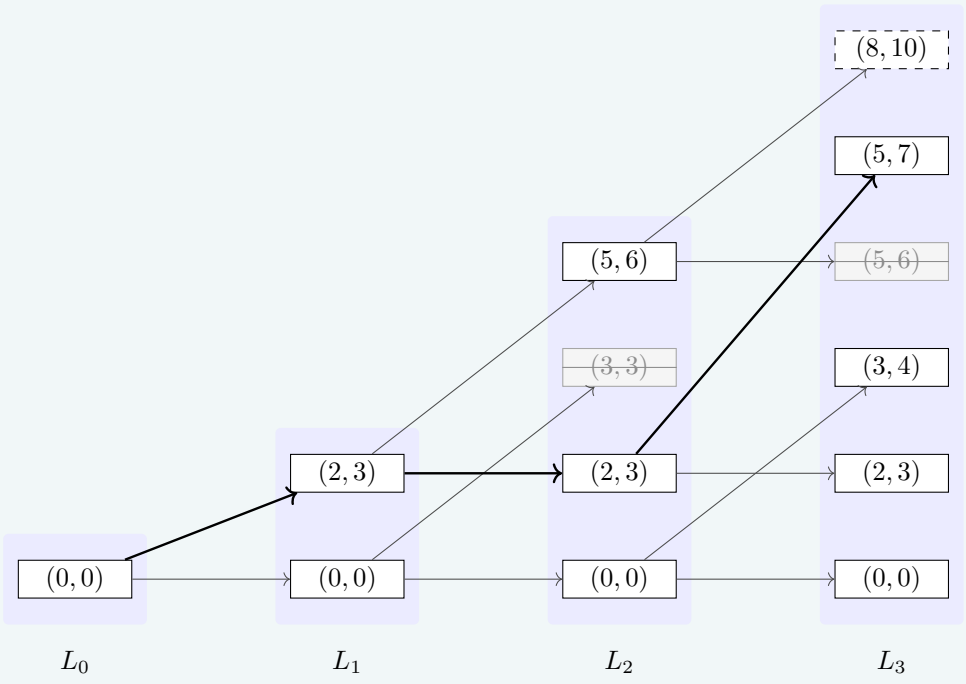
I profitti obiettivo con valore $+\infty$ nella riga non lasciano alcuno stato nella lista, e nemmeno un profitto obiettivo il cui peso minimo non è minore di quello di un profitto obiettivo più grande.

Esempio 2.20: Le liste di stati

Consideriamo l'istanza del KP dell'Esempio 2.13, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$
p_j	3	3	4
w_j	2	3	3

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.



I due stati barrati sono dominati e quindi vengono scartati: $(3, 3)$ è dominato da $(2, 3)$ e $(5, 6)$ è dominato da $(5, 7)$. Lo stato $(8, 10)$, con il bordo tratteggiato, ha un peso totale maggiore della capacità $c = 5$ dello zaino: resta nella lista, perché la capacità non si usa durante la ricorsione, e viene scartato solo dalla formula (2.7).

– 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 (2.6) si calcola scorrendo contemporaneamente le due liste, come nella fusione di due liste ordinate.

– 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**:

Algoritmo

Algorithm: DP2_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 DP2_Lists(n, p, w, c)
2      L := new list containing the single state (0, 0)
3      for j := 1 to n do
4          M := new empty list
5          for h := 1 to |L| do
6              accodare (L[h].w + w[j], L[h].p + p[j]) a M
7          end
8          L ← Merge(L, M)
9      end
10     z := 0
11     for h := 1 to |L| do
12         if L[h].w ≤ c then
13             z ← L[h].p
14         end
15     end
16     return z
```

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

Algoritmo

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
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
```

* 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. Qui nessuno stato viene scartato, perché la capacità dello zaino non si usa durante la ricorsione.
* 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.
* L'ultimo ciclo realizza la formula (2.7). Gli stati di L hanno pesi strettamente crescenti e profitti strettamente crescenti, e quindi la variabile ausiliaria z termina con il profitto dell'ultimo stato il cui peso non supera la capacità dello zaino.

∗ Infine, l'algoritmo restituisce il valore ottimo dell'istanza del KP, z.

Esempio 2.21: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del KP dell'Esempio 2.13, con uno zaino di capacità $c = 5$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$
p_j	3	3	4
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	$\{(2,3)\}$	$\{(0,0),(2,3)\}$
2	$\{(3,3),(5,6)\}$	$\{(0,0),(2,3),(5,6)\}$
3	$\{(3,4),(5,7),(8,10)\}$	$\{(0,0),(2,3),(3,4),(5,7),(8,10)\}$

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	$(2,3)$	M_1	$3 > \pi = 0$: accodare
2	$(0,0)$	L_1	$0 > \pi = -1$: accodare
2	$(2,3)$	L_1	$3 > \pi = 0$: accodare
2	$(3,3)$	M_2	$3 \leq \pi = 3$: scartare (dominato)
2	$(5,6)$	M_2	$6 > \pi = 3$: accodare
3	$(0,0)$	L_2	$0 > \pi = -1$: accodare
3	$(2,3)$	L_2	$3 > \pi = 0$: accodare
3	$(3,4)$	M_3	$4 > \pi = 3$: accodare
3	$(5,7)$	M_3	$7 > \pi = 4$: accodare
3	$(5,6)$	L_2	$6 \leq \pi = 7$: scartare (dominato)
3	$(8,10)$	M_3	$10 > \pi = 7$: accodare

L'ultimo ciclo scorre L_3 e tiene il profitto dell'ultimo stato il cui peso non supera $c = 5$: gli stati $(0,0)$, $(2,3)$, $(3,4)$ e $(5,7)$ danno $z \leftarrow 0$, $z \leftarrow 3$, $z \leftarrow 4$ e $z \leftarrow 7$, mentre lo stato $(8,10)$ non causa alcun aggiornamento poiché $8 > 5$. Pertanto il valore ottimo dell'istanza del KP è 7.

– Dopo l'esecuzione dell'algoritmo di PD, la lista L contiene gli stati non dominati L_n, ordinati per peso strettamente crescente e profitto strettamente crescente.

- Passo 2: calcolo di una soluzione ottima

- 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 $L_j[h]$.
- 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 2.22: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza del KP dell'Esempio 2.13. 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	$(2,3)$	1	1
2	1	$(0,0)$	0	1
2	2	$(2,3)$	0	2
2	3	$(5,6)$	1	2
3	1	$(0,0)$	0	1
3	2	$(2,3)$	0	2
3	3	$(3,4)$	1	1
3	4	$(5,7)$	1	2
3	5	$(8,10)$	1	3

Per esempio, lo stato $L_3[4] = (5,7)$ ha $\sigma_3[4] = 1$ e $\rho_3[4] = 2$: si ottiene selezionando l'oggetto 3 a partire dallo stato $L_2[2] = (2,3)$, poiché $2 + w_3 = 2 + 3 = 5$ e $3 + p_3 = 3 + 4 = 7$.

- La ricostruzione parte dalla posizione $\tilde{h}$ dello stato che raggiunge il massimo della formula (2.7), cioè la posizione in cui l'ultimo ciclo di `DP2_Lists` aggiorna z per l'ultima volta. In generale non è l'ultima posizione di L_n, perché gli ultimi stati di L_n possono avere un peso totale maggiore della capacità dello zaino.
- Si ricostruisce poi una soluzione ottima scorrendo gli oggetti da $j = n$ fino a $j = 1$, partendo da $h = \tilde{h}$:

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

– 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.

Algoritmo

Algorithm: DP2_Lists_Reconstruct

Input: Il numero n di oggetti, la posizione $\tilde{h}$ dello stato ottimo in L_n e i valori memorizzati σ e ρ

Output: Una soluzione ottima dell'istanza del KP

```
1  Function DP2_Lists_Reconstruct(n, h̃, σ, ρ)
2      x := new array[1..n]
3      for j := 1 to n do
4          x[j] ← 0
5      end
6      h := h̃
7      for j := n downto 1 do
8          x[j] ← σ_j[h]
9          h ← ρ_j[h]
10     end
11     return x
```

Esempio 2.23: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 2.13, l'ultima lista è

$$L_3 = \{(0,0), (2,3), (3,4), (5,7), (8,10)\}$$

e il massimo della formula (2.7) è raggiunto dallo stato $L_3[4] = (5,7)$, quindi $\tilde{h} = 4$ e il valore ottimo è 7.

Possiamo ricostruire una soluzione ottima partendo da $j = 3$ e $h = \tilde{h} = 4$ 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	$(5,7)$	1	2	0	0	1	$h \leftarrow 2$
2	2	2	$(2,3)$	0	2	0	0	1	$h \leftarrow 2$
3	1	2	$(2,3)$	1	1	1	0	1	$h \leftarrow 1$

Quindi il vettore

$$(1, 0, 1)$$

è una soluzione ottima. Il valore ottimo corrispondente è $3 + 4 = 7$.

2.3.3 Numero di passi

- Indichiamo con $S(n, P)$ il numero di passi eseguiti da `DP2_Lists` nel caso peggiore, dove n è il numero di oggetti e $P = \sum_{j=1}^{n} p_j$ è il profitto totale di tutti gli oggetti.

- Gli stati di una lista hanno profitti strettamente crescenti e ogni profitto appartiene a $\{0, 1, \ldots, P\}$: pertanto $|L_j| \leq P + 1$ per ogni $j \in \{0, 1, \ldots, n\}$, e $|M_j| = |L_{j-1}|$.

- Per ogni oggetto, `DP2_Lists` scorre una volta L_{j-1} per costruire M_j e poi chiama `Merge`, che scandisce $|L_{j-1}| + |M_j| \leq 2(P + 1)$ stati ed esegue un numero costante di operazioni per ciascuno di essi. Il ciclo finale scorre una volta L_n. Pertanto

$$S(n, P) = \Theta(nP).$$

- La matrice u ha sempre $(n + 1)(P + 1)$ celle, la maggior parte delle quali uguali a $+\infty$, mentre le liste contengono solo gli stati effettivamente raggiungibili. Sull'istanza del KP dell'Esempio 2.13 le liste contengono 1, 2, 3 e 5 stati, contro le 44 celle della matrice u.

- La lista L_j soddisfa anche $|L_j| \leq 2^j$, poiché gli oggetti $\{1, 2, \ldots, j\}$ hanno 2^j sottoinsiemi. Pertanto

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

2.4 Errori tipici

Attenzione

- **Aggiornare il vettore unico in ordine crescente di capacità.** La cella $v[k - w_j]$ conterrebbe già la riga j e l'oggetto j potrebbe essere selezionato più di una volta; le capacità si scorrono a partire da c verso il basso.

- **Dimenticare i casi base.** Senza oggetti disponibili il valore ottimo è 0 per ogni capacità: $v(0, k) = 0$ per ogni $k \in \{0, 1, \ldots, c\}$, non solo per $k = 0$.

- **Ricostruire con l'indice sbagliato.** A ogni passo l'indice dell'oggetto diminuisce di uno, mentre la capacità diminuisce di w_j solo se l'oggetto j è selezionato.

- **Chiamare polinomiale $\Theta(nc)$.** La capacità c si scrive con $\log c$ cifre, quindi il tempo di esecuzione è esponenziale nella dimensione dell'input.

- **Leggere $u(n, q)$ come il valore ottimo.** Nel secondo algoritmo di PD $u(n, q)$ è un peso minimo; il valore ottimo è il più grande q con $u(n, q) \leq c$.

2.5 In sintesi

In sintesi

- La ricorrenza $v(j, k)$ sceglie fra lasciare fuori l'oggetto j e prenderlo; ha la sottostruttura ottima e i sottoproblemi sovrapposti.
- La matrice si riempie bottom-up in $\Theta(nc)$ passi; un solo vettore scorso verso il basso dà il valore ottimo con sole $c + 1$ celle.
- Una soluzione ottima si ricava dalla matrice s delle decisioni, letta all'indietro a partire da (n, c).
- Il tempo di esecuzione $\Theta(nc)$ è pseudopolinomiale; le liste di stati non dominati memorizzano solo gli stati raggiungibili.
- Scambiando profitti e pesi si ottengono $\Theta(nP)$ passi, il che è meglio quando il profitto totale P è piccolo.

Cruscotto dell'esempio del KP. L'Esempio 2.1 ($c = 5$, tre oggetti) con i bound trovati finora:

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	14	14	valore ottimo, $\Theta(nc)$ passi

Capitolo 3

PD per il problema dello zaino illimitato

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: la ricorrenza $v(j, k)$ su oggetti e capacità e la ricostruzione di una soluzione.
- Da questo modulo, il capitolo «Principi della programmazione dinamica»: sottostruttura ottima, sottoproblemi sovrapposti, casi base.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- spiegare perché l'UKP differisce dal problema dello zaino 0/1 (KP) e scrivere la ricorrenza $v(k)$ sulle sole capacità;
- eseguire a mano l'algoritmo di PD sulle capacità e sulle coppie (tipo di oggetto, capacità);
- ricostruire una soluzione ottima, compreso il caso in cui una parte della capacità resta inutilizzata;
- contare i passi dei due algoritmi di PD e confrontarli.

Il problema di questo capitolo è il problema dello zaino illimitato (UKP): la sua definizione e le sue ipotesi sono nella sua pagina.

3.1 Istanze e soluzioni ammissibili

Esempio 3.1: Istanza dell'UKP

Consideriamo uno zaino di capacità $c = 7$ e 4 tipi di oggetto, con i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	4	5	10	11
w_j	2	3	5	6

Chiamiamo questa istanza dell'UKP Esempio 3.1.

Per ogni tipo di oggetto j, una soluzione ammissibile può usare al più $\lfloor c/w_j \rfloor$ copie del tipo di oggetto j, poiché con più copie si supererebbe la capacità c già con il solo tipo di oggetto j. Di conseguenza, il numero di soluzioni ammissibili è al più $\prod_{j=1}^{n} \left(\lfloor c/w_j \rfloor + 1\right)$.

- Il numero di possibili soluzioni dell'UKP cresce rapidamente con il numero n di tipi di oggetto e con la capacità c, poiché ogni variabile x_j può assumere in modo indipendente uno qualsiasi di $\lfloor c/w_j \rfloor + 1$ valori. Pertanto, l'enumerazione di tutte le possibili soluzioni diventa presto impraticabile al crescere di n o di c.

3.1.1 Codifica delle soluzioni ammissibili

Una soluzione ammissibile è descritta dal vettore intero

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

dove $x_j \in \mathbb{Z}_{\geq 0}$ è il numero di copie selezionate del tipo di oggetto j. La soluzione è ammissibile se

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

Il suo profitto totale è

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

La differenza principale rispetto al KP è che le variabili non sono più binarie. Nell'UKP ogni variabile x_j può assumere qualunque valore intero non negativo, purché il vincolo di capacità sia rispettato.

3.2 Un primo 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 dell'UKP data da:

1. uno zaino di capacità k,
2. l'insieme $\{1, 2, \ldots, n\}$ di n tipi di oggetto,
3. per ogni tipo di oggetto $j \in \{1, 2, \ldots, n\}$, un profitto p_j e un peso w_j.

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

$$v(k) = \begin{cases} 0, & \text{se } k = 0, \\ \max\limits_{j\,:\,w_j \le k} \{p_j + v(k - w_j)\}, & \text{se } k \ge 1 \text{ e } w_j \le k \text{ per qualche } j, \\ 0, & \text{se } k \ge 1 \text{ e } w_j > k \text{ per ogni } j. \end{cases} \tag{3.1}$$

- Il processo di decomposizione ricorsiva della formula (3.1) genera al più n **sottoproblemi indipendenti**, ciascuno corrispondente a una possibile scelta di un tipo di oggetto j, fra quelli con $w_j \le k$, da inserire come uno degli oggetti dello zaino di capacità k.
- Il caso base è $v(0) = 0$, poiché uno zaino di capacità 0 non può contenere alcun oggetto e ha profitto 0. Inoltre, se $w_j > k$ per ogni tipo di oggetto j, nessun oggetto entra nello zaino: l'unica scelta ammissibile è non inserire alcun oggetto, e quindi $v(k) = 0$.
- Il valore ottimo $v(k)$ si ottiene provando ogni tipo di oggetto j con $w_j \le k$ e scegliendo quello che massimizza la somma $p_j + v(k - w_j)$ del profitto del tipo di oggetto scelto e del valore ottimo del sottoproblema residuo.
- Poiché il tipo di oggetto j è disponibile in infinite copie, può essere scelto di nuovo anche nella soluzione del sottoproblema residuo di capacità $k - w_j$.
- Quindi una soluzione ottima per uno zaino di capacità k si costruisce combinando il tipo di oggetto j scelto con una soluzione ottima del sottoproblema residuo di capacità $k - w_j$.

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

- Scelte diverse del primo tipo di oggetto, fatte eventualmente per valori diversi di k, possono portare alla stessa capacità residua; quindi lo stesso valore $v(k)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (3.1).

⇝ **Sottoproblemi sovrapposti verificati!**

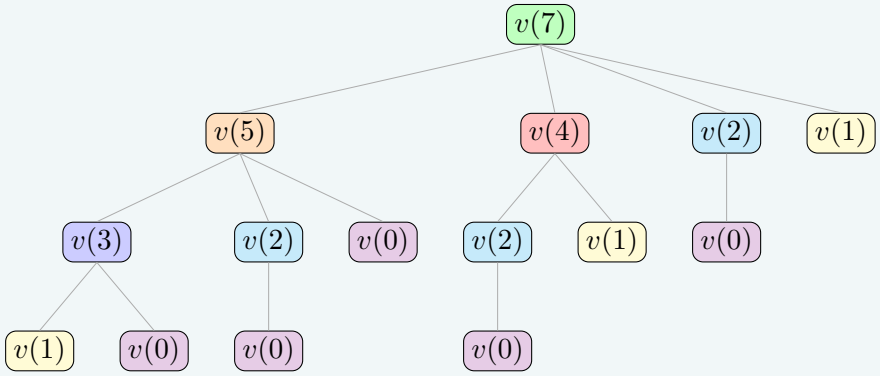
– Un **primo metodo** per calcolare il valore ottimo $v(c)$ dell'istanza dell'UKP è l'**approccio top-down**.

– Il metodo parte dalla chiamata con $k = c$ e valuta ricorsivamente la formula (3.1): ogni chiamata per uno zaino di capacità k genera altre chiamate per la capacità $k - w_j$, per ogni tipo di oggetto j con $w_j \leq k$, finché si raggiunge il caso base $v(0)$ o nessun tipo di oggetto entra nella capacità residua.

– 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 3.2: Albero di ricorsione

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con capacità $c = 7$ e tipi di oggetto di pesi $w_1 = 2$, $w_2 = 3$, $w_3 = 5$, $w_4 = 6$. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(7)$:



La funzione v viene chiamata 15 volte e gli stessi sottoproblemi vengono risolti più volte: $v(2)$ viene chiamata 3 volte, $v(1)$ viene chiamata 3 volte e $v(0)$ viene chiamata 5 volte. Tutte le altre chiamate avvengono una sola volta.

– Il valore $v(k)$, per una capacità generica k, si può calcolare quando sono noti i valori $v(k')$ per tutte le capacità più piccole $k' < k$.

Esempio 3.3: Calcolo dei valori della funzione v

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con uno zaino di capacità $c = 7$ e i seguenti profitti e pesi:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

Il valore della funzione v si può calcolare partendo dal caso base $v(0) = 0$ e

calcolando poi in ordine i valori $v(1), v(2), \ldots, v(7)$, come segue:

* Poiché nessun tipo di oggetto ha $w_j \leq 1$, abbiamo:

$$v(1) = 0.$$

* Calcolato $v(0)$, possiamo calcolare $v(2)$ come segue:

$$v(2) = p_1 + v(0) = 4 + 0 = 4.$$

* Calcolati $v(1)$ e $v(0)$, possiamo calcolare $v(3)$ come segue:

$$v(3) = \max\{p_1 + v(1), p_2 + v(0)\} = \max\{4 + 0, 5 + 0\} = 5.$$

* Calcolati $v(2)$ e $v(1)$, possiamo calcolare $v(4)$ come segue:

$$v(4) = \max\{p_1 + v(2), p_2 + v(1)\} = \max\{4 + 4, 5 + 0\} = 8.$$

* Calcolati $v(3)$, $v(2)$ e $v(0)$, possiamo calcolare $v(5)$ come segue:

$$v(5) = \max\{p_1 + v(3), p_2 + v(2), p_3 + v(0)\} = \max\{4+5, 5+4, 10+0\} = 10.$$

* Calcolati $v(4)$, $v(3)$, $v(1)$ e $v(0)$, possiamo calcolare $v(6)$ come segue:

$$\begin{aligned} v(6) &= \max\{p_1 + v(4), p_2 + v(3), p_3 + v(1), p_4 + v(0)\} \\ &= \max\{4 + 8, 5 + 5, 10 + 0, 11 + 0\} = 12. \end{aligned}$$

* Calcolati $v(5)$, $v(4)$, $v(2)$ e $v(1)$, possiamo calcolare $v(7)$ come segue:

$$\begin{aligned} v(7) &= \max\{p_1 + v(5), p_2 + v(4), p_3 + v(2), p_4 + v(1)\} \\ &= \max\{4 + 10, 5 + 8, 10 + 4, 11 + 0\} = 14. \end{aligned}$$

– 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 un sottoproblema di valore $v(k)$ dipende solo da sottoproblemi di valori $v(k')$, con $k' < k$, possiamo risolvere i sottoproblemi per valori crescenti 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 dell'UKP.

– Un **secondo metodo** per calcolare il valore ottimo $v(c)$ dell'istanza dell'UKP è l'**approccio bottom-up**.

– Il metodo parte dal caso base $v(0) = 0$ e valuta la formula (3.1) per capacità crescenti $k \in \{1, 2, \ldots, c\}$: ogni valore $v(k)$ si calcola con i valori già calcolati $v(0), v(1), \ldots, v(k-1)$, fino a ottenere il valore $v(c)$ dell'istanza dell'UKP.

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

Algoritmo

Algorithm: UKP_DP1

Input: Un'istanza dell'UKP data da n, dai vettori $p[1..n]$, $w[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza dell'UKP

```
1  Function UKP_DP1(n, p, w, c)
2      v := new array[0..c]
3      v[0] ← 0
4      for k := 1 to c do
5          m := 0
6          for j := 1 to n do
7              if w[j] ≤ k then
8                  if p[j] + v[k − w[j]] > m then
9                      m ← p[j] + v[k − w[j]]
10                 end
11             end
12         end
13         v[k] ← m
14     end
15     return v[c]
```

* Il vettore $v[0..c]$ è la **lookup table**: memorizza i valori ottimi di tutti i sottoproblemi, con il caso base $v[0] = 0$.
* Il ciclo esterno, su $k \in \{1, 2, \ldots, c\}$, calcola $v[k]$ in ordine crescente di capacità.
* Il ciclo interno, su $j \in \{1, 2, \ldots, n\}$, esamina ogni tipo di oggetto; la condizione $w[j] \leq k$ verifica se il tipo di oggetto j entra nella capacità k, e in m si tiene il valore $p[j] + v[k - w[j]]$ più grande trovato finora. Se nessun tipo di oggetto entra, m resta uguale al suo valore iniziale 0.
* Al termine del ciclo interno, $v[k] \leftarrow m$ memorizza il valore ottimo $v[k]$.
* Infine, l'algoritmo restituisce il valore ottimo $v[c]$ dell'istanza dell'UKP.

Esempio 3.4: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con uno zaino di capacità $c = 7$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	4	5	10	11
w_j	2	3	5	6

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

k	j	$v[k]$	m	Azione
–	–	0	–	$v[0] \leftarrow 0$
	–	–	0	$m := 0$
1	1	–	0	$w[1] = 2 > 1$: saltare
	2	–	0	$w[2] = 3 > 1$: saltare
	3	–	0	$w[3] = 5 > 1$: saltare
	4	–	0	$w[4] = 6 > 1$: saltare
	–	0	0	$v[1] \leftarrow m = 0$
	–	–	0	$m := 0$
2	1	–	4	$p[1]+v[0] = 4+0 = 4 > m = 0$: $m \leftarrow 4$
	2	–	4	$w[2] = 3 > 2$: saltare
	3	–	4	$w[3] = 5 > 2$: saltare
	4	–	4	$w[4] = 6 > 2$: saltare
	–	4	4	$v[2] \leftarrow m = 4$
	–	–	0	$m := 0$
3	1	–	4	$p[1]+v[1] = 4+0 = 4 > m = 0$: $m \leftarrow 4$
	2	–	5	$p[2]+v[0] = 5+0 = 5 > m = 4$: $m \leftarrow 5$
	3	–	5	$w[3] = 5 > 3$: saltare
	4	–	5	$w[4] = 6 > 3$: saltare
	–	5	5	$v[3] \leftarrow m = 5$
	–	–	0	$m := 0$
4	1	–	8	$p[1]+v[2] = 4+4 = 8 > m = 0$: $m \leftarrow 8$
	2	–	8	$p[2]+v[1] = 5+0 = 5 \leq m = 8$: nessun aggiornamento
	3	–	8	$w[3] = 5 > 4$: saltare
	4	–	8	$w[4] = 6 > 4$: saltare
	–	8	8	$v[4] \leftarrow m = 8$
	–	–	0	$m := 0$
5	1	–	9	$p[1]+v[3] = 4+5 = 9 > m = 0$: $m \leftarrow 9$
	2	–	9	$p[2]+v[2] = 5+4 = 9 = m$: nessun aggiornamento
	3	–	10	$p[3]+v[0] = 10+0 = 10 > m = 9$: $m \leftarrow 10$
	4	–	10	$w[4] = 6 > 5$: saltare
	–	10	10	$v[5] \leftarrow m = 10$

k	j	$v[k]$	m	Azione
	–	–	0	$m := 0$
6	1	–	12	$p[1]+v[4] = 4+8 = 12 > m = 0$: $m \leftarrow 12$
	2	–	12	$p[2]+v[3] = 5+5 = 10 \leq m = 12$: nessun aggiornamento
	3	–	12	$p[3]+v[1] = 10+0 = 10 \leq m = 12$: nessun aggiornamento
	4	–	12	$p[4]+v[0] = 11+0 = 11 \leq m = 12$: nessun aggiornamento
	–	12	12	$v[6] \leftarrow m = 12$
	–	–	0	$m := 0$
7	1	–	14	$p[1]+v[5] = 4+10 = 14 > m = 0$: $m \leftarrow 14$
	2	–	14	$p[2]+v[4] = 5+8 = 13 \leq m = 14$: nessun aggiornamento
	3	–	14	$p[3]+v[2] = 10+4 = 14 = m$: nessun aggiornamento
	4	–	14	$p[4]+v[1] = 11+0 = 11 \leq m = 14$: nessun aggiornamento
	–	14	14	$v[7] \leftarrow m = 14$

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

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$v[k]$	0	0	4	5	8	10	12	14

- Dopo l'esecuzione dell'algoritmo di PD, il vettore v contiene il valore ottimo $v[k]$ per ogni capacità.

- Passo 2: calcolo di una soluzione ottima
  - Per calcolare una soluzione ottima è necessario memorizzare alcune informazioni aggiuntive.
  - Per ogni capacità k, oltre al valore ottimo $v[k]$, memorizziamo $s[k]$, il tipo di oggetto selezionato come primo oggetto in una soluzione ottima per uno zaino di capacità k; se nessun tipo di oggetto entra nella capacità k, poniamo $s[k] = 0$.
  - Partendo da $k = c$, il valore $s[k]$ dà il tipo del primo oggetto selezionato; la capacità residua diventa $k - w_{s[k]}$ e si può applicare di nuovo la stessa regola, e così via, finché la capacità residua arriva a 0 o nessun tipo di oggetto vi entra ($s[k] = 0$); in questo caso la capacità residua resta inutilizzata.

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

Algoritmo

Algorithm: UKP_DP1_InfoSol

Input: Un'istanza dell'UKP data da n, dai vettori $p[1..n]$, $w[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza dell'UKP e il vettore delle informazioni $s[0..c]$

```
1  Function UKP_DP1_InfoSol(n, p, w, c)
2      v := new array[0..c]; s := new array[0..c]
3      v[0] ← 0; s[0] ← 0
4      for k := 1 to c do
5          m := 0; t := 0
6          for j := 1 to n do
7              if w[j] ≤ k then
8                  if p[j] + v[k − w[j]] > m then
9                      m ← p[j] + v[k − w[j]]; t ← j
10                 end
11             end
12         end
13         v[k] ← m; s[k] ← t
14     end
15     return v[c], s
```

* L'algoritmo di PD usa la variabile temporanea aggiuntiva t per registrare la decisione presa quando m viene aggiornato. Se il tipo di oggetto j è selezionato, $t \leftarrow j$ memorizza questa decisione; se nessun tipo di oggetto entra nella capacità k, t mantiene il suo valore iniziale 0.

* Al termine del ciclo interno, $s[k] \leftarrow t$ memorizza il tipo di oggetto selezionato, insieme al valore ottimo $v[k] \leftarrow m$.

Esempio 3.5: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con uno zaino di capacità $c = 7$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	4	5	10	11
w_j	2	3	5	6

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di k e j, la cella $s[k]$, il valore corrente di m e il valore corrente di t. La cella di m è evidenziata in rosso quando viene aggiornata; la cella di t e la cella di $s[k]$ sono evidenziate in blu quando vengono aggiornate.

k	j	$s[k]$	m	t	Azione
–	–	–	–	–	Stato iniziale: $v[0] \leftarrow 0$, $s[0] \leftarrow 0$
	–	–	0	0	$m := 0$, $t := 0$
1	1	–	0	0	$w[1] = 2 > 1$: saltare
	2	–	0	0	$w[2] = 3 > 1$: saltare
	3	–	0	0	$w[3] = 5 > 1$: saltare
	4	–	0	0	$w[4] = 6 > 1$: saltare
	–	0	0	0	$v[1] \leftarrow m = 0$, $s[1] \leftarrow t = 0$
	–	–	0	0	$m := 0$, $t := 0$
2	1	–	4	1	$4{+}0 = 4 > m = 0$: $m \leftarrow 4$, $t \leftarrow 1$
	2	–	4	1	$w[2] = 3 > 2$: saltare
	3	–	4	1	$w[3] = 5 > 2$: saltare
	4	–	4	1	$w[4] = 6 > 2$: saltare
	–	1	4	1	$v[2] \leftarrow m = 4$, $s[2] \leftarrow t = 1$
	–	–	0	0	$m := 0$, $t := 0$
3	1	–	4	1	$4{+}0 = 4 > m = 0$: $m \leftarrow 4$, $t \leftarrow 1$
	2	–	5	2	$5{+}0 = 5 > m = 4$: $m \leftarrow 5$, $t \leftarrow 2$
	3	–	5	2	$w[3] = 5 > 3$: saltare
	4	–	5	2	$w[4] = 6 > 3$: saltare
	–	2	5	2	$v[3] \leftarrow m = 5$, $s[3] \leftarrow t = 2$
	–	–	0	0	$m := 0$, $t := 0$
4	1	–	8	1	$4{+}4 = 8 > m = 0$: $m \leftarrow 8$, $t \leftarrow 1$
	2	–	8	1	$5{+}0 = 5 \leq m = 8$: nessun aggiornamento
	3	–	8	1	$w[3] = 5 > 4$: saltare
	4	–	8	1	$w[4] = 6 > 4$: saltare
	–	1	8	1	$v[4] \leftarrow m = 8$, $s[4] \leftarrow t = 1$
	–	–	0	0	$m := 0$, $t := 0$
5	1	–	9	1	$4{+}5 = 9 > m = 0$: $m \leftarrow 9$, $t \leftarrow 1$
	2	–	9	1	$5{+}4 = 9 = m$: nessun aggiornamento
	3	–	10	3	$10{+}0 = 10 > m = 9$: $m \leftarrow 10$, $t \leftarrow 3$
	4	–	10	3	$w[4] = 6 > 5$: saltare
	–	3	10	3	$v[5] \leftarrow m = 10$, $s[5] \leftarrow t = 3$

k	j	$s[k]$	m	t	Azione
	–	–	0	0	$m := 0$, $t := 0$
6	1	–	12	1	$4{+}8 = 12 > m = 0$: $m \leftarrow 12$, $t \leftarrow 1$
	2	–	12	1	$5{+}5 = 10 \leq m = 12$: nessun aggiornamento
	3	–	12	1	$10{+}0 = 10 \leq m = 12$: nessun aggiornamento
	4	–	12	1	$11{+}0 = 11 \leq m = 12$: nessun aggiornamento
	–	1	12	1	$v[6] \leftarrow m = 12$, $s[6] \leftarrow t = 1$
	–	–	0	0	$m := 0$, $t := 0$
7	1	–	14	1	$4{+}10 = 14 > m = 0$: $m \leftarrow 14$, $t \leftarrow 1$
	2	–	14	1	$5{+}8 = 13 \leq m = 14$: nessun aggiornamento
	3	–	14	1	$10{+}4 = 14 = m$: nessun aggiornamento
	4	–	14	1	$11{+}0 = 11 \leq m = 14$: nessun aggiornamento
	–	1	14	1	$v[7] \leftarrow m = 14$, $s[7] \leftarrow t = 1$

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

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$s[k]$	0	0	1	2	1	3	1	1

– Calcolato il vettore s, si può calcolare una soluzione ottima dell'istanza dell'UKP applicando ripetutamente la seguente regola, partendo da $k = c$ e da $x_j = 0$ per ogni $j \in \{1, 2, \ldots, n\}$:

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

finché $k = 0$ o $s[k] = 0$. Il secondo caso si verifica quando nessun tipo di oggetto entra nella capacità residua k, che resta quindi inutilizzata. Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, dove ogni componente x_j è il numero di copie del tipo di oggetto j in una soluzione ottima.

– Il seguente pseudocodice realizza il calcolo di una soluzione ottima, dato il vettore s.

Algoritmo

Algorithm: UKP_DP1_SolComputation

Input: Il numero n di tipi di oggetto, la capacità c, il vettore $w[1..n]$ e il vettore $s[0..c]$

Output: Una soluzione ottima dell'istanza dell'UKP

```
1  Function UKP_DP1_SolComputation(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 e s[k] > 0 do
8          x[s[k]] ← x[s[k]] + 1
9          k ← k − w[s[k]]
10     end
11     return x
```

Esempio 3.6: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 3.1, abbiamo il seguente vettore s:

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$s[k]$	0	0	1	2	1	3	1	1

Possiamo calcolare una soluzione ottima per uno zaino di capacità $c = 7$ come segue:

Iterazione	k	$s[k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$k \leftarrow k - w_{s[k]}$
–	–	–	0	0	0	0	$k = 7$
1	7	1	1	0	0	0	$k = 7 - 2 = 5$
2	5	3	1	0	1	0	$k = 5 - 5 = 0$

Una soluzione ottima è il vettore:

$$(1, 0, 1, 0)$$

Il suo peso totale è $w_1 + w_3 = 2 + 5 = 7 = c$ e il suo profitto totale è $p_1 + p_3 = 4 + 10 = 14$, che coincide con il valore ottimo $v(7) = 14$.

3.2.1 Numero di passi

- Indichiamo con $S(n, c)$ il numero di passi eseguiti da `UKP_DP1` nel caso peggiore, dove n è il numero di tipi di oggetto e c è la capacità dello zaino. Calcoliamo $S(n, c)$ contando quante volte vengono eseguite le righe dello pseudocodice di `UKP_DP1`.

- Nel caso peggiore, le condizioni delle righe 7 e 8 sono vere a ogni iterazione del ciclo interno, e quindi la riga 9 viene eseguita tante volte quanto la riga 8. Il numero di esecuzioni delle righe rilevanti è:

  - La Riga 2 viene eseguita 1 volta.
  - La Riga 3 viene eseguita 1 volta.
  - La Riga 4 viene eseguita c volte.
  - La Riga 5 viene eseguita c volte.
  - La Riga 6 viene eseguita cn volte.
  - La Riga 7 viene eseguita cn volte.
  - La Riga 8 viene eseguita cn volte.
  - La Riga 9 viene eseguita cn volte.
  - La Riga 13 viene eseguita c volte.
  - La Riga 15 viene eseguita 1 volta.

  Pertanto, il numero totale di passi nel caso peggiore è

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

  Quindi

  $$S(n, c) = 3 + 3c + 4cn.$$

  Dunque

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

  Infatti

  $$\lim_{n,c\to\infty} \frac{4cn + 3c + 3}{nc} = \lim_{n,c\to\infty} \left(4 + \frac{3}{n} + \frac{3}{nc}\right) = 4 \in \mathbb{R}_{>0}.$$

  L'algoritmo di PD esegue
  $$S(n, c) = \Theta(nc)$$
  passi nel caso peggiore.

- La versione `UKP_DP1_InfoSol` memorizza il vettore aggiuntivo s ed esegue solo un numero costante di operazioni in più dentro gli stessi cicli. Pertanto, il suo numero di passi resta $\Theta(nc)$.

3.3 Un secondo algoritmo di PD

- Passo 1: calcolo del valore ottimo

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

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

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

$$v(j,k) = \begin{cases} 0, & \text{se } j = 0, \\ v(j-1,k), & \text{se } j \geq 1 \text{ e } k < w_j, \\ \max\{v(j-1,k),\ p_j + v(j,k-w_j)\}, & \text{se } j \geq 1 \text{ e } k \geq w_j. \end{cases} \tag{3.2}$$

- Il processo di decomposizione ricorsiva della formula (3.2) genera al più 2 **sottoproblemi indipendenti**, corrispondenti alla scelta di inserire o no una copia del tipo di oggetto j in uno zaino di capacità k, usando solo i tipi di oggetto $1, 2, \ldots, j$.
- Se $w_j > k$, il tipo di oggetto j non può essere inserito e quindi l'unica opzione ammissibile è non usarlo.
- Il caso base è $v(0, k) = 0$ per ogni k, poiché senza tipi di oggetto disponibili l'unica scelta ammissibile è non inserire alcun oggetto, che ha sempre profitto 0.
- Se $j \geq 1$ e $k \geq w_j$, il valore ottimo $v(j, k)$ si ottiene provando le due opzioni possibili, non inserire il tipo di oggetto j oppure inserirne un'ulteriore copia, e scegliendo quella che massimizza il profitto. I valori corrispondenti sono, rispettivamente, $v(j-1, k)$ e $p_j + v(j, k-w_j)$.
- Se $j \geq 1$ e $k < w_j$, il tipo di oggetto j non può essere usato e quindi $v(j, k) = v(j-1, k)$.
- Il termine $p_j + v(j, k-w_j)$ si riferisce alla stessa riga j perché, dopo aver inserito una copia del tipo di oggetto j, sono ancora disponibili altre copie dello stesso tipo.
- Quindi una soluzione ottima per uno zaino di capacità k che usa solo i tipi di oggetto $1, 2, \ldots, j$ si costruisce combinando la decisione scelta sul tipo di oggetto j con una soluzione ottima del corrispondente sottoproblema residuo.

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

– Sequenze diverse di decisioni sui tipi di oggetto, 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 (3.2).

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

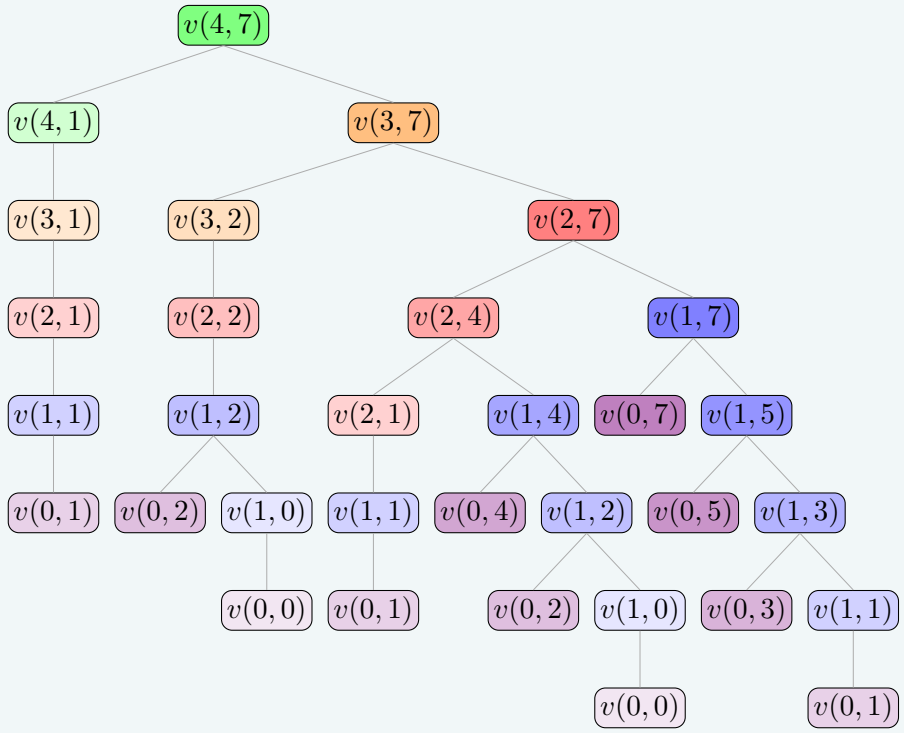
– Un **primo metodo** per calcolare il valore ottimo $v(n,c)$ dell'istanza dell'UKP è l'**approccio top-down**.

– Il metodo parte dalla chiamata con $(j,k) = (n,c)$ e valuta ricorsivamente la formula (3.2): ogni chiamata per la capacità k con i soli tipi di oggetto $1,2,\ldots,j$ genera altre chiamate, corrispondenti alle decisioni ammissibili sul tipo di oggetto j, finché si raggiunge uno dei casi base $v(0,k)$.

– 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 3.7: Albero di ricorsione

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con capacità $c = 7$ e tipi di oggetto di pesi $w_1 = 2$, $w_2 = 3$, $w_3 = 5$, $w_4 = 6$. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(4,7)$:



La funzione v viene chiamata 32 volte e gli stessi sottoproblemi vengono risolti più volte: $v(1,1)$ viene chiamata 3 volte, $v(0,1)$ 3 volte, $v(1,2)$ 2 volte, $v(0,2)$

2 volte, $v(1,0)$ 2 volte, $v(0,0)$ 2 volte e $v(2,1)$ 2 volte. Tutte le altre chiamate avvengono una sola volta.

– Il valore $v(j,k)$ si può calcolare quando è noto $v(j-1,k)$ e, se $k \geq w_j$, è noto anche $v(j,k-w_j)$. Pertanto, per un valore fissato di j, tutti i valori $v(j,k)$ si possono calcolare considerando i valori di k in ordine crescente.

– Partendo dal caso base $v(0,k) = 0$ per ogni k, i valori della funzione v si possono calcolare riga per riga, passando al valore successivo di j solo dopo aver calcolato tutti i valori $v(j,k)$ per il valore corrente di j.

Esempio 3.8: Calcolo dei valori della funzione v

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con uno zaino di capacità $c = 7$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	4	5	10	11
w_j	2	3	5	6

I valori della funzione v si calcolano come segue:

∗ Per $j = 1$ è disponibile solo il tipo di oggetto 1 ($w_1 = 2$). Pertanto:

$$v(1,0) = 0, \qquad v(1,1) = v(0,1) = 0,$$

$$v(1,2) = \max\{v(0,2), p_1 + v(1,0)\} = \max\{0, 4+0\} = 4,$$
$$v(1,3) = \max\{v(0,3), p_1 + v(1,1)\} = \max\{0, 4+0\} = 4,$$
$$v(1,4) = \max\{v(0,4), p_1 + v(1,2)\} = \max\{0, 4+4\} = 8,$$
$$v(1,5) = \max\{v(0,5), p_1 + v(1,3)\} = \max\{0, 4+4\} = 8,$$
$$v(1,6) = \max\{v(0,6), p_1 + v(1,4)\} = \max\{0, 4+8\} = 12,$$
$$v(1,7) = \max\{v(0,7), p_1 + v(1,5)\} = \max\{0, 4+8\} = 12.$$

∗ Per $j = 2$ sono disponibili i tipi di oggetto 1 e 2. Otteniamo:

$$v(2,0) = 0, \qquad v(2,1) = v(1,1) = 0, \qquad v(2,2) = v(1,2) = 4,$$

$$v(2,3) = \max\{v(1,3), p_2 + v(2,0)\} = \max\{4, 5+0\} = 5,$$
$$v(2,4) = \max\{v(1,4), p_2 + v(2,1)\} = \max\{8, 5+0\} = 8,$$
$$v(2,5) = \max\{v(1,5), p_2 + v(2,2)\} = \max\{8, 5+4\} = 9,$$
$$v(2,6) = \max\{v(1,6), p_2 + v(2,3)\} = \max\{12, 5+5\} = 12,$$
$$v(2,7) = \max\{v(1,7), p_2 + v(2,4)\} = \max\{12, 5+8\} = 13.$$

∗ Per $j = 3$ sono disponibili i tipi di oggetto 1, 2 e 3. Otteniamo:

$$v(3,0) = 0, \qquad v(3,1) = v(2,1) = 0, \qquad v(3,2) = v(2,2) = 4,$$

$$v(3,3) = v(2,3) = 5, \qquad v(3,4) = v(2,4) = 8,$$

$$v(3,5) = \max\{v(2,5), p_3 + v(3,0)\} = \max\{9, 10+0\} = 10,$$

$$v(3,6) = \max\{v(2,6), p_3 + v(3,1)\} = \max\{12, 10+0\} = 12,$$

$$v(3,7) = \max\{v(2,7), p_3 + v(3,2)\} = \max\{13, 10+4\} = 14.$$

∗ Per $j = 4$ sono disponibili tutti i tipi di oggetto. Otteniamo:

$$v(4,0) = 0, \quad v(4,1) = v(3,1) = 0, \quad v(4,2) = v(3,2) = 4,$$

$$v(4,3) = v(3,3) = 5, \quad v(4,4) = v(3,4) = 8, \quad v(4,5) = v(3,5) = 10,$$

$$v(4,6) = \max\{v(3,6), p_4 + v(4,0)\} = \max\{12, 11+0\} = 12,$$

$$v(4,7) = \max\{v(3,7), p_4 + v(4,1)\} = \max\{14, 11+0\} = 14.$$

– 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, oppure con lo stesso valore di j e una capacità più piccola, possiamo risolvere i sottoproblemi in ordine crescente di j e, per ogni valore fissato di j, in ordine crescente di capacità, partendo dai casi base $v(0,k) = 0$ per ogni $k \in \{0, 1, \ldots, c\}$ 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 dell'UKP.

– Un **secondo metodo** per calcolare il valore ottimo $v(n,c)$ dell'istanza dell'UKP è l'**approccio bottom-up**.

– Il metodo parte dai casi base $v(0,k) = 0$ per ogni $k \in \{0, 1, \ldots, c\}$ e valuta la formula (3.2) 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 dell'UKP.

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

Algoritmo

Algorithm: UKP_DP2

Input: Un'istanza dell'UKP data da n, dai vettori $p[1..n]$, $w[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza dell'UKP

```
1  Function UKP_DP2(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
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 p[j] + v[j][k − w[j]] > v[j − 1][k] then
12                     v[j][k] ← p[j] + v[j][k − w[j]]
13                 else
14                     v[j][k] ← v[j − 1][k]
15                 end
16             end
17         end
18     end
19     return v[n][c]
```

* La matrice $v[0..n][0..c]$ è la **lookup table**: memorizza i valori ottimi di tutti i sottoproblemi.
* L'inizializzazione pone $v[0][k] = 0$ per ogni capacità $k \in \{0, 1, \ldots, c\}$, poiché non selezionare alcun oggetto è sempre ammissibile, con profitto 0.
* Il ciclo esterno, su $j \in \{1, 2, \ldots, n\}$, considera i tipi di oggetto disponibili in ordine crescente.
* Il ciclo interno, su $k \in \{0, 1, \ldots, c\}$, calcola il valore migliore per ogni capacità usando solo i tipi di oggetto $1, 2, \ldots, j$.
* Se $k < w[j]$, il tipo di oggetto j non può essere usato e quindi $v[j][k] \leftarrow v[j-1][k]$.
* Altrimenti, l'algoritmo di PD confronta il valore ottenuto senza usare il tipo di oggetto j, cioè $v[j-1][k]$, con il valore ottenuto inserendo un'ulteriore copia del tipo di oggetto j, cioè $p[j] + v[j][k - w[j]]$.
* Infine, l'algoritmo restituisce il valore ottimo dell'istanza dell'UKP, $v[n][c]$.

Esempio 3.9: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con uno zaino di capacità $c = 7$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	4	5	10	11
w_j	2	3	5	6

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	$v[0][5] \leftarrow 0$
0	6	0	$v[0][6] \leftarrow 0$
0	7	0	$v[0][7] \leftarrow 0$
1	0	0	$k = 0 < w[1] = 2$: $v[1][0] \leftarrow v[0][0] = 0$
1	1	0	$k = 1 < w[1] = 2$: $v[1][1] \leftarrow v[0][1] = 0$
1	2	4	$\max\{v[0][2], p[1] + v[1][0]\} = \max\{0, 4+0\} = 4$
1	3	4	$\max\{v[0][3], p[1] + v[1][1]\} = \max\{0, 4+0\} = 4$
1	4	8	$\max\{v[0][4], p[1] + v[1][2]\} = \max\{0, 4+4\} = 8$
1	5	8	$\max\{v[0][5], p[1] + v[1][3]\} = \max\{0, 4+4\} = 8$
1	6	12	$\max\{v[0][6], p[1] + v[1][4]\} = \max\{0, 4+8\} = 12$
1	7	12	$\max\{v[0][7], p[1] + v[1][5]\} = \max\{0, 4+8\} = 12$
2	0	0	$k = 0 < w[2] = 3$: $v[2][0] \leftarrow v[1][0] = 0$
2	1	0	$k = 1 < w[2] = 3$: $v[2][1] \leftarrow v[1][1] = 0$
2	2	4	$k = 2 < w[2] = 3$: $v[2][2] \leftarrow v[1][2] = 4$
2	3	5	$\max\{v[1][3], p[2] + v[2][0]\} = \max\{4, 5+0\} = 5$
2	4	8	$\max\{v[1][4], p[2] + v[2][1]\} = \max\{8, 5+0\} = 8$
2	5	9	$\max\{v[1][5], p[2] + v[2][2]\} = \max\{8, 5+4\} = 9$
2	6	12	$\max\{v[1][6], p[2] + v[2][3]\} = \max\{12, 5+5\} = 12$
2	7	13	$\max\{v[1][7], p[2] + v[2][4]\} = \max\{12, 5+8\} = 13$

j	k	$v[j][k]$	Azione
3	0	0	$k = 0 < w[3] = 5$: $v[3][0] \leftarrow v[2][0] = 0$
3	1	0	$k = 1 < w[3] = 5$: $v[3][1] \leftarrow v[2][1] = 0$
3	2	4	$k = 2 < w[3] = 5$: $v[3][2] \leftarrow v[2][2] = 4$
3	3	5	$k = 3 < w[3] = 5$: $v[3][3] \leftarrow v[2][3] = 5$
3	4	8	$k = 4 < w[3] = 5$: $v[3][4] \leftarrow v[2][4] = 8$
3	5	10	$\max\{v[2][5], p[3] + v[3][0]\} = \max\{9, 10 + 0\} = 10$
3	6	12	$\max\{v[2][6], p[3] + v[3][1]\} = \max\{12, 10 + 0\} = 12$
3	7	14	$\max\{v[2][7], p[3] + v[3][2]\} = \max\{13, 10 + 4\} = 14$
4	0	0	$k = 0 < w[4] = 6$: $v[4][0] \leftarrow v[3][0] = 0$
4	1	0	$k = 1 < w[4] = 6$: $v[4][1] \leftarrow v[3][1] = 0$
4	2	4	$k = 2 < w[4] = 6$: $v[4][2] \leftarrow v[3][2] = 4$
4	3	5	$k = 3 < w[4] = 6$: $v[4][3] \leftarrow v[3][3] = 5$
4	4	8	$k = 4 < w[4] = 6$: $v[4][4] \leftarrow v[3][4] = 8$
4	5	10	$k = 5 < w[4] = 6$: $v[4][5] \leftarrow v[3][5] = 10$
4	6	12	$\max\{v[3][6], p[4] + v[4][0]\} = \max\{12, 11 + 0\} = 12$
4	7	14	$\max\{v[3][7], p[4] + v[4][1]\} = \max\{14, 11 + 0\} = 14$

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

$v[j][k]$	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
$j = 0$	0	0	0	0	0	0	0	0
$j = 1$	0	0	4	4	8	8	12	12
$j = 2$	0	0	4	5	8	9	12	13
$j = 3$	0	0	4	5	8	10	12	14
$j = 4$	0	0	4	5	8	10	12	14

  - Dopo l'esecuzione dell'algoritmo di PD, la matrice v contiene il valore ottimo $v[j][k]$ per ogni sottoproblema (j, k).

- Passo 2: calcolo di una soluzione ottima
  - Per calcolare una soluzione ottima è necessario memorizzare alcune informazioni aggiuntive.
  - Per ogni tipo di oggetto j e ogni capacità k, oltre al valore ottimo $v[j][k]$, memorizziamo $s[j][k]$, che è uguale a 1 se si usa una copia del tipo di oggetto j nel calcolo di $v[j][k]$, e a 0 altrimenti.
  - Partendo da $j = n$ e $k = c$, se $s[j][k] = 0$ il tipo di oggetto j non viene usato e j diminuisce di 1. Se $s[j][k] = 1$, si usa una copia del tipo di oggetto j e la capacità residua diventa $k - w_j$. La procedura si ferma quando la capacità residua arriva a 0 o quando $j = 0$; in questo secondo caso la capacità residua resta inutilizzata.

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

Algoritmo

Algorithm: UKP_DP2_InfoSol

Input: Un'istanza dell'UKP data da n, dai vettori $p[1..n]$, $w[1..n]$ e dalla capacità c

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

```
1  Function UKP_DP2_InfoSol(n, p, w, c)
2      v := new array[0..n][0..c]; s := new array[0..n][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
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]; s[j][k] ← 0
10             else
11                 if p[j] + v[j][k − w[j]] > v[j − 1][k] then
12                     v[j][k] ← p[j] + v[j][k − w[j]]; s[j][k] ← 1
13                 else
14                     v[j][k] ← v[j − 1][k]; s[j][k] ← 0
15                 end
16             end
17         end
18     end
19     return v[n][c], s
```

∗ L'algoritmo di PD usa la matrice aggiuntiva s, che registra la decisione presa quando si calcola $v[j][k]$. Se viene selezionata una copia del tipo di oggetto j, $s[j][k] \leftarrow 1$ memorizza questa decisione; altrimenti, $s[j][k] \leftarrow 0$ memorizza che il tipo di oggetto j non viene usato.

Esempio 3.10: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza dell'UKP dell'Esempio 3.1, con uno zaino di capacità $c = 7$ e i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	4	5	10	11
w_j	2	3	5	6

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	$s[0][5] \leftarrow 0$
0	6	0	$s[0][6] \leftarrow 0$
0	7	0	$s[0][7] \leftarrow 0$
1	0	0	$k < w[1]$: $s[1][0] \leftarrow 0$
1	1	0	$k < w[1]$: $s[1][1] \leftarrow 0$
1	2	1	$p[1] + v[1][0] = 4 + 0 > v[0][2] = 0$: $s[1][2] \leftarrow 1$
1	3	1	$p[1] + v[1][1] = 4 + 0 > v[0][3] = 0$: $s[1][3] \leftarrow 1$
1	4	1	$p[1] + v[1][2] = 4 + 4 > v[0][4] = 0$: $s[1][4] \leftarrow 1$
1	5	1	$p[1] + v[1][3] = 4 + 4 > v[0][5] = 0$: $s[1][5] \leftarrow 1$
1	6	1	$p[1] + v[1][4] = 4 + 8 > v[0][6] = 0$: $s[1][6] \leftarrow 1$
1	7	1	$p[1] + v[1][5] = 4 + 8 > v[0][7] = 0$: $s[1][7] \leftarrow 1$
2	0	0	$k < w[2]$: $s[2][0] \leftarrow 0$
2	1	0	$k < w[2]$: $s[2][1] \leftarrow 0$
2	2	0	$k < w[2]$: $s[2][2] \leftarrow 0$
2	3	1	$p[2] + v[2][0] = 5 + 0 > v[1][3] = 4$: $s[2][3] \leftarrow 1$
2	4	0	$p[2] + v[2][1] = 5 + 0 \leq v[1][4] = 8$: $s[2][4] \leftarrow 0$
2	5	1	$p[2] + v[2][2] = 5 + 4 > v[1][5] = 8$: $s[2][5] \leftarrow 1$
2	6	0	$p[2] + v[2][3] = 5 + 5 \leq v[1][6] = 12$: $s[2][6] \leftarrow 0$
2	7	1	$p[2] + v[2][4] = 5 + 8 > v[1][7] = 12$: $s[2][7] \leftarrow 1$

j	k	$s[j][k]$	Azione
3	0	0	$k < w[3]$: $s[3][0] \leftarrow 0$
3	1	0	$k < w[3]$: $s[3][1] \leftarrow 0$
3	2	0	$k < w[3]$: $s[3][2] \leftarrow 0$
3	3	0	$k < w[3]$: $s[3][3] \leftarrow 0$
3	4	0	$k < w[3]$: $s[3][4] \leftarrow 0$
3	5	1	$p[3] + v[3][0] = 10 + 0 > v[2][5] = 9$: $s[3][5] \leftarrow 1$
3	6	0	$p[3] + v[3][1] = 10 + 0 \leq v[2][6] = 12$: $s[3][6] \leftarrow 0$
3	7	1	$p[3] + v[3][2] = 10 + 4 > v[2][7] = 13$: $s[3][7] \leftarrow 1$
4	0	0	$k < w[4]$: $s[4][0] \leftarrow 0$
4	1	0	$k < w[4]$: $s[4][1] \leftarrow 0$
4	2	0	$k < w[4]$: $s[4][2] \leftarrow 0$
4	3	0	$k < w[4]$: $s[4][3] \leftarrow 0$
4	4	0	$k < w[4]$: $s[4][4] \leftarrow 0$
4	5	0	$k < w[4]$: $s[4][5] \leftarrow 0$
4	6	0	$p[4] + v[4][0] = 11 + 0 \leq v[3][6] = 12$: $s[4][6] \leftarrow 0$
4	7	0	$p[4] + v[4][1] = 11 + 0 \leq v[3][7] = 14$: $s[4][7] \leftarrow 0$

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

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$j=0$	0	0	0	0	0	0	0	0
$j=1$	0	0	1	1	1	1	1	1
$j=2$	0	0	0	1	0	1	0	1
$j=3$	0	0	0	0	0	1	0	1
$j=4$	0	0	0	0	0	0	0	0

– Calcolata la matrice s, si può calcolare una soluzione ottima per uno zaino di capacità c applicando ripetutamente la seguente regola, partendo da $(j,k)=(n,c)$ e da $x_h=0$ per ogni $h \in \{1,2,\ldots,n\}$:

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

La procedura si ferma quando $k=0$ o $j=0$; il secondo caso si verifica quando la capacità residua resta inutilizzata. Si ottiene così il vettore $(x_1,x_2,\ldots,x_n)$, dove ogni componente x_h è il numero di copie del tipo di oggetto h in una soluzione ottima.

– Il seguente pseudocodice realizza il calcolo di una soluzione ottima, data la matrice s.

Algoritmo

Algorithm: UKP_DP2_SolComputation

Input: Il numero di tipi di oggetto n, il vettore $w[1..n]$, la capacità c e la matrice $s[0..n][0..c]$

Output: Una soluzione ottima dell'istanza dell'UKP

```
1  Function UKP_DP2_SolComputation(n, w, c, s)
2      x := new array[1..n]
3      for h := 1 to n do
4          x[h] ← 0
5      end
6      j := n
7      k := c
8      while k > 0 e j > 0 do
9          if s[j][k] = 0 then
10             j ← j − 1
11         else
12             x[j] ← x[j] + 1
13             k ← k − w[j]
14         end
15     end
16     return x
```

Esempio 3.11: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 3.1, abbiamo la seguente matrice s:

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$j=0$	0	0	0	0	0	0	0	0
$j=1$	0	0	1	1	1	1	1	1
$j=2$	0	0	0	1	0	1	0	1
$j=3$	0	0	0	0	0	1	0	1
$j=4$	0	0	0	0	0	0	0	0

Possiamo calcolare una soluzione ottima per uno zaino di capacità $c=7$ come segue:

Iterazione	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Azione
1	4	7	0	0	0	0	0	$j \leftarrow 3$
2	3	7	1	0	0	1	0	$x[3] \leftarrow 1$, $k \leftarrow 7-5=2$
3	3	2	0	0	0	1	0	$j \leftarrow 2$
4	2	2	0	0	0	1	0	$j \leftarrow 1$
5	1	2	1	1	0	1	0	$x[1] \leftarrow 1$, $k \leftarrow 2-2=0$

Una soluzione ottima è il vettore:

$$(1,0,1,0)$$

Il suo peso totale è $w_1+w_3=2+5=7=c$ e il suo profitto totale è $p_1+p_3=4+10=14$, che coincide con il valore ottimo $v(4,7)=14$.

3.3.1 Numero di passi

- Indichiamo con $S(n,c)$ il numero di passi eseguiti da `UKP_DP2` nel caso peggiore, dove n è il numero di tipi di oggetto e c è la capacità dello zaino. Calcoliamo $S(n,c)$ contando quante volte vengono eseguite le righe dello pseudocodice di `UKP_DP2`.

- Nel caso peggiore, la riga 11 viene verificata ogni volta che il tipo di oggetto entra nello zaino. Poiché i pesi degli oggetti sono interi positivi, questo accade al più nc volte. 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.

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

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

L'algoritmo di PD esegue

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

passi nel caso peggiore.

- La versione `UKP_DP2_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 $\Theta(nc)$.

3.4 Errori tipici

Attenzione

- **Usare la riga $j-1$ nella seconda ricorrenza.** Il termine $p_j + v(j, k - w_j)$ si riferisce alla stessa riga j: è questo che consente più copie del tipo di oggetto j.
- **Supporre che la capacità sia riempita.** Una parte della capacità può restare inutilizzata: quando nessun tipo di oggetto entra, $v(k) = 0$, non $-\infty$; nell'Esempio 3.1, $v(1) = 0$.
- **Ricostruire oltre la fine.** La ricostruzione si ferma quando la capacità residua è 0, oppure quando $s[k] = 0$ nel primo algoritmo di PD e $j = 0$ nel secondo; altrimenti si legge fuori dai vettori.
- **Dimenticare il caso base.** $v(0) = 0$, e $v[0..c]$ ha $c+1$ celle, non c.

3.5 In sintesi

In sintesi

- Nell'UKP un tipo di oggetto si può usare un numero qualsiasi di volte, quindi il valore ottimo dipende solo dalla capacità residua: $v(k) = \max_{j:\, w_j \le k}\{p_j + v(k - w_j)\}$.
- Il primo algoritmo di PD riempie un vettore di $c+1$ celle in $\Theta(nc)$ passi; il secondo riempie la matrice $v(j,k)$ e riusa la stessa riga per le copie successive.
- Una soluzione ottima si legge all'indietro dalle informazioni memorizzate, fermandosi quando la capacità residua è nulla o inutilizzata.
- Il tempo di esecuzione è pseudopolinomiale, come per il KP.

Cruscotto dell'esempio dell'UKP. L'Esempio 3.1 ($c = 7$, quattro tipi di oggetto):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	14	14	valore ottimo, soluzione $(1, 0, 1, 0)$

Capitolo 4

PD per il problema dello zaino limitato

Prerequisiti

- Da questo modulo, i capitoli «PD per il problema dello zaino 0/1» e «PD per il problema dello zaino illimitato»: la ricorrenza $v(j, k)$ del problema dello zaino 0/1 (KP) e del problema dello zaino illimitato (UKP) su oggetti e capacità, e la ricostruzione di una soluzione.
- Da questo modulo, il capitolo «Principi della programmazione dinamica»: sottostruttura ottima, sottoproblemi sovrapposti, casi base.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- spiegare perché il KP ($b_j = 1$) e l'UKP ($b_j \geq \lfloor c/w_j \rfloor$) sono casi particolari del BKP, e scriverne la ricorrenza;
- eseguire a mano l'algoritmo di PD, provando ogni numero ammissibile d di copie di ciascun tipo di oggetto;
- riempire la matrice s dei numeri di copie e ricostruire da essa una soluzione ottima;
- contare i passi dell'algoritmo di PD e dire quando $\Theta(nc\,b_{\max})$ è stretto.

Il problema di questo capitolo è il problema dello zaino limitato (BKP): la sua definizione e le sue ipotesi sono nella sua pagina.

4.1 Istanze e soluzioni ammissibili

Esempio 4.1: Istanza del BKP

Consideriamo uno zaino di capacità $c = 10$ e 3 tipi di oggetto, con i seguenti profitti, pesi e limiti:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

Chiamiamo questa istanza del BKP Esempio 4.1.

La differenza principale rispetto al problema dello zaino 0/1 (KP) è che ogni tipo di oggetto può essere selezionato più di una volta: la variabile x_j non è più binaria, ma è limitata da b_j. Il BKP si riduce al KP quando $b_j = 1$ per ogni tipo di oggetto j, e al problema dello zaino illimitato (UKP) quando ogni limite b_j è abbastanza grande da non essere vincolante.

- Il numero di vettori possibili $(x_1, x_2, \ldots, x_n)$, con $x_j \in \{0, 1, \ldots, b_j\}$ per ogni tipo di oggetto j, è
$$\prod_{j=1}^{n} (b_j + 1).$$
Alcuni di questi vettori possono non rispettare la capacità dello zaino. Quindi il numero di soluzioni ammissibili è al più $\prod_{j=1}^{n}(b_j + 1)$, una quantità che cresce esponenzialmente con n. Enumerare tutte le soluzioni possibili diventa rapidamente impraticabile al crescere di n.

4.1.1 Codifica delle soluzioni ammissibili

Una *soluzione ammissibile* del BKP è una scelta del numero di copie di ciascun tipo di oggetto che rispetta sia i limiti b_j sia la capacità c dello zaino.

- Una soluzione ammissibile del BKP si può descrivere con il vettore intero
$$(x_1, x_2, \ldots, x_n)$$
dove ogni componente ha il seguente significato:
$$x_j = \text{numero di copie selezionate del tipo di oggetto } j,$$
$$\forall j \in \{1, 2, \ldots, n\}, \quad 0 \leq x_j \leq b_j.$$

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

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

4.2 Un algoritmo di PD

- Passo 1: calcolo del valore ottimo

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 BKP data da:

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

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

$$v(j,k) = \begin{cases} 0 & \text{se } j = 0 \text{ o } k = 0, \\ \max\limits_{d \in \{0,1,\ldots,\min\{b_j, \lfloor k/w_j \rfloor\}\}} \left\{ d\,p_j + v(j-1, k - d\,w_j) \right\} & \text{se } 1 \leq j \leq n \\ & \text{e } 1 \leq k \leq c. \end{cases} \tag{4.1}$$

- Il processo di decomposizione ricorsiva della formula (4.1) genera al più $b_j + 1$ **sottoproblemi indipendenti**, corrispondenti ai possibili numeri $d \in \{0, 1, \ldots, \min\{b_j, \lfloor k/w_j \rfloor\}\}$ di copie del tipo di oggetto j inserite in uno zaino di capacità k, usando solo i tipi di oggetto $1, 2, \ldots, j$.
- I casi base sono $v(0, k) = 0$, perché nessun tipo di oggetto è disponibile, e $v(j, 0) = 0$, perché non c'è capacità disponibile.
- Il valore ottimo $v(j, k)$ si ottiene provando ogni numero ammissibile d di copie del tipo di oggetto j e scegliendo quello che massimizza la somma $d\,p_j + v(j-1, k-d\,w_j)$ del profitto delle d copie e del valore ottimo del sottoproblema residuo, ristretto ai tipi di oggetto $1, 2, \ldots, j-1$.

– Quindi una soluzione ottima della sottoistanza con capacità k e tipi di oggetto disponibili $\{1, 2, \ldots, j\}$ si costruisce combinando il numero d scelto di copie del tipo di oggetto j con una soluzione ottima del corrispondente sottoproblema residuo.

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

– Scelte diverse del numero di copie di un tipo di oggetto, fatte eventualmente per valori diversi di j e k, possono portare allo stesso sottoproblema, e quindi lo stesso valore $v(j, k)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (4.1).

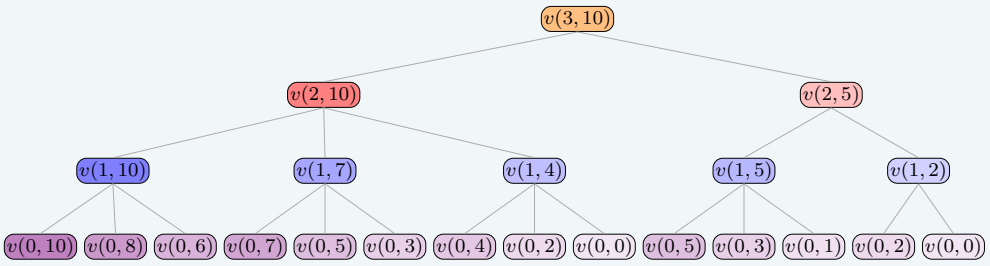
$\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

– Un **primo metodo** per calcolare il valore ottimo $v(n, c)$ dell'istanza del BKP è l'**approccio top-down**. Il metodo parte dalla chiamata con $j = n$ e $k = c$ e valuta ricorsivamente la formula (4.1): ogni chiamata considera il tipo di oggetto j e ne prova ogni numero ammissibile d di copie, finché non resta alcun tipo di oggetto o 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 effettuate da quella chiamata. Le foglie rappresentano i casi base.

Esempio 4.2: Albero di ricorsione

Consideriamo l'istanza del BKP dell'Esempio 4.1. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(3, 10)$:



La funzione v viene chiamata 22 volte, e gli stessi sottoproblemi vengono risolti più volte: $v(0, 5)$ viene chiamata 2 volte, $v(0, 3)$ viene chiamata 2 volte, $v(0, 2)$ viene chiamata 2 volte e $v(0, 0)$ viene chiamata 2 volte. Tutte le altre chiamate vengono effettuate una sola volta.

– Il valore $v(j, k)$ si può calcolare una volta noti i valori $v(j-1, k')$ per ogni capacità residua $k' \leq k$. Quindi, fissato j, tutti i valori $v(j, k)$ si possono calcolare considerando i valori di capacità k in ordine crescente.

Esempio 4.3: Calcolo dei valori della funzione v

Consideriamo l'istanza del BKP dell'Esempio 4.1, con uno zaino di capacità $c = 10$ e i seguenti profitti, pesi e limiti:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

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 $v(1, 4)$, provando $d = 0, 1, 2$ copie del tipo di oggetto 1 (perché $b_1 = 2$ e $\lfloor 4/2 \rfloor = 2$):

$$v(1,4) = \max\{v(0,4),\ p_1 + v(0,2),\ 2p_1 + v(0,0)\} = \max\{0,\ 6,\ 12\} = 12$$

* Possiamo calcolare anche $v(1, 10)$, provando $d = 0, 1, 2$ copie (perché $b_1 = 2$ limita il numero di copie anche se $\lfloor 10/2 \rfloor = 5$):

$$v(1,10) = \max\{v(0,10),\ p_1 + v(0,8),\ 2p_1 + v(0,6)\} = \max\{0,\ 6,\ 12\} = 12$$

* Allo stesso modo calcoliamo $v(1, 7)$, $v(1, 5)$ e $v(1, 2)$:

$$v(1,7) = \max\{v(0,7),\ p_1 + v(0,5),\ 2p_1 + v(0,3)\} = \max\{0,\ 6,\ 12\} = 12$$

$$v(1,5) = \max\{v(0,5),\ p_1 + v(0,3),\ 2p_1 + v(0,1)\} = \max\{0,\ 6,\ 12\} = 12$$

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

* Calcolati $v(1, 5)$ e $v(1, 2)$, possiamo calcolare $v(2, 5)$, provando $d = 0, 1$ copie del tipo di oggetto 2 (perché $b_2 = 2$ ma $\lfloor 5/3 \rfloor = 1$):

$$v(2,5) = \max\{v(1,5),\ p_2 + v(1,2)\} = \max\{12,\ 10 + 6\} = 16$$

* Calcolati $v(1, 10)$, $v(1, 7)$ e $v(1, 4)$, possiamo calcolare $v(2, 10)$, provando $d = 0, 1, 2$ copie del tipo di oggetto 2:

$$\begin{aligned} v(2,10) &= \max\{v(1,10),\ p_2 + v(1,7),\ 2p_2 + v(1,4)\} \\ &= \max\{12,\ 10 + 12,\ 20 + 12\} = 32 \end{aligned}$$

* Infine, calcolati $v(2, 10)$ e $v(2, 5)$, possiamo calcolare $v(3, 10)$, provando $d = 0, 1$ copie del tipo di oggetto 3 (perché $b_3 = 1$):

$$v(3,10) = \max\{v(2,10),\ p_3 + v(2,5)\} = \max\{32,\ 12 + 16\} = 32$$

– Possiamo evitare di risolvere ripetutamente gli stessi sottoproblemi memorizzando il

valore di ciascun sottoproblema appena viene calcolato. I valori sono memorizzati in una **tabella di lookup**.

– Poiché la soluzione di ogni sottoproblema dipende solo da sottoproblemi con un valore di j minore, possiamo risolvere i sottoproblemi in ordine crescente di j e, per ogni valore fissato di j, in ordine crescente di capacità, partendo 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 BKP.

– Un **secondo metodo** per calcolare il valore ottimo $v(n,c)$ dell'istanza del BKP è l'**approccio bottom-up**. Il metodo parte dai casi base e valuta la formula (4.1) per valori crescenti di $j \in \{1,2,\ldots,n\}$ e $k \in \{0,1,\ldots,c\}$: ogni valore $v(j,k)$ si calcola usando valori già calcolati, fino a ottenere il valore $v(n,c)$ dell'istanza del BKP.

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

Algoritmo

Algorithm: BKP_DP

Input: Un'istanza del BKP data da n, dai vettori $p[1..n]$, $w[1..n]$, $b[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza del BKP

```
1  Function BKP_DP(n, p, w, b, c)
2      v := new array[0..n][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0
5      end
6      for j := 1 to n do
7          for k := 0 to c do
8              m := 0
9              for d := 0 to min{b[j], ⌊k/w[j]⌋} do
10                 if d p[j] + v[j − 1][k − d w[j]] > m then
11                     m ← d p[j] + v[j − 1][k − d w[j]]
12                 end
13             end
14             v[j][k] ← m
15         end
16     end
17     return v[n][c]
```

∗ La matrice $v[0..n][0..c]$ è la **tabella di lookup**: 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 tipo di oggetto è disponibile.
* Il ciclo esterno, su $j \in \{1, 2, \ldots, n\}$, calcola le righe della matrice in ordine crescente di tipo di oggetto.
* Il ciclo interno, su $k \in \{0, 1, \ldots, c\}$, calcola il valore ottimo $v[j][k]$ per ogni capacità k.
* Il ciclo più interno, su $d \in \{0, 1, \ldots, \min\{b[j], \lfloor k/w[j] \rfloor\}\}$, prova ogni numero ammissibile di copie del tipo di oggetto j e conserva in m il valore massimo trovato finora.
* Al termine del ciclo più interno, $v[j][k] \leftarrow m$ memorizza il valore ottimo $v[j][k]$.
* Infine, il valore ottimo dell'istanza del BKP viene restituito come $v[n][c]$.

Esempio 4.4: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del BKP dell'Esempio 4.1, con uno zaino di capacità $c = 10$ e i seguenti profitti, pesi e limiti:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

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	$v[0][5] \leftarrow 0$
0	6	0	$v[0][6] \leftarrow 0$
0	7	0	$v[0][7] \leftarrow 0$
0	8	0	$v[0][8] \leftarrow 0$
0	9	0	$v[0][9] \leftarrow 0$
0	10	0	$v[0][10] \leftarrow 0$
1	0	0	$\max\{v[0][0]\} = 0$
1	1	0	$\max\{v[0][1]\} = 0$
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	12	$\max\{v[0][4],\ p[1]+v[0][2],\ 2p[1]+v[0][0]\} = \max\{0, 6, 12\} = 12$
1	5	12	$\max\{v[0][5],\ p[1]+v[0][3],\ 2p[1]+v[0][1]\} = \max\{0, 6, 12\} = 12$
1	6	12	$\max\{v[0][6],\ p[1]+v[0][4],\ 2p[1]+v[0][2]\} = \max\{0, 6, 12\} = 12$
1	7	12	$\max\{v[0][7],\ p[1]+v[0][5],\ 2p[1]+v[0][3]\} = \max\{0, 6, 12\} = 12$
1	8	12	$\max\{v[0][8],\ p[1]+v[0][6],\ 2p[1]+v[0][4]\} = \max\{0, 6, 12\} = 12$
1	9	12	$\max\{v[0][9],\ p[1]+v[0][7],\ 2p[1]+v[0][5]\} = \max\{0, 6, 12\} = 12$
1	10	12	$\max\{v[0][10],\ p[1]+v[0][8],\ 2p[1]+v[0][6]\} = \max\{0, 6, 12\} = 12$

j	k	$v[j][k]$	Azione
2	0	0	$\max\{v[1][0]\} = 0$
2	1	0	$\max\{v[1][1]\} = 0$
2	2	6	$\max\{v[1][2]\} = 6$
2	3	10	$\max\{v[1][3], p[2]+v[1][0]\} = \max\{6, 10+0\} = 10$
2	4	12	$\max\{v[1][4], p[2]+v[1][1]\} = \max\{12, 10+0\} = 12$
2	5	16	$\max\{v[1][5], p[2]+v[1][2]\} = \max\{12, 10+6\} = 16$
2	6	20	$\max\{v[1][6], p[2]+v[1][3], 2p[2]+v[1][0]\} = \max\{12, 16, 20\} = 20$
2	7	22	$\max\{v[1][7], p[2]+v[1][4], 2p[2]+v[1][1]\} = \max\{12, 22, 20\} = 22$
2	8	26	$\max\{v[1][8], p[2]+v[1][5], 2p[2]+v[1][2]\} = \max\{12, 22, 26\} = 26$
2	9	26	$\max\{v[1][9], p[2]+v[1][6], 2p[2]+v[1][3]\} = \max\{12, 22, 26\} = 26$
2	10	32	$\max\{v[1][10], p[2]+v[1][7], 2p[2]+v[1][4]\} = \max\{12, 22, 32\} = 32$
3	0	0	$\max\{v[2][0]\} = 0$
3	1	0	$\max\{v[2][1]\} = 0$
3	2	6	$\max\{v[2][2]\} = 6$
3	3	10	$\max\{v[2][3]\} = 10$
3	4	12	$\max\{v[2][4]\} = 12$
3	5	16	$\max\{v[2][5], p[3]+v[2][0]\} = \max\{16, 12+0\} = 16$
3	6	20	$\max\{v[2][6], p[3]+v[2][1]\} = \max\{20, 12+0\} = 20$
3	7	22	$\max\{v[2][7], p[3]+v[2][2]\} = \max\{22, 12+6\} = 22$
3	8	26	$\max\{v[2][8], p[3]+v[2][3]\} = \max\{26, 12+10\} = 26$
3	9	26	$\max\{v[2][9], p[3]+v[2][4]\} = \max\{26, 12+12\} = 26$
3	10	32	$\max\{v[2][10], p[3]+v[2][5]\} = \max\{32, 12+16\} = 32$

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

$v[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	6	6	12	12	12	12	12	12	12
$j=2$	0	0	6	10	12	16	20	22	26	26	32
$j=3$	0	0	6	10	12	16	20	22	26	26	32

  - Dopo l'esecuzione dell'algoritmo di PD, la matrice v contiene il valore ottimo $v[j][k]$ per ogni tipo di oggetto $j \in \{0, 1, \ldots, n\}$ e ogni valore di capacità $k \in \{0, 1, \ldots, c\}$.

- 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 tipo di oggetto j e valore di capacità k, oltre al valore ottimo $v[j][k]$, memorizziamo $s[j][k]$, il numero di copie del tipo di oggetto j selezionate in una soluzione ottima del sottoproblema ristretto ai tipi di oggetto $\{1, 2, \ldots, j\}$ e alla capacità k.

– Più precisamente, $s[j][k]$ è uguale al valore d che massimizza il membro destro della formula (4.1) nel calcolo di $v[j][k]$.

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

Algoritmo

Algorithm: BKP_DP_InfoSol

Input: Un'istanza del BKP data da n, dai vettori $p[1..n]$, $w[1..n]$, $b[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza del BKP e la matrice dei numeri di copie $s[0..n][0..c]$

```
1  Function BKP_DP_InfoSol(n, p, w, b, c)
2      v := new array[0..n][0..c]; s := new array[0..n][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
6      for j := 1 to n do
7          for k := 0 to c do
8              m := 0; t := 0
9              for d := 0 to min{b[j], ⌊k/w[j]⌋} do
10                 if d p[j] + v[j − 1][k − d w[j]] > m then
11                     m ← d p[j] + v[j − 1][k − d w[j]]; t ← d
12                 end
13             end
14             v[j][k] ← m; s[j][k] ← t
15         end
16     end
17     return v[n][c], s
```

∗ L'algoritmo di PD usa la variabile temporanea aggiuntiva t per tenere traccia della decisione presa quando m viene aggiornato. Se vengono selezionate d copie del tipo di oggetto j, $t \leftarrow d$ memorizza questa decisione. Al termine del ciclo più interno, $s[j][k] \leftarrow t$ memorizza il numero di copie del tipo di oggetto j usate, insieme al valore ottimo $v[j][k] \leftarrow m$.

Esempio 4.5: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza del BKP dell'Esempio 4.1, con uno zaino di capacità $c = 10$ e i seguenti profitti, pesi e limiti:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

Per ogni cella con $j \geq 1$ la tabella riporta i valori $d\,p[j]+v[j-1][k-d\,w[j]]$ per $d = 0, 1, \ldots, \min\{b[j], \lfloor k/w[j] \rfloor\}$, in quest'ordine, e il numero t di copie memorizzato in $s[j][k]$. In caso di pareggio si conserva il d minimo, perché m viene aggiornato solo quando il valore è strettamente maggiore. La riga $j = 0$ è $s[0][k] = 0$ per ogni k.

j	k	$s[j][k]$	Valori per $d = 0, 1, \ldots$
1	0	0	0
1	1	0	0
1	2	1	0, 6
1	3	1	0, 6
1	4	2	0, 6, 12
1	5	2	0, 6, 12
1	6	2	0, 6, 12
1	7	2	0, 6, 12
1	8	2	0, 6, 12
1	9	2	0, 6, 12
1	10	2	0, 6, 12
2	0	0	0
2	1	0	0
2	2	0	6
2	3	1	6, 10
2	4	0	12, 10
2	5	1	12, 16
2	6	2	12, 16, 20
2	7	1	12, 22, 20
2	8	2	12, 22, 26
2	9	2	12, 22, 26
2	10	2	12, 22, 32
3	0	0	0
3	1	0	0
3	2	0	6
3	3	0	10
3	4	0	12
3	5	0	16, 12
3	6	0	20, 12
3	7	0	22, 18
3	8	0	26, 22
3	9	0	26, 24
3	10	0	32, 28

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

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

Per esempio, $s[1][4] = 2$ perché la formula (4.1) seleziona $d = 2$ copie del tipo di oggetto 1 nel calcolo di $v[1][4] = 12$, e $s[2][10] = 2$ perché vengono selezionate $d = 2$ copie del tipo di oggetto 2 nel calcolo di $v[2][10] = 32$. Poiché $s[3][10] = 0$, nella soluzione ottima dell'istanza originale non si usa alcuna copia del tipo di oggetto 3.

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

$$x_j \leftarrow s[j][k], \qquad k \leftarrow k - s[j][k]\, w_j.$$

Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, in cui ogni componente x_j è il numero di copie del tipo di oggetto j in una soluzione ottima.

Algoritmo

Algorithm: BKP_Reconstruct

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

Output: Una soluzione ottima dell'istanza del BKP

```
1 Function BKP_Reconstruct(n, c, w, s)
2     x := new array[1..n]
3     k := c
4     for j := n downto 1 do
5         x[j] ← s[j][k]
6         k ← k − x[j] w[j]
7     end
8     return x
```

Esempio 4.6: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 4.1, abbiamo la seguente matrice s:

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

Possiamo ricostruire una soluzione ottima partendo da $j = 3$ e $k = 10$ come segue:

Iterazione	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	Azione
–	–	–	–	0	0	0	$k = 10$
1	3	10	0	0	0	0	$x[3] \leftarrow 0$, $k = 10$
2	2	10	2	0	2	0	$x[2] \leftarrow 2$, $k \leftarrow 10 - 2 \cdot 3 = 4$
3	1	4	2	2	2	0	$x[1] \leftarrow 2$, $k \leftarrow 4 - 2 \cdot 2 = 0$

Quindi il vettore

$$(2, 2, 0)$$

è una soluzione ottima. Il peso totale corrispondente è $2 \cdot 2 + 2 \cdot 3 = 4 + 6 = 10 = c$. Il valore ottimo corrispondente è $v(3, 10) = 32$.

4.2.1 Numero di passi

- Sia $b_{\max} = \max\{b[j] : j \in \{1, \ldots, n\}\}$, e indichiamo con $S(n, c, b_{\max})$ il numero di passi eseguiti da `BKP_DP` nel caso peggiore, dove n è il numero di tipi di oggetto e c è la capacità dello zaino. Calcoliamo $S(n, c, b_{\max})$ contando quante volte vengono eseguite le righe dello pseudocodice di `BKP_DP`.

- Per ogni coppia (j, k) il ciclo più interno su d esegue $\min\{b_j, \lfloor k/w_j \rfloor\} + 1$ iterazioni, cioè al più $b_{\max} + 1$. Contiamo $b_{\max} + 1$ iterazioni per ogni coppia (j, k) e supponiamo che la condizione alla riga 10 sia vera a ogni iterazione, così che $S(n, c, b_{\max})$ sia un upper bound sul numero di passi. Il numero di esecuzioni delle righe rilevanti è, al più:

  - 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 9 viene eseguita al più $n(c + 1)(b_{\max} + 1)$ volte.
  - La riga 10 viene eseguita al più $n(c + 1)(b_{\max} + 1)$ volte.
  - La riga 11 viene eseguita al più $n(c + 1)(b_{\max} + 1)$ volte.
  - La riga 14 viene eseguita $n(c + 1)$ volte.
  - La riga 17 viene eseguita 1 volta.

  Quindi il numero totale di passi nel caso peggiore è

  $$1 + (c + 1) + (c + 1) + n + n(c + 1) + n(c + 1) + 3n(c + 1)(b_{\max} + 1) + n(c + 1) + 1.$$

  Pertanto,

  $$S(n, c, b_{\max}) = 3n(c + 1)(b_{\max} + 1) + 3n(c + 1) + n + 2c + 4.$$

  Di conseguenza,

  $$S(n, c, b_{\max}) = \Theta(nc\, b_{\max}).$$

Infatti,

$$\begin{aligned}
&\lim_{n,c,b_{\max}\to\infty} \frac{3n(c+1)(b_{\max}+1)+3n(c+1)+n+2c+4}{nc\,b_{\max}} \\
&= \lim_{n,c,b_{\max}\to\infty} \left(\frac{3(c+1)(b_{\max}+1)}{c\,b_{\max}} + \frac{3(c+1)}{c\,b_{\max}} + \frac{1}{c\,b_{\max}} + \frac{2}{n\,b_{\max}} + \frac{4}{nc\,b_{\max}}\right) \\
&= 3 \in \mathbb{R}_{>0}.
\end{aligned}$$

L'algoritmo di PD esegue

$$S(n, c, b_{\max}) = \Theta(nc\,b_{\max})$$

passi nel caso peggiore.

- La versione `BKP_DP_InfoSol` memorizza la matrice aggiuntiva s ed esegue solo un numero costante di operazioni in più negli stessi cicli. Quindi il suo numero di passi resta $\Theta(nc\,b_{\max})$.
- Sostituire ogni limite b_j con $\min\{b_j, \lfloor c/w_j \rfloor\}$ non cambia il problema, quindi possiamo supporre $b_{\max} \leq c$. Sotto questa ipotesi il bound $\Theta(nc\,b_{\max})$ è stretto: è raggiunto dalle istanze in cui tutti i pesi sono uguali a 1 e tutti i limiti sono uguali a $b_{\max}$. Infatti il ciclo più interno esegue allora $\sum_{k=0}^{c}(\min\{b_{\max}, k\}+1) \geq (c+1)\,b_{\max}/2$ iterazioni per ogni tipo di oggetto.

4.3 Errori tipici

Attenzione

- **Provare più di b_j copie.** Il numero di copie del tipo di oggetto j varia in $\{0, 1, \ldots, \min\{b_j, \lfloor k/w_j \rfloor\}\}$: contano sia il limite sia la capacità.
- **Ricostruire con l'aggiornamento sbagliato.** Dopo aver letto $s[j][k]$ la capacità diminuisce di $s[j][k]\,w_j$, non di w_j, e l'indice del tipo di oggetto diminuisce sempre di uno.
- **Contare i passi con $b_{\max}$ per ogni cella.** Il ciclo più interno ha $\min\{b_j, \lfloor k/w_j \rfloor\}+1$ iterazioni, quindi $b_{\max}+1$ è solo un upper bound.
- **Risolvere i pareggi in modo incoerente.** L'algoritmo di PD aggiorna m solo con un valore strettamente maggiore, quindi $s[j][k]$ conserva il numero minimo di copie che raggiunge il massimo.

4.4 In sintesi

In sintesi

- Il BKP generalizza sia il KP ($b_j = 1$) sia l'UKP ($b_j = \lfloor c/w_j \rfloor$); la ricorrenza prova ogni numero d di copie del tipo di oggetto j.
- La matrice $v(j,k)$ si riempie in $\Theta(nc\,b_{\max})$ passi, un numero pseudo-polinomiale.
- La matrice s memorizza il numero di copie di ciascun tipo di oggetto, e una soluzione ottima si legge all'indietro a partire da (n,c).
- Sostituire b_j con $\min\{b_j, \lfloor c/w_j \rfloor\}$ lascia invariato il problema e dà $b_{\max} \leq c$.

Quadro riassuntivo dell'esempio del BKP. Esempio 4.1 ($c = 10$, tre tipi di oggetto):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	32	32	valore ottimo, soluzione $(2,2,0)$

Capitolo 5

PD per il problema del subset sum

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: la ricorrenza $v(j, k)$ del problema dello zaino 0/1 (KP) su oggetti e capacità, la matrice s delle decisioni e la ricostruzione di una soluzione.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema del subset sum e la sua soluzione come problema dello zaino 0/1 con $p_j = w_j$.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- scrivere la ricorrenza $r(j, k) \in \{true, false\}$ delle somme raggiungibili, con i suoi casi base;
- eseguire a mano l'algoritmo di PD e leggere la risposta dell'istanza dell'SSP in $r(n, c)$;
- riempire la matrice s e ricostruire un sottoinsieme di peso totale c quando la risposta è sì;
- mettere in relazione $r(j, k)$ con l'algoritmo di PD del KP con $p_j = w_j$, e contare i $\Theta(nc)$ passi.

Il problema di questo capitolo è il problema del subset sum (SSP): la sua definizione e le sue ipotesi sono nella sua pagina.

5.1 Un algoritmo di PD

Esempio 5.1: Istanza dell'SSP

Consideriamo un valore obiettivo $c = 10$ e un insieme $\mathscr{I} = \{1, 2, 3, 4\}$ di 4 oggetti ($n = 4$), con i seguenti pesi:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
w_j	3	5	4	7

Chiamiamo questa istanza dell'SSP Esempio 5.1.

- Un sottoinsieme $S \subseteq \mathscr{I}$ si descrive con il vettore binario $(x_1, x_2, \ldots, x_n)$, dove $x_j = 1$ se l'oggetto j appartiene a S e $x_j = 0$ altrimenti.
- La risposta all'istanza dell'SSP è **sì** se qualche vettore soddisfa $\sum_{j=1}^{n} w_j x_j = c$, e **no** altrimenti.
- Passo 1: calcolo della risposta

  Per ogni $j \in \{0, 1, \ldots, n\}$ e $k \in \{0, 1, \ldots, c\}$, sia $r(j, k) \in \{true, false\}$ la risposta della sottoistanza dell'SSP data da:

  1. un valore obiettivo k,
  2. un insieme $\{1, 2, \ldots, j\}$ di oggetti disponibili,
  3. per ogni oggetto $h \in \{1, 2, \ldots, j\}$, un peso w_h.

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

$$r(j,k) = \begin{cases} true & \text{se } k = 0, \\ false & \text{se } j = 0 \text{ e } k \geq 1, \\ r(j-1,k) & \text{se } j,k \geq 1 \text{ e } w_j > k, \\ r(j-1,k) \vee r(j-1,k-w_j) & \text{se } j,k \geq 1 \text{ e } w_j \leq k. \end{cases} \tag{5.1}$$

  - Il simbolo $\vee$ è l'**or** logico: $r(j, k)$ è *true* se almeno uno dei due valori è *true*.
  - I casi base sono $r(j, 0) = true$, perché il sottoinsieme vuoto ha peso totale 0, e $r(0, k) = false$ per $k \geq 1$, perché nessun oggetto è disponibile.

- Un sottoinsieme di $\{1, 2, \ldots, j\}$ con peso totale k o non contiene l'oggetto j, e allora è un sottoinsieme di $\{1, 2, \ldots, j-1\}$ con peso totale k, oppure contiene l'oggetto j, e allora gli altri oggetti formano un sottoinsieme di $\{1, 2, \ldots, j-1\}$ con peso totale $k - w_j$.

- Quindi la formula (5.1) genera al più 2 **sottoproblemi indipendenti**, e un sottoinsieme per (j, k) si costruisce da un sottoinsieme di uno di essi aggiungendo o no l'oggetto j.

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

- Decisioni diverse sugli oggetti possono portare alla stessa coppia (j, k), e quindi lo stesso valore $r(j, k)$ può servire più volte durante la valutazione ricorsiva della formula (5.1).

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

- Il valore $r(j, k)$ dipende solo dai valori $r(j-1, k')$ con $k' \leq k$. Quindi l'**approccio bottom-up** calcola le righe $j = 0, 1, \ldots, n$ in quest'ordine, ciascuna per k crescente, e memorizza ogni valore in una **tabella di lookup**.

- L'SSP si può risolvere anche con l'algoritmo di PD del KP, con $p_j = w_j$ per ogni oggetto j. Il suo valore $v(j, k)$ è il massimo peso totale, non superiore a k, di un sottoinsieme di $\{1, 2, \ldots, j\}$, e

$$r(j, k) = \mathit{true} \iff v(j, k) = k.$$

- La ricorrenza (5.1) memorizza un solo bit per cella, e le sue operazioni logiche sostituiscono le somme e i confronti della ricorrenza del KP.

- Passo 2: calcolo di un sottoinsieme

  - Quando la risposta è sì, un sottoinsieme di peso totale c si ricostruisce dalla matrice s: $s[j][k] = 1$ se l'oggetto j è usato per raggiungere il valore *true* di $r[j][k]$, e $s[j][k] = 0$ altrimenti.

  - L'algoritmo di PD preferisce non usare l'oggetto j: $s[j][k] = 1$ solo se $r[j-1][k]$ è *false* e $r[j-1][k-w_j]$ è *true*.

Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, che memorizza la matrice s insieme alla matrice r:

Algoritmo

Algorithm: SSP_DP

Input: Un'istanza dell'SSP data da n, dal vettore $w[1..n]$ e dal valore obiettivo c

Output: La risposta $r[n][c]$ dell'istanza dell'SSP e la matrice $s[0..n][0..c]$

```
1  Function SSP_DP(n, w, c)
2      r := new array[0..n][0..c]; s := new array[0..n][0..c]
3      r[0][0] ← true; s[0][0] ← 0
4      for k := 1 to c do
5          r[0][k] ← false; s[0][k] ← 0
6      end
7      for j := 1 to n do
8          for k := 0 to c do
9              r[j][k] ← r[j − 1][k]; s[j][k] ← 0
10             if r[j][k] = false e w[j] ≤ k e r[j − 1][k − w[j]] = true then
11                 r[j][k] ← true; s[j][k] ← 1
12             end
13         end
14     end
15     return r[n][c], s
```

– La matrice $r[0..n][0..c]$ è la **tabella di lookup** delle risposte, e la matrice $s[0..n][0..c]$ memorizza le decisioni.

– Le righe 3–5 riempiono la riga $j = 0$: senza oggetti è raggiungibile solo il valore obiettivo 0.

– Per ogni coppia (j, k), l'algoritmo di PD copia prima la risposta senza l'oggetto j, e poi usa l'oggetto j solo se questa risposta è *false* e l'oggetto j rende raggiungibile il valore k.

– Infine, la risposta dell'istanza dell'SSP viene restituita come $r[n][c]$, insieme alla matrice s.

Esempio 5.2: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'SSP dell'Esempio 5.1, con valore obiettivo $c = 10$ e pesi $w = (3, 5, 4, 7)$.

Per ogni passo dell'algoritmo di PD con $j \geq 1$, la tabella riporta i valori di j e k, le celle $r[j][k]$ e $s[j][k]$ e l'operazione eseguita; T sta per *true* e F per *false*. La riga $j = 0$ è $r[0][0] = \text{T}$ e $r[0][k] = \text{F}$ per $k \geq 1$, con $s[0][k] = 0$.

j	k	$r[j][k]$	$s[j][k]$	Azione
1	0	T	0	$w_1 = 3 > 0$: $r[0][0] = \mathrm{T}$
1	1	F	0	$w_1 = 3 > 1$: $r[0][1] = \mathrm{F}$
1	2	F	0	$w_1 = 3 > 2$: $r[0][2] = \mathrm{F}$
1	3	T	1	$r[0][3] \vee r[0][0] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
1	4	F	0	$r[0][4] \vee r[0][1] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	5	F	0	$r[0][5] \vee r[0][2] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	6	F	0	$r[0][6] \vee r[0][3] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	7	F	0	$r[0][7] \vee r[0][4] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	8	F	0	$r[0][8] \vee r[0][5] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	9	F	0	$r[0][9] \vee r[0][6] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	10	F	0	$r[0][10] \vee r[0][7] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	0	T	0	$w_2 = 5 > 0$: $r[1][0] = \mathrm{T}$
2	1	F	0	$w_2 = 5 > 1$: $r[1][1] = \mathrm{F}$
2	2	F	0	$w_2 = 5 > 2$: $r[1][2] = \mathrm{F}$
2	3	T	0	$w_2 = 5 > 3$: $r[1][3] = \mathrm{T}$
2	4	F	0	$w_2 = 5 > 4$: $r[1][4] = \mathrm{F}$
2	5	T	1	$r[1][5] \vee r[1][0] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
2	6	F	0	$r[1][6] \vee r[1][1] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	7	F	0	$r[1][7] \vee r[1][2] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	8	T	1	$r[1][8] \vee r[1][3] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
2	9	F	0	$r[1][9] \vee r[1][4] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	10	F	0	$r[1][10] \vee r[1][5] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$

j	k	$r[j][k]$	$s[j][k]$	Azione
3	0	T	0	$w_3 = 4 > 0$: $r[2][0] = \mathrm{T}$
3	1	F	0	$w_3 = 4 > 1$: $r[2][1] = \mathrm{F}$
3	2	F	0	$w_3 = 4 > 2$: $r[2][2] = \mathrm{F}$
3	3	T	0	$w_3 = 4 > 3$: $r[2][3] = \mathrm{T}$
3	4	T	1	$r[2][4] \vee r[2][0] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
3	5	T	0	$r[2][5] \vee r[2][1] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
3	6	F	0	$r[2][6] \vee r[2][2] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
3	7	T	1	$r[2][7] \vee r[2][3] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
3	8	T	0	$r[2][8] \vee r[2][4] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
3	9	T	1	$r[2][9] \vee r[2][5] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
3	10	F	0	$r[2][10] \vee r[2][6] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
4	0	T	0	$w_4 = 7 > 0$: $r[3][0] = \mathrm{T}$
4	1	F	0	$w_4 = 7 > 1$: $r[3][1] = \mathrm{F}$
4	2	F	0	$w_4 = 7 > 2$: $r[3][2] = \mathrm{F}$
4	3	T	0	$w_4 = 7 > 3$: $r[3][3] = \mathrm{T}$
4	4	T	0	$w_4 = 7 > 4$: $r[3][4] = \mathrm{T}$
4	5	T	0	$w_4 = 7 > 5$: $r[3][5] = \mathrm{T}$
4	6	F	0	$w_4 = 7 > 6$: $r[3][6] = \mathrm{F}$
4	7	T	0	$r[3][7] \vee r[3][0] = \mathrm{T} \vee \mathrm{T} = \mathrm{T}$
4	8	T	0	$r[3][8] \vee r[3][1] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
4	9	T	0	$r[3][9] \vee r[3][2] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
4	10	T	1	$r[3][10] \vee r[3][3] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$

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

$r[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	T	F	F	F	F	F	F	F	F	F	F
$j=1$	T	F	F	T	F	F	F	F	F	F	F
$j=2$	T	F	F	T	F	T	F	F	T	F	F
$j=3$	T	F	F	T	T	T	F	T	T	T	F
$j=4$	T	F	F	T	T	T	F	T	T	T	T

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

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

Poiché $r[4][10] = \text{T}$, la risposta all'istanza dell'SSP è **sì**. Per esempio, $s[4][10] = 1$ perché $r[3][10] = \text{F}$ e $r[3][3] = \text{T}$: l'oggetto 4 serve per raggiungere il valore obiettivo 10.

– Calcolata la matrice s, se $r[n][c]$ è *true* si ricostruisce un sottoinsieme di peso totale c scorrendo gli oggetti da $j = n$ fino a $j = 1$, partendo da $k = c$ e da $x_j = 0$ per ogni oggetto j:

$$x_j \leftarrow s[j][k], \qquad k \leftarrow k - s[j][k]\, w_j.$$

– La ricostruzione termina con $k = 0$, perché ogni passo mantiene $r[j-1][k]$ uguale a *true*.

Algoritmo

Algorithm: SSP_Reconstruct

Input: Il numero n di oggetti, il valore obiettivo c, il vettore $w[1..n]$ e la matrice $s[0..n][0..c]$, con $r[n][c] = true$

Output: Un sottoinsieme di peso totale c, come vettore binario

```
1 Function SSP_Reconstruct(n, c, w, s)
2     x := new array[1..n]
3     k := c
4     for j := n downto 1 do
5         x[j] ← s[j][k]
6         k ← k − x[j] w[j]
7     end
8     return x
```

Esempio 5.3: Calcolo di un sottoinsieme

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 5.1, partiamo da $j = 4$ e $k = 10$:

Iterazione	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Azione
–	–	–	–	0	0	0	0	$k = 10$
1	4	10	1	0	0	0	1	$x[4] \leftarrow 1$, $k \leftarrow 10 - 7 = 3$
2	3	3	0	0	0	0	1	$x[3] \leftarrow 0$, $k = 3$
3	2	3	0	0	0	0	1	$x[2] \leftarrow 0$, $k = 3$
4	1	3	1	1	0	0	1	$x[1] \leftarrow 1$, $k \leftarrow 3 - 3 = 0$

La risposta all'istanza dell'SSP dell'Esempio 5.1 è **sì**: il vettore $(1, 0, 0, 1)$ seleziona il sottoinsieme $\{1, 4\}$, il cui peso totale è $3 + 7 = c$.

5.1.1 Numero di passi

- Indichiamo con $S(n, c)$ il numero di passi eseguiti da `SSP_DP` nel caso peggiore. Contiamo quante volte viene eseguita ogni riga dello pseudocodice, supponendo che la condizione alla riga 10 sia sempre vera:
  - Le righe 2 e 3 vengono eseguite 1 volta ciascuna.
  - La riga 4 viene eseguita c volte, e la riga 5 c volte.
  - La riga 7 viene eseguita n volte.
  - Le righe 8, 9, 10 e 11 vengono eseguite $n(c + 1)$ volte ciascuna.
  - La riga 15 viene eseguita 1 volta.
- Quindi il numero totale di passi nel caso peggiore è

$$S(n, c) = 1 + 1 + 2c + n + 4n(c + 1) + 1 = 4n(c + 1) + n + 2c + 3.$$

  Di conseguenza,

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

  poiché

$$\lim_{n,c\to\infty} \frac{4n(c + 1) + n + 2c + 3}{nc} = \lim_{n,c\to\infty} \left(4 + \frac{4}{c} + \frac{1}{c} + \frac{2}{n} + \frac{3}{nc}\right) = 4 \in \mathbb{R}_{>0}.$$

L'algoritmo di PD per l'SSP esegue $S(n, c) = \Theta(nc)$ passi nel caso peggiore, lo stesso numero dell'algoritmo di PD del KP.

- La ricostruzione esegue $\Theta(n)$ passi, uno per ogni oggetto.
- Il numero di passi contiene il valore obiettivo c, un numero dell'input, e non solo la dimensione dell'input: è **pseudo-polinomiale**.

5.2 Errori tipici

Attenzione

- **Porre $r(j, 0) =$ *false*.** Il valore obiettivo 0 è sempre raggiungibile con il sottoinsieme vuoto; con $r(j, 0) = \mathit{false}$ nessun valore diventerebbe mai *true*.
- **Porre $r(0, k) =$ *true* per $k \geq 1$.** Senza oggetti è raggiungibile solo il valore obiettivo 0.
- **Leggere la risposta nella cella sbagliata.** La risposta è $r(n, c)$; un valore *true* in un'altra cella dell'ultima riga dice che è raggiungibile un valore obiettivo diverso.
- **Ricostruire quando la risposta è no.** Se $r(n, c)$ è *false*, nessun sottoinsieme ha peso totale c, e la matrice s non dà nulla da ricostruire.

5.3 In sintesi

In sintesi

- La ricorrenza $r(j, k) = r(j-1, k) \vee r(j-1, k-w_j)$ dice se il valore obiettivo k è raggiungibile con gli oggetti $1, 2, \ldots, j$.
- I casi base sono $r(j, 0) = \mathit{true}$ e $r(0, k) = \mathit{false}$ per $k \geq 1$; la risposta è $r(n, c)$.
- La matrice s memorizza se l'oggetto j è usato, e un sottoinsieme di peso totale c si legge all'indietro a partire da (n, c).
- L'algoritmo di PD esegue $\Theta(nc)$ passi, come l'algoritmo di PD del KP con $p_j = w_j$.

Quadro riassuntivo dell'esempio dell'SSP. Esempio 5.1 ($c = 10$, quattro oggetti):

Metodo	Risposta	Origine
Algoritmo di PD	sì	$r(4, 10) = \mathit{true}$, sottoinsieme $\{1, 4\}$

Capitolo 6

PD per il problema dello zaino a scelta multipla

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: la ricorrenza $v(j, k)$ del problema dello zaino 0/1 (KP) su oggetti e capacità, la matrice s delle decisioni e la ricostruzione di una soluzione.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema dello zaino a scelta multipla, le sue classi $\mathscr{J}_h$ e la sua condizione di ammissibilità.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- scrivere la ricorrenza $v(h, k)$ su classi e capacità, con il valore $-\infty$ per le sottoistanze senza soluzione ammissibile;
- eseguire a mano l'algoritmo di PD, provando ogni oggetto della classe h che entra nello zaino;
- riempire la matrice s degli oggetti scelti e ricostruire una soluzione ottima, un oggetto per classe;
- contare i $\Theta(nc)$ passi dell'algoritmo di PD.

Il problema di questo capitolo è il problema dello zaino a scelta multipla (MCKP): la sua definizione e le sue ipotesi sono nella sua pagina.

6.1 Un algoritmo di PD

Esempio 6.1: Istanza dell'MCKP

Consideriamo uno zaino di capacità $c = 10$ e un insieme $\mathscr{I} = \{1, 2, \ldots, 7\}$ di 7 oggetti ($n = 7$), partizionato in $q = 3$ classi: $\mathscr{J}_1 = \{1, 2\}$, $\mathscr{J}_2 = \{3, 4, 5\}$ e $\mathscr{J}_3 = \{6, 7\}$. I profitti e i pesi sono:

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
p_j	4	8	3	5	7	3	7
w_j	1	4	1	2	5	2	3

Chiamiamo questa istanza dell'MCKP Esempio 6.1.

- Una soluzione ammissibile sceglie esattamente un oggetto in ogni classe, quindi si descrive con il sottoinsieme selezionato $S \subseteq \mathscr{I}$, con $|S \cap \mathscr{J}_h| = 1$ per ogni classe h, oppure con il vettore binario $(x_1, x_2, \ldots, x_n)$ di S.

- Le classi sono gli stadi dell'algoritmo di PD: la classe h si decide dopo le classi $1, 2, \ldots, h-1$, e la sua decisione è l'oggetto scelto in essa.

- Passo 1: calcolo del valore ottimo

  Per ogni $h \in \{0, 1, \ldots, q\}$ e $k \in \{0, 1, \ldots, c\}$, sia $v(h, k)$ il valore ottimo della sottoistanza dell'MCKP data da:

  1. uno zaino di capacità k,
  2. le classi $\mathscr{J}_1, \mathscr{J}_2, \ldots, \mathscr{J}_h$,
  3. per ogni oggetto $j \in \mathscr{J}_1 \cup \mathscr{J}_2 \cup \cdots \cup \mathscr{J}_h$, un profitto p_j e un peso w_j,

  con $v(h, k) = -\infty$ se questa sottoistanza non ha soluzioni ammissibili.

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

$$v(h,k) = \begin{cases} 0 & \text{se } h = 0, \\ -\infty & \text{se } h \geq 1 \text{ e } \forall j \in \mathscr{J}_h\colon\ w_j > k, \\ \max\limits_{j \in \mathscr{J}_h\colon w_j \leq k} \{p_j + v(h-1, k-w_j)\} & \text{se } h \geq 1 \text{ e } \exists j \in \mathscr{J}_h\colon\ w_j \leq k. \end{cases} \tag{6.1}$$

  - Il valore $-\infty$ segue le regole $p_j + (-\infty) = -\infty$ e $\max\{-\infty, a\} = a$: una sottoistanza senza soluzioni ammissibili non dà mai un valore migliore.

- I casi base sono $v(0, k) = 0$ per ogni k: senza classi, il sottoinsieme vuoto è ammissibile e ha profitto 0, qualunque sia la capacità.
- La formula (6.1) genera al più $|\mathscr{J}_h|$ **sottoproblemi indipendenti**, uno per ogni oggetto j della classe h che entra nella capacità k: il sottoproblema residuo ha le classi $1, 2, \ldots, h-1$ e la capacità $k - w_j$.
- Quindi una soluzione ottima per (h, k) si costruisce combinando il miglior oggetto j della classe h con una soluzione ottima del suo sottoproblema residuo.

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

- Scelte diverse nelle classi $h+1, h+2, \ldots, q$ possono lasciare la stessa capacità residua, e quindi lo stesso valore $v(h, k)$ può servire più volte durante la valutazione ricorsiva della formula (6.1).

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

- Il valore $v(h, k)$ dipende solo dai valori $v(h-1, k')$ con $k' < k$. Quindi l'**approccio bottom-up** calcola le righe $h = 0, 1, \ldots, q$ in quest'ordine, ciascuna per k crescente, e memorizza ogni valore in una **tabella di lookup**.
- Il valore ottimo dell'istanza dell'MCKP è $v(q, c)$, e l'istanza è inammissibile se e solo se $v(q, c) = -\infty$.

- Passo 2: calcolo di una soluzione ottima
  - Per ogni classe h e capacità k, memorizziamo $s[h][k]$, l'oggetto della classe h scelto in una soluzione ottima della sottoistanza (h, k), cioè l'oggetto che massimizza il membro destro della formula (6.1).
  - Se $v[h][k] = -\infty$, non si sceglie alcun oggetto e $s[h][k] = 0$.

Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, che memorizza la matrice s insieme alla matrice v:

Algoritmo

Algorithm: MCKP_DP

Input: Un'istanza dell'MCKP data da q, dalle classi $\mathscr{J}_1, \ldots, \mathscr{J}_q$, dai vettori $p[1..n]$, $w[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza dell'MCKP e la matrice $s[0..q][0..c]$

```
1  Function MCKP_DP(q, J, p, w, c)
2      v := new array[0..q][0..c]; s := new array[0..q][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
6      for h := 1 to q do
7          for k := 0 to c do
8              m := −∞; t := 0
9              foreach j ∈ J_h do
10                 if w[j] ≤ k e p[j] + v[h − 1][k − w[j]] > m then
11                     m ← p[j] + v[h − 1][k − w[j]]; t ← j
12                 end
13             end
14             v[h][k] ← m; s[h][k] ← t
15         end
16     end
17     return v[q][c], s
```

– La variabile m parte da $-\infty$, non da 0: si deve scegliere esattamente un oggetto della classe h, quindi il valore di una scelta vuota non è disponibile.

– Il ciclo più interno prova ogni oggetto j della classe h che entra nello zaino, conserva in m il valore massimo trovato finora e in t l'oggetto che lo dà.

– Se nessun oggetto della classe h dà un valore maggiore di $-\infty$, la cella mantiene $v[h][k] = -\infty$ e $s[h][k] = 0$.

– In caso di pareggio si conserva l'oggetto incontrato per primo in $\mathscr{J}_h$, perché m viene aggiornato solo quando il valore è strettamente maggiore.

Esempio 6.2: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'MCKP dell'Esempio 6.1, con capacità $c = 10$ e classi $\mathscr{J}_1 = \{1, 2\}$, $\mathscr{J}_2 = \{3, 4, 5\}$, $\mathscr{J}_3 = \{6, 7\}$.

Per ogni passo dell'algoritmo di PD con $h \geq 1$, la tabella riporta i valori di h e k, le celle $v[h][k]$ e $s[h][k]$ e l'operazione eseguita. La riga $h = 0$ è $v[0][k] = 0$ e $s[0][k] = 0$ per ogni k.

h	k	$v[h][k]$	$s[h][k]$	Azione
1	0	$-\infty$	0	nessun oggetto di $\mathscr{J}_1$ entra: $-\infty$
1	1	4	1	$\max\{p_1+v[0][0]\} = \max\{4\} = 4$
1	2	4	1	$\max\{p_1+v[0][1]\} = \max\{4\} = 4$
1	3	4	1	$\max\{p_1+v[0][2]\} = \max\{4\} = 4$
1	4	8	2	$\max\{p_1+v[0][3],\ p_2+v[0][0]\} = \max\{4,8\} = 8$
1	5	8	2	$\max\{p_1+v[0][4],\ p_2+v[0][1]\} = \max\{4,8\} = 8$
1	6	8	2	$\max\{p_1+v[0][5],\ p_2+v[0][2]\} = \max\{4,8\} = 8$
1	7	8	2	$\max\{p_1+v[0][6],\ p_2+v[0][3]\} = \max\{4,8\} = 8$
1	8	8	2	$\max\{p_1+v[0][7],\ p_2+v[0][4]\} = \max\{4,8\} = 8$
1	9	8	2	$\max\{p_1+v[0][8],\ p_2+v[0][5]\} = \max\{4,8\} = 8$
1	10	8	2	$\max\{p_1+v[0][9],\ p_2+v[0][6]\} = \max\{4,8\} = 8$
2	0	$-\infty$	0	nessun oggetto di $\mathscr{J}_2$ entra: $-\infty$
2	1	$-\infty$	0	$\max\{p_3+v[1][0]\} = \max\{-\infty\} = -\infty$
2	2	7	3	$\max\{p_3+v[1][1],\ p_4+v[1][0]\} = \max\{7,-\infty\} = 7$
2	3	9	4	$\max\{p_3+v[1][2],\ p_4+v[1][1]\} = \max\{7,9\} = 9$
2	4	9	4	$\max\{p_3+v[1][3],\ p_4+v[1][2]\} = \max\{7,9\} = 9$
2	5	11	3	$\max\{p_3+v[1][4],\ p_4+v[1][3],\ p_5+v[1][0]\} = \max\{11,9,-\infty\} = 11$
2	6	13	4	$\max\{p_3+v[1][5],\ p_4+v[1][4],\ p_5+v[1][1]\} = \max\{11,13,11\} = 13$
2	7	13	4	$\max\{p_3+v[1][6],\ p_4+v[1][5],\ p_5+v[1][2]\} = \max\{11,13,11\} = 13$
2	8	13	4	$\max\{p_3+v[1][7],\ p_4+v[1][6],\ p_5+v[1][3]\} = \max\{11,13,11\} = 13$
2	9	15	5	$\max\{p_3+v[1][8],\ p_4+v[1][7],\ p_5+v[1][4]\} = \max\{11,13,15\} = 15$
2	10	15	5	$\max\{p_3+v[1][9],\ p_4+v[1][8],\ p_5+v[1][5]\} = \max\{11,13,15\} = 15$

h	k	$v[h][k]$	$s[h][k]$	Azione
3	0	$-\infty$	0	nessun oggetto di $\mathscr{J}_3$ entra: $-\infty$
3	1	$-\infty$	0	nessun oggetto di $\mathscr{J}_3$ entra: $-\infty$
3	2	$-\infty$	0	$\max\{p_6+v[2][0]\} = \max\{-\infty\} = -\infty$
3	3	$-\infty$	0	$\max\{p_6+v[2][1],\ p_7+v[2][0]\} = \max\{-\infty,-\infty\} = -\infty$
3	4	10	6	$\max\{p_6+v[2][2],\ p_7+v[2][1]\} = \max\{10,-\infty\} = 10$
3	5	14	7	$\max\{p_6+v[2][3],\ p_7+v[2][2]\} = \max\{12,14\} = 14$
3	6	16	7	$\max\{p_6+v[2][4],\ p_7+v[2][3]\} = \max\{12,16\} = 16$
3	7	16	7	$\max\{p_6+v[2][5],\ p_7+v[2][4]\} = \max\{14,16\} = 16$
3	8	18	7	$\max\{p_6+v[2][6],\ p_7+v[2][5]\} = \max\{16,18\} = 18$
3	9	20	7	$\max\{p_6+v[2][7],\ p_7+v[2][6]\} = \max\{16,20\} = 20$
3	10	20	7	$\max\{p_6+v[2][8],\ p_7+v[2][7]\} = \max\{16,20\} = 20$

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

$v[h][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$h=0$	0	0	0	0	0	0	0	0	0	0	0
$h=1$	$-\infty$	4	4	4	8	8	8	8	8	8	8
$h=2$	$-\infty$	$-\infty$	7	9	9	11	13	13	13	15	15
$h=3$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	10	14	16	16	18	20	20

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

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

Per esempio, $v[2][1] = -\infty$: ogni scelta di un oggetto in $\mathscr{J}_1$ e di uno in $\mathscr{J}_2$ pesa almeno $w_1 + w_3 = 2$. Il valore ottimo dell'istanza è $v[3][10] = 20$, raggiunto con l'oggetto $s[3][10] = 7$ della classe 3.

– Calcolata la matrice s, se $v[q][c] > -\infty$ si ricostruisce una soluzione ottima scorrendo le classi da $h = q$ fino a $h = 1$, partendo da $k = c$ e da $x_j = 0$ per ogni oggetto j:

$$j \leftarrow s[h][k], \qquad x_j \leftarrow 1, \qquad k \leftarrow k - w_j.$$

– Ogni classe dà esattamente un oggetto, quindi la ricostruzione seleziona q oggetti.

Algoritmo

Algorithm: MCKP_Reconstruct

Input: Il numero q di classi, la capacità c, il vettore $w[1..n]$ e la matrice $s[0..q][0..c]$, con $v[q][c] > -\infty$

Output: Una soluzione ottima dell'istanza dell'MCKP, come vettore binario

```
1 Function MCKP_Reconstruct(q, c, w, s)
2     x := new array[1..n], con x[j] = 0 per ogni j
3     k := c
4     for h := q downto 1 do
5         j ← s[h][k]; x[j] ← 1
6         k ← k − w[j]
7     end
8     return x
```

Esempio 6.3: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 6.1, partiamo da $h = 3$ e $k = 10$:

Iterazione	h	k	$j = s[h][k]$	Selezione	Capacità
–	–	–	–	–	$k = 10$
1	3	10	7	$x[7] \leftarrow 1$	$k \leftarrow 10 - 3 = 7$
2	2	7	4	$x[4] \leftarrow 1$	$k \leftarrow 7 - 2 = 5$
3	1	5	2	$x[2] \leftarrow 1$	$k \leftarrow 5 - 4 = 1$

Una soluzione ottima dell'istanza dell'MCKP dell'Esempio 6.1 è $S = \{2, 4, 7\}$, cioè il vettore $x = (0, 1, 0, 1, 0, 0, 1)$. Il suo peso totale è $4 + 2 + 3 = 9 \leq c$ e il suo valore ottimo è $v(3, 10) = 20$.

6.1.1 Numero di passi

- Indichiamo con $S(n, q, c)$ il numero di passi eseguiti da `MCKP_DP` nel caso peggiore. Il ciclo più interno scorre gli oggetti della classe h, quindi su tutte le classi viene eseguito $|\mathscr{J}_1| + |\mathscr{J}_2| + \cdots + |\mathscr{J}_q| = n$ volte per ogni capacità k. Supponendo che la condizione alla riga 10 sia sempre vera:
  - Le righe 2 e 17 vengono eseguite 1 volta ciascuna.
  - Le righe 3 e 4 vengono eseguite $c + 1$ volte ciascuna.
  - La riga 6 viene eseguita q volte.
  - Le righe 7, 8 e 14 vengono eseguite $q(c + 1)$ volte ciascuna.
  - Le righe 9, 10 e 11 vengono eseguite $n(c + 1)$ volte ciascuna.
- Quindi il numero totale di passi nel caso peggiore è

$$S(n, q, c) = 3n(c + 1) + 3q(c + 1) + q + 2c + 4.$$

  Poiché ogni classe contiene almeno un oggetto, $1 \leq q \leq n$, e

$$3n(c + 1) \;\leq\; S(n, q, c) \;\leq\; 6n(c + 1) + n + 2c + 4.$$

  Entrambi i bound sono $\Theta(nc)$, quindi $S(n, q, c) = \Theta(nc)$.

  L'algoritmo di PD per l'MCKP esegue

$$S(n, q, c) = \Theta\big(c\,(|\mathscr{J}_1| + |\mathscr{J}_2| + \cdots + |\mathscr{J}_q|)\big) = \Theta(nc)$$

  passi nel caso peggiore.
- La ricostruzione esegue $\Theta(q)$ passi, uno per ogni classe.

6.2 Errori tipici

Attenzione

- **Far partire m da 0 invece che da $-\infty$.** Con $m = 0$ una classe senza oggetti che entrano nello zaino verrebbe saltata, e la soluzione non conterrebbe esattamente un oggetto per classe.
- **Aggiungere l'opzione «nessun oggetto della classe h».** Il termine $v(h-1,k)$ della ricorrenza del KP qui non ha posto: ogni classe dà esattamente un oggetto.
- **Porre $v(0,k) = -\infty$ per $k \geq 1$.** Senza classi il sottoinsieme vuoto è ammissibile per ogni capacità: la capacità non deve essere riempita.
- **Ricostruire un'istanza inammissibile.** Se $v(q,c) = -\infty$ l'istanza non ha soluzioni ammissibili, e $s[q][c] = 0$ non dà alcun oggetto da selezionare.

6.3 In sintesi

In sintesi

- Gli stadi dell'algoritmo di PD sono le classi: $v(h,k) = \max_{j \in \mathscr{J}_h : w_j \leq k} \{p_j + v(h-1, k-w_j)\}$, con $v(0,k) = 0$.
- Il valore $-\infty$ segnala le sottoistanze senza soluzione ammissibile, e l'istanza è inammissibile se e solo se $v(q,c) = -\infty$.
- La matrice s memorizza l'oggetto scelto in ogni classe, e una soluzione ottima si legge all'indietro a partire da (q,c), un oggetto per classe.
- L'algoritmo di PD esegue $\Theta(c \sum_{h=1}^{q} |\mathscr{J}_h|) = \Theta(nc)$ passi.

Quadro riassuntivo dell'esempio dell'MCKP. Esempio 6.1 ($c = 10$, tre classi):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	20	20	valore ottimo, soluzione $S = \{2,4,7\}$

Capitolo 7

PD per il problema dello zaino multidimensionale

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: la ricorrenza $v(j, k)$ del problema dello zaino 0/1 (KP) su oggetti e capacità, la matrice s delle decisioni e la ricostruzione di una soluzione.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema dello zaino multidimensionale, con una capacità c_d e un peso w_{dj} per ogni dimensione d.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- scrivere la ricorrenza $v(j, k_1, \ldots, k_m)$ sugli oggetti e sul vettore delle capacità;
- eseguire a mano l'algoritmo di PD su un'istanza con $m = 2$ dimensioni, una matrice per ogni oggetto;
- ricostruire una soluzione ottima dalle tabelle s;
- contare i $\Theta(m\, n \prod_{d=1}^{m}(c_d + 1))$ passi e spiegare perché il numero di stati cresce esponenzialmente con m.

Il problema di questo capitolo è il problema dello zaino multidimensionale (MDKP): la sua definizione e le sue ipotesi sono nella sua pagina.

7.1 Un algoritmo di PD

- Indichiamo con m il numero di dimensioni, quindi $\mathscr{D} = \{1, 2, \ldots, m\}$, perché nei capitoli di PD la lettera k indica una capacità. L'oggetto j ha il **vettore dei pesi** $\boldsymbol{w}_j = (w_{1j}, w_{2j}, \ldots, w_{mj})$, e lo zaino ha il vettore delle capacità $\boldsymbol{c} = (c_1, c_2, \ldots, c_m)$.

- Per due vettori, $\boldsymbol{w}_j \leq \boldsymbol{k}$ significa $w_{dj} \leq k_d$ per ogni dimensione $d \in \mathscr{D}$.

- Una soluzione si descrive con il vettore binario $(x_1, x_2, \ldots, x_n)$, dove $x_j = 1$ se l'oggetto j è selezionato; è ammissibile se $\sum_{j=1}^{n} w_{dj} x_j \leq c_d$ per ogni $d \in \mathscr{D}$.

Esempio 7.1: Istanza dell'MDKP

Consideriamo uno zaino con $m = 2$ dimensioni e capacità $c_1 = 5$ e $c_2 = 4$, e un insieme $\mathscr{I} = \{1, 2, 3, 4\}$ di 4 oggetti ($n = 4$), con i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	9	8	3	9
w_{1j}	1	4	2	3
w_{2j}	2	1	1	4

Chiamiamo questa istanza dell'MDKP Esempio 7.1.

- Passo 1: calcolo del valore ottimo

Per ogni $j \in \{0, 1, \ldots, n\}$ e ogni vettore $\boldsymbol{k} = (k_1, k_2, \ldots, k_m)$ con $k_d \in \{0, 1, \ldots, c_d\}$ per ogni $d \in \mathscr{D}$, sia $v(j, \boldsymbol{k}) = v(j, k_1, \ldots, k_m)$ il valore ottimo della sottoistanza dell'MDKP data da:

1. uno zaino con capacità k_d in ogni dimensione $d \in \mathscr{D}$,
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 vettore dei pesi $\boldsymbol{w}_h$.

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

$$v(j, \boldsymbol{k}) = \begin{cases} 0 & \text{se } j = 0, \\ v(j-1, \boldsymbol{k}) & \text{se } j \geq 1 \text{ e } w_{dj} > k_d \text{ per qualche } d, \\ \max\left\{ v(j-1, \boldsymbol{k}), \; p_j + v(j-1, \boldsymbol{k} - \boldsymbol{w}_j) \right\} & \text{se } j \geq 1 \text{ e } \boldsymbol{w}_j \leq \boldsymbol{k}. \end{cases} \tag{7.1}$$

- Con $m = 1$ la formula (7.1) è la ricorrenza del KP: l'unica differenza è che la capacità k diventa il vettore $\boldsymbol{k}$, e «l'oggetto j entra nello zaino» significa $\boldsymbol{w}_j \leq \boldsymbol{k}$.
- I casi base sono $v(0, \boldsymbol{k}) = 0$ per ogni vettore $\boldsymbol{k}$, perché nessun oggetto è disponibile.
- La formula genera al più 2 **sottoproblemi indipendenti**: l'oggetto j non è selezionato, e il sottoproblema residuo ha lo stesso vettore $\boldsymbol{k}$, oppure è selezionato, e il sottoproblema residuo ha il vettore $\boldsymbol{k} - \boldsymbol{w}_j$.

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

- Decisioni diverse sugli oggetti $j+1, \ldots, n$ possono lasciare lo stesso vettore di capacità residue, e quindi lo stesso valore $v(j, \boldsymbol{k})$ può servire più volte.

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

- Il valore $v(j, \boldsymbol{k})$ dipende solo da valori della riga $j-1$. Quindi l'**approccio bottom-up** calcola le righe $j = 0, 1, \ldots, n$ in quest'ordine, e in ogni riga tutti i vettori $\boldsymbol{k}$ in un ordine qualsiasi.
- Lo **stato** dell'algoritmo di PD è la coppia $(j, \boldsymbol{k})$: per ogni j la tabella ha una cella per ogni vettore $\boldsymbol{k}$, cioè $\prod_{d=1}^{m}(c_d + 1)$ celle. Con $m = 2$ la riga j è una matrice, con una riga per ogni k_1 e una colonna per ogni k_2.

- Passo 2: calcolo di una soluzione ottima
  - Per ogni oggetto j e vettore $\boldsymbol{k}$ memorizziamo $s[j][\boldsymbol{k}] = 1$ se l'oggetto j è selezionato nella soluzione ottima trovata per $(j, \boldsymbol{k})$, e $s[j][\boldsymbol{k}] = 0$ altrimenti.
  - L'insieme di tutti i vettori $\boldsymbol{k}$ è $\mathscr{C} = \{0, 1, \ldots, c_1\} \times \{0, 1, \ldots, c_2\} \times \cdots \times \{0, 1, \ldots, c_m\}$.

Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, che memorizza le tabelle s insieme alle tabelle v:

Algoritmo

Algorithm: MDKP_DP

Input: Un'istanza dell'MDKP data da n, m, dal vettore $p[1..n]$, dai vettori dei pesi $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ e dal vettore delle capacità $\boldsymbol{c}$

Output: Il valore ottimo dell'istanza dell'MDKP e le tabelle $s[j][\boldsymbol{k}]$

```
1  Function MDKP_DP(n, m, p, w, c)
2      v := new array[0..n][0..c_1] ··· [0..c_m]; s := new array[0..n][0..c_1] ··· [0..c_m]
3      foreach k ∈ 𝒞 do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
6      for j := 1 to n do
7          foreach k ∈ 𝒞 do
8              v[j][k] ← v[j − 1][k]; s[j][k] ← 0
9              if w_j ≤ k e p[j] + v[j − 1][k − w_j] > v[j][k] then
10                 v[j][k] ← p[j] + v[j − 1][k − w_j]; s[j][k] ← 1
11             end
12         end
13     end
14     return v[n][c], s
```

- La struttura è quella dell'algoritmo di PD del KP; il ciclo sulle capacità diventa un ciclo su tutti i vettori $\boldsymbol{k} \in \mathscr{C}$.
- Il test $\boldsymbol{w}_j \leq \boldsymbol{k}$ confronta m coppie di numeri, e il vettore $\boldsymbol{k} - \boldsymbol{w}_j$ richiede m sottrazioni.
- In caso di pareggio l'oggetto j non viene selezionato, perché il valore viene aggiornato solo quando è strettamente maggiore.

Esempio 7.2: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'MDKP dell'Esempio 7.1, con capacità $c_1 = 5$ e $c_2 = 4$. La riga $j = 0$ è $v[0][\boldsymbol{k}] = 0$ per ogni $\boldsymbol{k}$. Per ogni oggetto j calcoliamo la matrice $v[j][k_1][k_2]$, con una riga per ogni $k_1 \in \{0, 1, \ldots, 5\}$ e una colonna per ogni $k_2 \in \{0, 1, \ldots, 4\}$.

Questi sono i calcoli della cella $\boldsymbol{k} = \boldsymbol{c} = (5, 4)$ per ogni oggetto, e delle celle usate dalla ricostruzione:

- $v[1][(5,4)] = \max\{v[0][(5,4)],\ p_1 + v[0][(4,2)]\} = \max\{0,\ 9 + 0\} = 9$
- $v[1][(1,3)] = \max\{v[0][(1,3)],\ p_1 + v[0][(0,1)]\} = \max\{0,\ 9 + 0\} = 9$
- $v[2][(5,4)] = \max\{v[1][(5,4)],\ p_2 + v[1][(1,3)]\} = \max\{9,\ 8 + 9\} = 17$
- $v[3][(5,4)] = \max\{v[2][(5,4)],\ p_3 + v[2][(3,3)]\} = \max\{17,\ 3 + 9\} = 17$
- $v[4][(5,4)] = \max\{v[3][(5,4)],\ p_4 + v[3][(2,0)]\} = \max\{17,\ 9 + 0\} = 17$

Oggetto $j = 1$, con $p_1 = 9$ e $\boldsymbol{w}_1 = (1, 2)$:

$v[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	9	9	9
$k_1 = 2$	0	0	9	9	9
$k_1 = 3$	0	0	9	9	9
$k_1 = 4$	0	0	9	9	9
$k_1 = 5$	0	0	9	9	9

Oggetto $j = 2$, con $p_2 = 8$ e $\boldsymbol{w}_2 = (4, 1)$:

$v[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	9	9	9
$k_1 = 2$	0	0	9	9	9
$k_1 = 3$	0	0	9	9	9
$k_1 = 4$	0	8	9	9	9
$k_1 = 5$	0	8	9	17	17

Oggetto $j = 3$, con $p_3 = 3$ e $\boldsymbol{w}_3 = (2, 1)$:

$v[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	9	9	9
$k_1 = 2$	0	3	9	9	9
$k_1 = 3$	0	3	9	12	12
$k_1 = 4$	0	8	9	12	12
$k_1 = 5$	0	8	9	17	17

Oggetto $j = 4$, con $p_4 = 9$ e $\boldsymbol{w}_4 = (3, 4)$:

$v[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	9	9	9
$k_1 = 2$	0	3	9	9	9
$k_1 = 3$	0	3	9	12	12
$k_1 = 4$	0	8	9	12	12
$k_1 = 5$	0	8	9	17	17

Esempio 7.3: Le tabelle s

Per l'istanza dell'Esempio 7.1, queste sono le matrici $s[j][k_1][k_2]$ calcolate dall'algoritmo di PD; un 1 indica che l'oggetto j è selezionato nella soluzione ottima trovata per (j, k_1, k_2).

Oggetto $j = 1$, con $p_1 = 9$ e $\boldsymbol{w}_1 = (1, 2)$:

$s[1][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	1	1	1
$k_1=2$	0	0	1	1	1
$k_1=3$	0	0	1	1	1
$k_1=4$	0	0	1	1	1
$k_1=5$	0	0	1	1	1

Oggetto $j=2$, con $p_2=8$ e $\boldsymbol{w}_2=(4,1)$:

$s[2][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	0	0	0
$k_1=2$	0	0	0	0	0
$k_1=3$	0	0	0	0	0
$k_1=4$	0	1	0	0	0
$k_1=5$	0	1	0	1	1

Oggetto $j=3$, con $p_3=3$ e $\boldsymbol{w}_3=(2,1)$:

$s[3][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	0	0	0
$k_1=2$	0	1	0	0	0
$k_1=3$	0	1	0	1	1
$k_1=4$	0	0	0	1	1
$k_1=5$	0	0	0	0	0

Oggetto $j=4$, con $p_4=9$ e $\boldsymbol{w}_4=(3,4)$:

$s[4][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	0	0	0
$k_1=2$	0	0	0	0	0
$k_1=3$	0	0	0	0	0
$k_1=4$	0	0	0	0	0
$k_1=5$	0	0	0	0	0

– Calcolate le tabelle s, si ricostruisce una soluzione ottima scorrendo gli oggetti da $j=n$ fino a $j=1$, partendo da $\boldsymbol{k}=\boldsymbol{c}$ e da $x_j=0$ per ogni oggetto j:

$$x_j \leftarrow s[j][\boldsymbol{k}], \qquad \boldsymbol{k} \leftarrow \boldsymbol{k} - s[j][\boldsymbol{k}]\,\boldsymbol{w}_j.$$

Algoritmo

Algorithm: MDKP_Reconstruct

Input: Il numero n di oggetti, il vettore delle capacità $\boldsymbol{c}$, i vettori dei pesi $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ e le tabelle s

Output: Una soluzione ottima dell'istanza dell'MDKP

```
1 Function MDKP_Reconstruct(n, c, w, s)
2     x := new array[1..n]
3     k := c
4     for j := n downto 1 do
5         x[j] ← s[j][k]
6         k ← k − x[j] w_j
7     end
8     return x
```

Esempio 7.4: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 7.1, partiamo da $j = 4$ e $\boldsymbol{k} = (5, 4)$:

Iterazione	j	$\boldsymbol{k}$	$s[j][\boldsymbol{k}]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Azione
–	–	–	–	0	0	0	0	$\boldsymbol{k} = (5, 4)$
1	4	$(5, 4)$	0	0	0	0	0	$x[4] \leftarrow 0$, $\boldsymbol{k} = (5, 4)$
2	3	$(5, 4)$	0	0	0	0	0	$x[3] \leftarrow 0$, $\boldsymbol{k} = (5, 4)$
3	2	$(5, 4)$	1	0	1	0	0	$x[2] \leftarrow 1$, $\boldsymbol{k} \leftarrow (5, 4) - (4, 1) = (1, 3)$
4	1	$(1, 3)$	1	1	1	0	0	$x[1] \leftarrow 1$, $\boldsymbol{k} \leftarrow (1, 3) - (1, 2) = (0, 1)$

Il vettore $x = (1, 1, 0, 0)$ è una soluzione ottima dell'istanza dell'MDKP dell'Esempio 7.1. I suoi pesi sono $1 + 4 = 5 \leq c_1$ e $2 + 1 = 3 \leq c_2$, e il suo valore ottimo è $v(4, 5, 4) = 17$.

Con la sola prima dimensione, il KP con capacità $c_1 = 5$ ha valore ottimo 18, raggiunto dagli oggetti $\{1, 4\}$; il loro peso totale nella seconda dimensione è $2 + 4 = 6 > c_2$. Le due dimensioni vanno controllate contemporaneamente, come fa la formula (7.1).

7.1.1 Numero di passi

- Sia $N = |\mathscr{C}| = \prod_{d=1}^{m}(c_d+1)$ il numero di vettori $\boldsymbol{k}$, e indichiamo con $S(n, m, \boldsymbol{c})$ il numero di passi eseguiti da `MDKP_DP` nel caso peggiore. Le righe 9 e 10 lavorano su vettori di m componenti, quindi contiamo m passi per ciascuna delle loro esecuzioni:
  - Le righe 2 e 14 vengono eseguite 1 volta ciascuna.
  - Le righe 3 e 4 vengono eseguite N volte ciascuna.
  - La riga 6 viene eseguita n volte.
  - Le righe 7 e 8 vengono eseguite nN volte ciascuna.
  - Le righe 9 e 10 vengono eseguite al più nN volte ciascuna, con m passi ciascuna.
- Quindi il numero totale di passi nel caso peggiore è

$$S(n, m, \boldsymbol{c}) = 2m\,nN + 2nN + 2N + n + 2,$$

  e, poiché $2m\,nN \le S(n, m, \boldsymbol{c}) \le 9m\,nN$,

$$S(n, m, \boldsymbol{c}) = \Theta\Big(m\,n\prod_{d=1}^{m}(c_d+1)\Big).$$

> L'algoritmo di PD per l'MDKP esegue $\Theta(m\,n\prod_{d=1}^{m}(c_d+1))$ passi nel caso peggiore. Per un numero m fissato di dimensioni questo è $\Theta(n\prod_{d=1}^{m}c_d)$, un numero pseudo-polinomiale di passi.

- La ricostruzione esegue $\Theta(m\,n)$ passi.

7.1.2 La maledizione della dimensionalità

- Ogni nuova dimensione moltiplica il numero di stati per c_d+1. Se tutte le capacità sono uguali a c, la tabella ha $(n+1)(c+1)^m$ celle: il numero di stati cresce **esponenzialmente** con m.
- La dimensione dell'input cresce solo linearmente con m: una dimensione in più aggiunge $n+1$ numeri, i pesi w_{dj} e la capacità c_d.
- Questa esplosione del numero di stati quando lo stato diventa un vettore si chiama **maledizione della dimensionalità**.

m	celle $(n+1)(c+1)^m$, con $n = 50$ e $c = 100$
1	5 151
2	520 251
3	52 545 351
4	5 307 080 451
5	536 015 125 551

- Con $m = 1$ la tabella ha qualche migliaio di celle; con $m = 5$ ne ha più di $5 \cdot 10^{11}$, troppe per essere memorizzate. L'algoritmo di PD è quindi utile per l'MDKP solo quando m e le capacità sono piccoli.

7.2 Errori tipici

Attenzione

- **Controllare una sola dimensione.** L'oggetto j entra in $\boldsymbol{k}$ solo se $w_{dj} \leq k_d$ per *ogni* dimensione d; una dimensione non basta.
- **Aggiornare una sola capacità.** Quando l'oggetto j è selezionato, tutte le componenti di $\boldsymbol{k}$ diminuiscono, di $\boldsymbol{w}_j$.
- **Risolvere un KP per dimensione.** Il KP con la sola capacità c_d dà un upper bound, ma la sua soluzione ottima può violare un'altra dimensione.
- **Chiamare polinomiale l'algoritmo di PD.** Il suo numero di passi contiene il prodotto delle capacità, che è esponenziale in m e nella dimensione delle capacità.

7.3 In sintesi

In sintesi

- La ricorrenza del KP si estende all'MDKP sostituendo la capacità k con il vettore $\boldsymbol{k} = (k_1, \ldots, k_m)$: $v(j, \boldsymbol{k}) = \max\{v(j-1, \boldsymbol{k}),\ p_j + v(j-1, \boldsymbol{k} - \boldsymbol{w}_j)\}$ quando $\boldsymbol{w}_j \leq \boldsymbol{k}$.
- La tabella ha $(n+1)\prod_{d=1}^{m}(c_d+1)$ celle e l'algoritmo di PD esegue $\Theta(m\,n\prod_{d=1}^{m}(c_d+1))$ passi, cioè $\Theta(n\prod_{d=1}^{m} c_d)$ per m fissato.
- Il numero di stati cresce esponenzialmente con m: la maledizione della dimensionalità.
- Le tabelle s danno una soluzione ottima, letta all'indietro a partire da $(n, \boldsymbol{c})$.

Quadro riassuntivo dell'esempio dell'MDKP. Esempio 7.1 ($\boldsymbol{c} = (5, 4)$, quattro oggetti):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	17	17	valore ottimo, soluzione $x = (1, 1, 0, 0)$

Capitolo 8

PD per il problema dello zaino con setup

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: la ricorrenza $v(j, k)$ del problema dello zaino 0/1 (KP) su oggetti e capacità, la matrice s delle decisioni e la ricostruzione di una soluzione.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema dello zaino con setup, le sue classi, i costi di setup f_h e i pesi di setup s_h.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- scrivere la ricorrenza del KPS: per ogni classe, il valore migliore senza aprirla e il valore migliore dopo averne pagato il setup;
- eseguire a mano l'algoritmo di PD, classe per classe, con una ricorrenza del KP dentro ogni classe aperta;
- ricostruire una soluzione ottima dalle tabelle delle scelte s e o;
- contare i $\Theta(nc)$ passi dell'algoritmo di PD.

Il problema di questo capitolo è il problema dello zaino con setup (KPS): la sua definizione e le sue ipotesi sono nella sua pagina.

8.1 Un algoritmo di PD

- Numeriamo gli oggetti classe per classe, in modo che $\mathscr{J}_h = \{\ell_{h-1}+1, \ldots, \ell_h\}$ per ogni classe h, con $\ell_0 = 0$ e $\ell_q = n$. La classe h ha $t_h = |\mathscr{J}_h| = \ell_h - \ell_{h-1}$ oggetti, e il suo i-esimo oggetto è $j = \ell_{h-1} + i$.

- Una soluzione si descrive con il vettore binario $(x_1, x_2, \ldots, x_n)$. La classe h è **aperta** se almeno uno dei suoi oggetti è selezionato: allora il suo peso di setup s_h si aggiunge al peso totale, e il suo costo di setup f_h si sottrae dal ricavo totale.

Esempio 8.1: Istanza del KPS

Consideriamo uno zaino di capacità $c = 10$ e un insieme $\mathscr{I} = \{1, 2, \ldots, 5\}$ di 5 oggetti ($n = 5$), partizionato in $q = 2$ classi: $\mathscr{J}_1 = \{1, 2\}$, con costo di setup $f_1 = 2$ e peso di setup $s_1 = 2$, e $\mathscr{J}_2 = \{3, 4, 5\}$, con costo di setup $f_2 = 3$ e peso di setup $s_2 = 1$. I ricavi e i pesi sono:

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
p_j	6	5	4	7	3
w_j	3	2	2	4	1

Chiamiamo questa istanza del KPS Esempio 8.1.

- Passo 1: calcolo del valore ottimo
  - Le classi sono gli stadi dell'algoritmo di PD. Per ogni classe h ci sono due opzioni: la classe h resta chiusa, oppure la classe h viene aperta, se ne paga il setup e si selezionano alcuni dei suoi oggetti.
  - Dentro una classe aperta, la scelta degli oggetti è un KP: l'algoritmo di PD del KP scorre gli oggetti della classe, partendo dal valore rimasto dopo aver pagato il setup.

Per ogni $h \in \{0, 1, \ldots, q\}$ e $k \in \{0, 1, \ldots, c\}$, sia $v(h, k)$ il valore ottimo della sottoistanza del KPS con capacità k e le classi $\mathscr{J}_1, \ldots, \mathscr{J}_h$. Per ogni $h \in \{1, \ldots, q\}$ e $i \in \{0, 1, \ldots, t_h\}$, sia $u(h, i, k)$ il valore ottimo della stessa sottoistanza quando la classe h è aperta (il suo setup è pagato) e sono disponibili solo i suoi primi i oggetti, con $u(h, i, k) = -\infty$ se il peso di setup non entra nello zaino.

Le formule ricorsive per calcolare $u(h, i, k)$ e $v(h, k)$ sono le seguenti, dove $j = \ell_{h-1} + i$ è l'i-esimo oggetto della classe h:

$$u(h,i,k) = \begin{cases} -\infty & \text{se } i = 0 \text{ e } k < s_h, \\ v(h-1, k-s_h) - f_h & \text{se } i = 0 \text{ e } k \geq s_h, \\ u(h, i-1, k) & \text{se } i \geq 1 \text{ e } w_j > k, \\ \max\Big\{u(h,i-1,k), & \text{se } i \geq 1 \text{ e } w_j \leq k, \\ \quad p_j + u(h, i-1, k-w_j)\Big\} & \end{cases} \tag{8.1}$$

$$v(h,k) = \begin{cases} 0 & \text{se } h = 0, \\ \max\big\{v(h-1,k),\ u(h,t_h,k)\big\} & \text{se } h \geq 1. \end{cases} \tag{8.2}$$

- La riga $u(h, 0, \cdot)$ **paga il setup** della classe h: il peso di setup s_h lascia la capacità $k - s_h$ alle classi $1, 2, \ldots, h-1$, e il costo di setup f_h viene sottratto.
- Le righe $u(h, 1, \cdot), \ldots, u(h, t_h, \cdot)$ sono la ricorrenza del KP sugli oggetti della classe h, con $-\infty + p_j = -\infty$.
- La formula (8.2) confronta le due opzioni per la classe h: chiusa, con valore $v(h-1, k)$, oppure aperta, con valore $u(h, t_h, k)$.
- La riga $u(h, 0, \cdot)$ ammette una classe aperta senza oggetti selezionati, che non è una soluzione del KPS. Questo valore non viene mai scelto in (8.2), perché $f_h \geq 1$ dà

  $$v(h-1, k-s_h) - f_h \ < \ v(h-1, k).$$

  Quindi, quando il massimo in (8.2) è raggiunto solo da $u(h, t_h, k)$, almeno un oggetto della classe h è selezionato.
- Una soluzione ottima per (h, k) si costruisce da una soluzione ottima di una sottoistanza più piccola: $(h-1, k)$ se la classe h è chiusa e, dentro una classe aperta, $(h, i-1, \cdot)$ con o senza l'oggetto j.

  $\rightsquigarrow$ **Sottostruttura ottima verificata!**
- Decisioni diverse sulle classi $h+1, \ldots, q$ possono lasciare la stessa capacità k, e quindi lo stesso valore $v(h, k)$ può servire più volte.

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

  - L'**approccio bottom-up** calcola, per $h = 1, 2, \ldots, q$, prima le righe $u(h, i, \cdot)$ per $i = 0, 1, \ldots, t_h$, e poi la riga $v(h, \cdot)$, ciascuna per k crescente. Il valore ottimo dell'istanza del KPS è $v(q, c)$.

- Passo 2: calcolo di una soluzione ottima
  - Si memorizzano due tabelle delle scelte: $s[h][i][k] = 1$ se l'i-esimo oggetto della classe h è selezionato nel valore ottimo $u[h][i][k]$, e $o[h][k] = 1$ se la classe h è aperta nel valore ottimo $v[h][k]$.
  - Il peso di setup s_h ha un indice; la scelta $s[h][i][k]$ ne ha tre.

  Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, che memorizza le tabelle delle scelte insieme ai valori:

Algoritmo

Algorithm: KPS_DP

Input: Un'istanza del KPS data da q, dalle classi $\mathscr{J}_1, \ldots, \mathscr{J}_q$ numerate come sopra, dai costi di setup f_h e dai pesi di setup s_h, dai vettori $p[1..n]$, $w[1..n]$ e dalla capacità c

Output: Il valore ottimo dell'istanza del KPS e le tabelle delle scelte s e o

```
1  Function KPS_DP(q, J, p, w, c)
2      v, o := new arrays[0..q][0..c]; u, s := new arrays[1..q][0..t_h][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0; o[0][k] ← 0
5      end
6      for h := 1 to q do
7          for k := 0 to c do
8              u[h][0][k] ← −∞
9              if k ≥ s_h then
10                 u[h][0][k] ← v[h − 1][k − s_h] − f_h
11             end
12         end
13         for i := 1 to t_h do
14             j := ℓ_{h−1} + i
15             for k := 0 to c do
16                 u[h][i][k] ← u[h][i − 1][k]; s[h][i][k] ← 0
17                 if w[j] ≤ k e p[j] + u[h][i − 1][k − w[j]] > u[h][i][k] then
18                     u[h][i][k] ← p[j] + u[h][i − 1][k − w[j]]; s[h][i][k] ← 1
19                 end
20             end
21         end
22         for k := 0 to c do
23             v[h][k] ← v[h − 1][k]; o[h][k] ← 0
24             if u[h][t_h][k] > v[h][k] then
25                 v[h][k] ← u[h][t_h][k]; o[h][k] ← 1
26             end
27         end
28     end
29     return v[q][c], s, o
```

– Per ogni classe, il primo ciclo su k paga il setup dove entra nello zaino, i cicli su i e k eseguono l'algoritmo di PD del KP sugli oggetti della classe, e l'ultimo ciclo su k confronta la classe chiusa con quella aperta.

– In caso di pareggio l'oggetto non viene selezionato e la classe resta chiusa, perché i valori vengono aggiornati solo quando sono strettamente maggiori.

Esempio 8.2: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del KPS dell'Esempio 8.1. Le righe si calcolano nell'ordine della tabella qui sotto, ciascuna per $k = 0, 1, \ldots, 10$; questi sono alcuni dei calcoli:

- $u[1][0][1] = -\infty$, perché $k = 1 < s_1 = 2$
- $v[1][7] = \max\{v[0][7],\ u[1][2][7]\} = \max\{0,\ 9\} = 9$
- $v[2][10] = \max\{v[1][10],\ u[2][3][10]\} = \max\{9,\ 11\} = 11$
- $u[2][3][10] = \max\{u[2][2][10],\ p_5 + u[2][2][9]\} = \max\{10,\ 3 + 8\} = 11$
- $u[2][2][9] = \max\{u[2][1][9],\ p_4 + u[2][1][5]\} = \max\{6,\ 7 + 1\} = 8$
- $u[2][1][5] = \max\{u[2][0][5],\ p_3 + u[2][0][3]\} = \max\{0,\ 4 - 3\} = 1$
- $u[2][0][3] = v[1][2] - f_2 = 0 - 3 = -3$
- $v[1][2] = \max\{v[0][2],\ u[1][2][2]\} = \max\{0,\ -2\} = 0$

Questa è la tabella dei valori dopo l'esecuzione dell'algoritmo di PD, riga per riga nell'ordine di calcolo:

Riga	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$v[0][k]$	0	0	0	0	0	0	0	0	0	0	0
$u[1][0][k]$ (setup)	$-\infty$	$-\infty$	-2	-2	-2	-2	-2	-2	-2	-2	-2
$u[1][1][k]$ ($j=1$)	$-\infty$	$-\infty$	-2	-2	-2	4	4	4	4	4	4
$u[1][2][k]$ ($j=2$)	$-\infty$	$-\infty$	-2	-2	3	4	4	9	9	9	9
$v[1][k]$	0	0	0	0	3	4	4	9	9	9	9
$u[2][0][k]$ (setup)	$-\infty$	-3	-3	-3	-3	0	1	1	6	6	6
$u[2][1][k]$ ($j=3$)	$-\infty$	-3	-3	1	1	1	1	4	6	6	10
$u[2][2][k]$ ($j=4$)	$-\infty$	-3	-3	1	1	4	4	8	8	8	10
$u[2][3][k]$ ($j=5$)	$-\infty$	-3	0	1	4	4	7	8	11	11	11
$v[2][k]$	0	0	0	1	4	4	7	9	11	11	11

Il valore ottimo dell'istanza è $v[2][10] = 11$. I valori -2 e -3 nelle righe di setup sono i costi di setup pagati prima di selezionare qualsiasi oggetto.

Esempio 8.3: Le tabelle delle scelte

Per l'istanza dell'Esempio 8.1, queste sono le tabelle delle scelte calcolate dall'algoritmo di PD: le righe $s[h][i][\cdot]$ degli oggetti e le righe $o[h][\cdot]$ delle classi.

Riga	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$s[1][1][k]$ $(j=1)$	0	0	0	0	0	1	1	1	1	1	1
$s[1][2][k]$ $(j=2)$	0	0	0	0	1	0	0	1	1	1	1
$o[1][k]$	0	0	0	0	1	1	1	1	1	1	1
$s[2][1][k]$ $(j=3)$	0	0	0	1	1	1	0	1	0	0	1
$s[2][2][k]$ $(j=4)$	0	0	0	0	0	1	1	1	1	1	0
$s[2][3][k]$ $(j=5)$	0	0	1	0	1	0	1	0	1	1	1
$o[2][k]$	0	0	0	1	1	0	1	0	1	1	1

– Una soluzione ottima si ricostruisce scorrendo le classi da $h = q$ fino a $h = 1$, partendo da $k = c$ e da $x_j = 0$ per ogni oggetto j.

– Se $o[h][k] = 0$, la classe h è chiusa e k non cambia. Se $o[h][k] = 1$, gli oggetti della classe h si scorrono da $i = t_h$ fino a $i = 1$ come nel KP, e alla fine il peso di setup s_h si sottrae da k.

Algoritmo

Algorithm: KPS_Reconstruct

Input: Il numero q di classi, i pesi di setup s_h, il vettore $w[1..n]$, la capacità c e le tabelle delle scelte s e o

Output: Una soluzione ottima dell'istanza del KPS

```
1  Function KPS_Reconstruct(q, w, c, s, o)
2      x := new array[1..n], con x[j] = 0 per ogni j
3      k := c
4      for h := q downto 1 do
5          if o[h][k] = 1 then
6              for i := t_h downto 1 do
7                  j := ℓ_{h-1} + i; x[j] ← s[h][i][k]
8                  k ← k − x[j] w[j]
9              end
10             k ← k − s_h
11         end
12     end
13     return x
```

Esempio 8.4: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 8.1, partiamo da $h = 2$ e $k = 10$:

Iterazione	Stadio	k	Scelta	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	Azione
–	–	–	–	0	0	0	0	0	$k = 10$
1	$h = 2$	10	$o[2][10] = 1$	0	0	0	0	0	classe 2 aperta
2	$j = 5$	10	$s[2][3][10] = 1$	0	0	0	0	1	$k \leftarrow 10 - 1 = 9$
3	$j = 4$	9	$s[2][2][9] = 1$	0	0	0	1	1	$k \leftarrow 9 - 4 = 5$
4	$j = 3$	5	$s[2][1][5] = 1$	0	0	1	1	1	$k \leftarrow 5 - 2 = 3$
–	–	3	–	0	0	1	1	1	setup: $k \leftarrow 3 - s_2 = 2$
5	$h = 1$	2	$o[1][2] = 0$	0	0	1	1	1	classe 1 chiusa

Il vettore $x = (0, 0, 1, 1, 1)$ è una soluzione ottima dell'istanza del KPS dell'Esempio 8.1: solo la classe 2 è aperta, e sono selezionati gli oggetti $\{3, 4, 5\}$. Il suo peso totale è $2 + 4 + 1 + s_2 = 8 \leq c$ e il suo valore ottimo è $v(2, 10) = 4 + 7 + 3 - f_2 = 11$.

Gli oggetti della classe 1 hanno il rapporto maggiore fra ricavo totale e peso totale (11/5 contro 14/7), ma la classe 1 resta chiusa. Con la sola classe 1 il valore migliore è $v[1][10] = 9$, mentre la classe 2 aperta dà $u[2][3][10] = 11$; dopo i suoi oggetti e il suo peso di setup resta solo $k = 2$ per la classe 1, e $o[1][2] = 0$.

8.1.1 Numero di passi

- Indichiamo con $S(n, q, c)$ il numero di passi eseguiti da `KPS_DP` nel caso peggiore. I cicli su i scorrono gli oggetti di ogni classe, quindi su tutte le classi vengono eseguiti $t_1 + t_2 + \cdots + t_q = n$ volte. Supponendo che le condizioni alle righe 9, 17 e 24 siano sempre vere:
  - Le righe 2 e 29 vengono eseguite 1 volta ciascuna.
  - Le righe 3 e 4 vengono eseguite $c + 1$ volte ciascuna.
  - La riga 6 viene eseguita q volte.
  - Le righe 7, 8, 9, 10, 22, 23, 24 e 25 vengono eseguite $q(c + 1)$ volte ciascuna.
  - Le righe 13 e 14 vengono eseguite n volte ciascuna.
  - Le righe 15, 16, 17 e 18 vengono eseguite $n(c + 1)$ volte ciascuna.
- Quindi il numero totale di passi nel caso peggiore è

$$S(n, q, c) = 4n(c + 1) + 8q(c + 1) + 2n + q + 2c + 4.$$

  Poiché ogni classe contiene almeno un oggetto, $1 \leq q \leq n$, e

$$4n(c + 1) \leq S(n, q, c) \leq 12n(c + 1) + 3n + 2c + 4.$$

  Entrambi i bound sono $\Theta(nc)$, quindi $S(n, q, c) = \Theta(nc)$.

  L'algoritmo di PD per il KPS esegue $S(n, q, c) = \Theta(nc)$ passi nel caso peggiore: i setup aggiungono $2q$ righe alle n righe della ricorrenza del KP.

- La ricostruzione esegue $\Theta(n)$ passi.

8.2 Errori tipici

Attenzione

- **Pagare il setup per ogni oggetto.** Il setup della classe h si paga una volta sola, nella riga $u(h, 0, \cdot)$; le righe degli oggetti aggiungono solo p_j e w_j.
- **Far partire gli oggetti di una classe da $v(h-1,k)$.** Le righe di una classe aperta partono da $v(h-1, k-s_h) - f_h$: partire da $v(h-1,k)$ selezionerebbe oggetti senza pagare il setup.
- **Dimenticare il peso di setup nella ricostruzione.** Dopo gli oggetti di una classe aperta, la capacità diminuisce anche di s_h.
- **Confondere s_h con $s[h][i][k]$.** Il peso di setup della classe h ha un indice; la scelta dell'i-esimo oggetto della classe h ne ha tre.

8.3 In sintesi

In sintesi

- Per ogni classe h l'algoritmo di PD confronta la classe chiusa, $v(h-1,k)$, con la classe aperta, $u(h, t_h, k)$.
- La classe aperta parte da $u(h,0,k) = v(h-1, k-s_h) - f_h$, che paga il setup una volta sola, e prosegue con la ricorrenza del KP sui suoi oggetti.
- Le tabelle delle scelte s (oggetti) e o (classi) danno una soluzione ottima, letta all'indietro a partire da (q, c).
- L'algoritmo di PD esegue $\Theta(nc)$ passi, come l'algoritmo di PD del KP.

Quadro riassuntivo dell'esempio del KPS. Esempio 8.1 ($c = 10$, due classi):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	11	11	valore ottimo, soluzione $x = (0,0,1,1,1)$

Capitolo 9

PD per il problema degli zaini multipli

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: la ricorrenza $v(j, k)$ del problema dello zaino 0/1 (KP) su oggetti e capacità, la matrice s delle decisioni e la ricostruzione di una soluzione.
- Dal modulo *Problemi di ottimizzazione combinatoria*, il capitolo «Problemi di zaino»: il problema degli zaini multipli, con m zaini di capacità $c_1, \ldots, c_m$.
- Dal modulo *Algoritmi*: le convenzioni dello pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Obiettivi

Al termine di questo capitolo saremo in grado di:

- scrivere la ricorrenza $v(j, k_1, \ldots, k_m)$ sugli oggetti e sulle capacità residue degli m zaini;
- eseguire a mano l'algoritmo di PD su un'istanza con $m = 2$ zaini, una matrice per ogni oggetto;
- ricostruire dalle tabelle s una soluzione ottima, con lo zaino di ogni oggetto selezionato;
- contare i $\Theta(m\, n \prod_{i=1}^{m}(c_i + 1))$ passi e spiegare perché il numero di stati cresce esponenzialmente con m.

Il problema di questo capitolo è il problema degli zaini multipli (MKP): la sua definizione e le sue ipotesi sono nella sua pagina.

9.1 Un algoritmo di PD

- Indichiamo con $i \in \mathscr{K} = \{1, 2, \ldots, m\}$ uno zaino generico, perché nei capitoli di PD la lettera k indica una capacità. Lo zaino i ha capacità c_i.
- Una soluzione assegna ogni oggetto j a uno zaino oppure lo esclude: si descrive con il vettore $(a_1, a_2, \ldots, a_n)$, dove $a_j = i$ se l'oggetto j è inserito nello zaino i, e $a_j = 0$ se l'oggetto j non è selezionato.
- Il vettore $\boldsymbol{e}_i$ è l'i-esimo vettore unitario di lunghezza m: $\boldsymbol{k} - w_j\boldsymbol{e}_i$ è il vettore $\boldsymbol{k}$ con la componente k_i diminuita di w_j.

Esempio 9.1: Istanza dell'MKP

Consideriamo $m = 2$ zaini con capacità $c_1 = 5$ e $c_2 = 4$, e un insieme $\mathscr{I} = \{1, 2, 3, 4\}$ di 4 oggetti ($n = 4$), con i seguenti profitti e pesi:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	3	9	7	5
w_j	3	4	4	2

Chiamiamo questa istanza dell'MKP Esempio 9.1.

- Passo 1: calcolo del valore ottimo

Per ogni $j \in \{0, 1, \ldots, n\}$ e ogni vettore $\boldsymbol{k} = (k_1, k_2, \ldots, k_m)$ con $k_i \in \{0, 1, \ldots, c_i\}$ per ogni $i \in \mathscr{K}$, sia $v(j, \boldsymbol{k}) = v(j, k_1, \ldots, k_m)$ il valore ottimo della sottoistanza dell'MKP data da:

1. m zaini, lo zaino i con capacità k_i,
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 formula ricorsiva per calcolare $v(j, \boldsymbol{k})$ è la seguente:

$$v(j, \boldsymbol{k}) = \begin{cases} 0 & \text{se } j = 0, \\ v(j-1, \boldsymbol{k}) & \text{se } j \geq 1 \text{ e } w_j > k_i \text{ per ogni } i, \\ \max\left\{v(j-1, \boldsymbol{k}), \; p_j + \max\limits_{i:\, w_j \leq k_i} v(j-1, \boldsymbol{k} - w_j\boldsymbol{e}_i)\right\} & \text{se } j \geq 1 \text{ e } w_j \leq k_i \text{ per qualche } i. \end{cases} \tag{9.1}$$

- Le componenti di $\boldsymbol{k}$ sono le **capacità residue** degli zaini: nella visione top-down, le decisioni sugli oggetti $j+1,\ldots,n$ hanno già usato $c_i - k_i$ unità dello zaino i.
- Con $m = 1$ la formula (9.1) è la ricorrenza del KP.
- I casi base sono $v(0, \boldsymbol{k}) = 0$ per ogni vettore $\boldsymbol{k}$, perché nessun oggetto è disponibile.
- La formula genera al più $m+1$ **sottoproblemi indipendenti**: l'oggetto j viene escluso, oppure viene inserito in uno degli zaini i con $w_j \leq k_i$, e allora diminuisce solo la capacità dello zaino i.

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

- Assegnamenti diversi degli oggetti $j+1,\ldots,n$ possono lasciare le stesse capacità residue, e quindi lo stesso valore $v(j, \boldsymbol{k})$ può servire più volte.

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

- Il valore $v(j, \boldsymbol{k})$ dipende solo da valori della riga $j-1$. Quindi l'**approccio bottom-up** calcola le righe $j = 0, 1, \ldots, n$ in quest'ordine, e in ogni riga tutti i vettori $\boldsymbol{k}$ in un ordine qualsiasi. Con $m = 2$ la riga j è una matrice, con una riga per ogni k_1 e una colonna per ogni k_2.

- Passo 2: calcolo di una soluzione ottima
  - Per ogni oggetto j e vettore $\boldsymbol{k}$ memorizziamo $s[j][\boldsymbol{k}] = i$ se l'oggetto j è inserito nello zaino i nella soluzione ottima trovata per $(j, \boldsymbol{k})$, e $s[j][\boldsymbol{k}] = 0$ se l'oggetto j è escluso.
  - L'insieme di tutti i vettori $\boldsymbol{k}$ è $\mathscr{C} = \{0, 1, \ldots, c_1\} \times \{0, 1, \ldots, c_2\} \times \cdots \times \{0, 1, \ldots, c_m\}$.

Questo è lo pseudocodice dell'**algoritmo di PD con l'approccio bottom-up**, che memorizza le tabelle s insieme alle tabelle v:

Algoritmo

Algorithm: MKP_DP

Input: Un'istanza dell'MKP data da n, m, dai vettori $p[1..n]$, $w[1..n]$ e dal vettore delle capacità $\boldsymbol{c}$

Output: Il valore ottimo dell'istanza dell'MKP e le tabelle $s[j][\boldsymbol{k}]$

```
1  Function MKP_DP(n, m, p, w, c)
2      v := new array[0..n][0..c_1] ··· [0..c_m]; s := new array[0..n][0..c_1] ··· [0..c_m]
3      foreach k ∈ 𝒞 do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
6      for j := 1 to n do
7          foreach k ∈ 𝒞 do
8              v[j][k] ← v[j − 1][k]; s[j][k] ← 0
9              for i := 1 to m do
10                 if w[j] ≤ k_i e p[j] + v[j − 1][k − w[j] e_i] > v[j][k] then
11                     v[j][k] ← p[j] + v[j − 1][k − w[j] e_i]; s[j][k] ← i
12                 end
13             end
14         end
15     end
16     return v[n][c], s
```

- Per ogni vettore $\boldsymbol{k}$, l'algoritmo di PD prima esclude l'oggetto j, e poi prova a inserirlo in ogni zaino $i = 1, 2, \ldots, m$ in cui entra.
- In caso di pareggio si conserva la prima opzione incontrata: escludere l'oggetto j, poi lo zaino di indice minimo, perché il valore viene aggiornato solo quando è strettamente maggiore.

Esempio 9.2: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'MKP dell'Esempio 9.1, con capacità $c_1 = 5$ e $c_2 = 4$. La riga $j = 0$ è $v[0][\boldsymbol{k}] = 0$ per ogni $\boldsymbol{k}$. Per ogni oggetto j calcoliamo la matrice $v[j][k_1][k_2]$, con una riga per ogni $k_1 \in \{0, 1, \ldots, 5\}$ e una colonna per ogni $k_2 \in \{0, 1, \ldots, 4\}$.

Questi sono i calcoli della cella $\boldsymbol{k} = \boldsymbol{c} = (5, 4)$ per ogni oggetto, e delle celle usate dalla ricostruzione:

- $v[1][(5,4)] = \max\{v[0][(5,4)],\ p_1 + v[0][(2,4)],\ p_1 + v[0][(5,1)]\} = \max\{0,\ 3 + 0,\ 3 + 0\} = 3$
- $v[1][(3,0)] = \max\{v[0][(3,0)],\ p_1 + v[0][(0,0)]\} = \max\{0,\ 3 + 0\} = 3$
- $v[2][(5,4)] = \max\{v[1][(5,4)],\ p_2 + v[1][(1,4)],\ p_2 + v[1][(5,0)]\} = \max\{3,\ 9 + 3,\ 9 + 3\} = 12$

- $v[2][(3,4)] = \max\{v[1][(3,4)],\ p_2 + v[1][(3,0)]\} = \max\{3,\ 9+3\} = 12$
- $v[3][(5,4)] = \max\{v[2][(5,4)],\ p_3 + v[2][(1,4)],\ p_3 + v[2][(5,0)]\} = \max\{12,\ 7+9,\ 7+9\} = 16$
- $v[3][(3,4)] = \max\{v[2][(3,4)],\ p_3 + v[2][(3,0)]\} = \max\{12,\ 7+3\} = 12$
- $v[4][(5,4)] = \max\{v[3][(5,4)],\ p_4 + v[3][(3,4)],\ p_4 + v[3][(5,2)]\} = \max\{16,\ 5+12,\ 5+9\} = 17$

Oggetto $j = 1$, con $p_1 = 3$ e $w_1 = 3$:

$v[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	3
$k_1 = 1$	0	0	0	3	3
$k_1 = 2$	0	0	0	3	3
$k_1 = 3$	3	3	3	3	3
$k_1 = 4$	3	3	3	3	3
$k_1 = 5$	3	3	3	3	3

Oggetto $j = 2$, con $p_2 = 9$ e $w_2 = 4$:

$v[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	9
$k_1 = 1$	0	0	0	3	9
$k_1 = 2$	0	0	0	3	9
$k_1 = 3$	3	3	3	3	12
$k_1 = 4$	9	9	9	12	12
$k_1 = 5$	9	9	9	12	12

Oggetto $j = 3$, con $p_3 = 7$ e $w_3 = 4$:

$v[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	9
$k_1 = 1$	0	0	0	3	9
$k_1 = 2$	0	0	0	3	9
$k_1 = 3$	3	3	3	3	12
$k_1 = 4$	9	9	9	12	16
$k_1 = 5$	9	9	9	12	16

Oggetto $j = 4$, con $p_4 = 5$ e $w_4 = 2$:

$v[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	5	5	9
$k_1 = 1$	0	0	5	5	9
$k_1 = 2$	5	5	5	8	14
$k_1 = 3$	5	5	8	8	14
$k_1 = 4$	9	9	14	14	16
$k_1 = 5$	9	9	14	14	17

Esempio 9.3: Le tabelle s

Per l'istanza dell'Esempio 9.1, queste sono le matrici $s[j][k_1][k_2]$ calcolate dall'algoritmo di PD: 0 indica che l'oggetto j è escluso, 1 e 2 lo zaino in cui è inserito.

Oggetto $j = 1$, con $p_1 = 3$ e $w_1 = 3$:

$s[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	2	2
$k_1 = 1$	0	0	0	2	2
$k_1 = 2$	0	0	0	2	2
$k_1 = 3$	1	1	1	1	1
$k_1 = 4$	1	1	1	1	1
$k_1 = 5$	1	1	1	1	1

Oggetto $j = 2$, con $p_2 = 9$ e $w_2 = 4$:

$s[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	2
$k_1 = 1$	0	0	0	0	2
$k_1 = 2$	0	0	0	0	2
$k_1 = 3$	0	0	0	0	2
$k_1 = 4$	1	1	1	1	1
$k_1 = 5$	1	1	1	1	1

Oggetto $j = 3$, con $p_3 = 7$ e $w_3 = 4$:

$s[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	0	0	0
$k_1 = 2$	0	0	0	0	0
$k_1 = 3$	0	0	0	0	0
$k_1 = 4$	0	0	0	0	1
$k_1 = 5$	0	0	0	0	1

Oggetto $j = 4$, con $p_4 = 5$ e $w_4 = 2$:

$s[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	2	2	0
$k_1 = 1$	0	0	2	2	0
$k_1 = 2$	1	1	1	1	1
$k_1 = 3$	1	1	2	1	1
$k_1 = 4$	0	0	2	2	0
$k_1 = 5$	0	0	2	2	1

– Calcolate le tabelle s, si ricostruisce una soluzione ottima scorrendo gli oggetti da $j = n$ fino a $j = 1$, partendo da $\boldsymbol{k} = \boldsymbol{c}$ e da $a_j = 0$ per ogni oggetto j:

$$a_j \leftarrow s[j][\boldsymbol{k}], \qquad \text{e, se } a_j = i \geq 1, \quad \boldsymbol{k} \leftarrow \boldsymbol{k} - w_j \boldsymbol{e}_i.$$

Algoritmo

Algorithm: MKP_Reconstruct

Input: Il numero n di oggetti, il vettore delle capacità $\boldsymbol{c}$, il vettore $w[1..n]$ e le tabelle s

Output: Una soluzione ottima dell'istanza dell'MKP, come vettore degli zaini

```
1  Function MKP_Reconstruct(n, c, w, s)
2      a := new array[1..n]
3      k := c
4      for j := n downto 1 do
5          a[j] ← s[j][k]
6          if a[j] ≥ 1 then
7              k ← k − w[j] e_{a[j]}
8          end
9      end
10     return a
```

Esempio 9.4: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 9.1, partiamo da $j = 4$ e $\boldsymbol{k} = (5, 4)$:

Iterazione	j	$\boldsymbol{k}$	$s[j][\boldsymbol{k}]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
–	–	–	–	0	0	0	0	$\boldsymbol{k} = (5, 4)$
1	4	$(5, 4)$	1	0	0	0	1	$a[4] \leftarrow 1$, $\boldsymbol{k} \leftarrow (3, 4)$
2	3	$(3, 4)$	0	0	0	0	1	$a[3] \leftarrow 0$, $\boldsymbol{k} = (3, 4)$
3	2	$(3, 4)$	2	0	2	0	1	$a[2] \leftarrow 2$, $\boldsymbol{k} \leftarrow (3, 0)$
4	1	$(3, 0)$	1	1	2	0	1	$a[1] \leftarrow 1$, $\boldsymbol{k} \leftarrow (0, 0)$

Il vettore $a = (1, 2, 0, 1)$ è una soluzione ottima dell'istanza dell'MKP dell'Esempio 9.1: lo zaino 1 contiene gli oggetti $\{1, 4\}$, di peso totale $3+2 = 5 \leq c_1$, e lo zaino 2 contiene gli oggetti $\{2\}$, di peso totale $4 \leq c_2$. Il valore ottimo è $v(4, 5, 4) = 17$.

Riempire gli zaini uno dopo l'altro non basta: il miglior riempimento del solo zaino 1 è $\{2\}$, di profitto 9, e il miglior riempimento dello zaino 2 con gli oggetti rimasti è $\{3\}$, di profitto 7, per un totale di $16 < 17$.

9.1.1 Numero di passi

- Sia $N = |\mathscr{C}| = \prod_{i=1}^{m}(c_i + 1)$ il numero di vettori $\boldsymbol{k}$, e indichiamo con $S(n, m, \boldsymbol{c})$ il numero di passi eseguiti da `MKP_DP` nel caso peggiore. Inserire l'oggetto j nello zaino i cambia solo la componente k_i, quindi le righe 10 e 11 contano un passo ciascuna:
  - Le righe 2 e 16 vengono eseguite 1 volta ciascuna.
  - Le righe 3 e 4 vengono eseguite N volte ciascuna.
  - La riga 6 viene eseguita n volte.
  - Le righe 7 e 8 vengono eseguite nN volte ciascuna.
  - Le righe 9, 10 e 11 vengono eseguite al più $m\,nN$ volte ciascuna.
- Quindi il numero totale di passi nel caso peggiore è

$$S(n, m, \boldsymbol{c}) = 3m\,nN + 2nN + 2N + n + 2,$$

  e, poiché $3m\,nN \le S(n, m, \boldsymbol{c}) \le 10m\,nN$,

$$S(n, m, \boldsymbol{c}) = \Theta\Big(m\,n\prod_{i=1}^{m}(c_i + 1)\Big).$$

> L'algoritmo di PD per l'MKP esegue $\Theta(m\,n\prod_{i=1}^{m}(c_i+1))$ passi nel caso peggiore. Per un numero m fissato di zaini questo è $\Theta(n\prod_{i=1}^{m} c_i)$, un numero pseudo-polinomiale di passi.

- La ricostruzione esegue $\Theta(n)$ passi.

9.1.2 La crescita con il numero di zaini

- Ogni nuovo zaino moltiplica il numero di stati per $c_i + 1$. Se tutte le capacità sono uguali a c, la tabella ha $(n + 1)(c + 1)^m$ celle: il numero di stati cresce **esponenzialmente** con m, mentre la dimensione dell'input cresce di un solo numero, la capacità c_i.
- Con $n = 50$ oggetti e capacità $c_i = 100$, la tabella ha $51 \cdot 101 = 5151$ celle con $m = 1$, $51 \cdot 101^2 = 520\,251$ con $m = 2$, e $51 \cdot 101^5 \approx 5.4 \cdot 10^{11}$ con $m = 5$.
- Zaini con la stessa capacità danno stati simmetrici: se $c_1 = c_2$, i vettori $(k_1, k_2, \dots)$ e $(k_2, k_1, \dots)$ hanno lo stesso valore ottimo, e l'algoritmo di PD li calcola entrambi.

9.2 Errori tipici

Attenzione

- **Diminuire tutte le capacità.** L'oggetto j va in un solo zaino: $\boldsymbol{k} - w_j \boldsymbol{e}_i$ diminuisce la componente k_i, non tutte come nel problema dello zaino multidimensionale.
- **Usare un'unica capacità $c_1 + \cdots + c_m$.** Il KP con la capacità totale dà solo un upper bound: la sua soluzione può non essere divisibile fra gli zaini.
- **Riempire gli zaini uno dopo l'altro.** Un riempimento ottimo dello zaino 1, seguito da un riempimento ottimo dello zaino 2 con gli oggetti rimasti, può essere peggiore dell'ottimo.
- **Dimenticare l'opzione «oggetto escluso».** Il termine $v(j-1, \boldsymbol{k})$ è sempre un candidato, anche quando l'oggetto j entra in qualche zaino.

9.3 In sintesi

In sintesi

- Lo stato dell'algoritmo di PD è dato dall'oggetto j e dal vettore $\boldsymbol{k}$ delle capacità residue degli m zaini.
- L'oggetto j viene escluso o inserito in uno zaino i in cui entra: $m+1$ opzioni, e diminuisce solo k_i.
- L'algoritmo di PD esegue $\Theta(m\, n \prod_{i=1}^{m}(c_i+1))$ passi: il numero di stati cresce esponenzialmente con m.
- Le tabelle s memorizzano lo zaino di ogni oggetto, e una soluzione ottima si legge all'indietro a partire da $(n, \boldsymbol{c})$.

Quadro riassuntivo dell'esempio dell'MKP. Esempio 9.1 ($\boldsymbol{c} = (5, 4)$, quattro oggetti):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	17	17	valore ottimo, soluzione $a = (1, 2, 0, 1)$

Problemi di lot sizing

Capitolo 10

PD per il problema del lot sizing senza capacità

Prerequisiti

- Dal modulo *Problemi di ottimizzazione combinatoria*, i capitoli «Problemi di lot sizing» e «Formulazioni dei problemi di lot sizing»: il problema a un solo prodotto, la sua istanza di riferimento e il suo modello PLI con le variabili x_t, s_t, y_t.
- Da questo modulo, il capitolo «Principi della programmazione dinamica» e gli algoritmi di PD dei capitoli precedenti: ricorrenza, casi base, tabella di lookup e ricostruzione di una soluzione.
- Dal modulo *Algoritmi*: lo pseudocodice (Capitolo 2) e la dimensione dell'input (Capitolo 11).

Dove eravamo

- **Come problema e come modello**: il lot sizing a un solo prodotto e le sue formulazioni PLI (capitoli «Problemi di lot sizing» e «Formulazioni dei problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*).
- **Ora, come tabella**: l'algoritmo di PD di Wagner e Whitin dà il valore ottimo del problema senza capacità in $\Theta(m^2)$ passi.

Obiettivi

Al termine di questo capitolo saremo in grado di:

- enunciare la proprietà zero-inventory ordering e dimostrarla con l'argomento di scambio;
- calcolare il costo $c_{j,t}$ di un lotto di produzione, e la matrice dei costi, in $O(m^2)$ passi;
- scrivere la ricorrenza $v(t)$ sull'ultimo lotto di produzione, eseguire a mano l'algoritmo di PD e ricostruire un piano ottimo;
- leggere la ricorrenza come un cammino minimo nella rete dei lotti di produzione;

- contare i passi dell'algoritmo di PD e mostrare che sono $\Theta(m^2)$.

Il problema di questo capitolo è il problema di lot sizing senza capacità (ULS): la sua definizione e le sue ipotesi sono nella sua pagina.

10.1 Notazione, modello e soluzioni ammissibili

- Per ogni $j, t \in T$, indichiamo con
$$D_{j,t} = \sum_{k=j}^{t} d_k$$
la domanda totale dei periodi $j, j+1, \ldots, t$, con $D_{j,t} = 0$ se $j > t$.

- Il modello PLI è il modello ILP_{ULS} del capitolo «Formulazioni dei problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*, sezione «Modelli PLI per le varianti»: il modello ILP_{CLSP} senza i vincoli di capacità e con il valore di big-M $D_{t,m}$ del periodo t, la domanda totale dei periodi rimanenti. Le sue variabili sono la produzione x_t, la scorta s_t alla fine del periodo t e il setup $y_t \in \{0, 1\}$, per ogni periodo $t \in T$.

Richiamiamo il modello ILP_{ULS} del problema di lot sizing senza capacità (ULS):

$$\min \quad \sum_{t \in T} (sc_t\, y_t + vc_t\, x_t + hc_t\, s_t) \tag{10.1a}$$

$$\text{soggetto a} \quad s_{t-1} + x_t = d_t + s_t, \qquad \forall t \in T, \tag{10.1b}$$

$$x_t \le D_{t,m}\, y_t, \qquad \forall t \in T, \tag{10.1c}$$

$$s_0 = s_m = 0, \tag{10.1d}$$

$$x_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{10.1e}$$

$$s_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{10.1f}$$

$$y_t \in \{0, 1\}, \qquad \forall t \in T. \tag{10.1g}$$

Questo modello PLI si chiama modello ILP_{ULS}.

- Descrizione della funzione obiettivo e dei vincoli del modello ILP_{ULS}:
  - la funzione obiettivo (10.1a) minimizza il costo totale di setup, produzione e mantenimento a scorta;
  - i vincoli (10.1b) sono i vincoli di bilanciamento del flusso;
  - i vincoli (10.1c) impongono un setup in ogni periodo con produzione positiva;
  - il vincolo (10.1d) impone una scorta vuota all'inizio e alla fine dell'orizzonte;
  - infine, i vincoli (10.1e)–(10.1g) definiscono le variabili del modello PLI.

Esempio 10.1: Lot sizing senza capacità

Consideriamo $m = 5$ periodi con i seguenti dati:

t	1	2	3	4	5
d_t	20	10	30	40	10
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	1	1	1	1	1

Chiamiamo questa istanza dell'ULS Esempio 10.1. È l'istanza di riferimento del capitolo «Problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*, contata in singoli pezzi invece che in decine e senza le capacità (costo di setup 50, costo di mantenimento a scorta 1, nessun costo variabile di produzione).

10.1.1 Codifica delle soluzioni ammissibili

Una *soluzione ammissibile* dell'ULS è un piano di produzione che soddisfa la domanda di ogni periodo, partendo e terminando con la scorta vuota.

- Una soluzione ammissibile dell'ULS si può descrivere con un vettore soluzione

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

dove ogni componente ha il seguente significato:

$$x_t = \text{numero di unità prodotte nel periodo } t, \qquad \forall t \in \{1, 2, \ldots, m\}.$$

La scorta alla fine del periodo t è determinata dal vettore:

$$s_t = \sum_{j=1}^{t} x_j - D_{1,t}, \qquad \forall t \in \{1, 2, \ldots, m\}.$$

Poiché la soluzione è ammissibile, le scorte sono non negative e l'ultima è vuota:

$$s_t \geq 0 \quad \forall t \in \{1, 2, \ldots, m-1\}, \qquad s_m = 0.$$

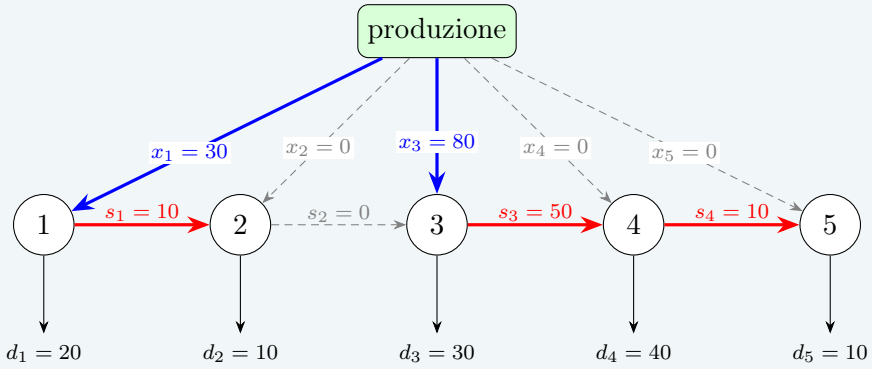
Nel periodo t si paga un setup se e solo se $x_t > 0$. Il costo totale di una soluzione ammissibile è

$$\sum_{t=1,\ x_t>0}^{m} sc_t + \sum_{t=1}^{m} vc_t\, x_t + \sum_{t=1}^{m} hc_t\, s_t.$$

Esempio 10.2: Una soluzione ammissibile

Consideriamo l'istanza dell'ULS dell'Esempio 10.1. La rete seguente rappresenta la soluzione ammissibile $(30, 0, 80, 0, 0)$: ogni periodo è un nodo, la produzione x_t entra nel nodo t, la

domanda d_t ne esce, e la scorta s_t scorre dal nodo t al nodo $t+1$. Gli archi spessi portano un flusso positivo.



I setup sono nei periodi 1, 3, le scorte sono $(s_1, \dots, s_5) = (10, 0, 50, 10, 0)$, e il costo totale è (non c'è costo variabile di produzione)

$$\underbrace{50 + 50}_{\text{setup}} + \underbrace{1 \cdot 10 + 1 \cdot 0 + 1 \cdot 50 + 1 \cdot 10 + 1 \cdot 0}_{\text{mantenimento}} = 100 + 70 = 170.$$

- Una quantità prodotta può assumere molti valori, e un'istanza dell'ULS ha un numero enorme di soluzioni ammissibili (infinite, se non si richiede che le quantità siano intere). La proprietà seguente mostra che basta considerarne pochissime.

10.1.2 La proprietà zero-inventory ordering

Teorema 10.1: Zero-inventory ordering (Wagner e Whitin, 1958)

Ogni istanza dell'ULS ha una soluzione ottima in cui nessun periodo con produzione positiva inizia con una scorta positiva, cioè

$$s_{t-1}\, x_t = 0 \qquad \forall t \in \{1, 2, \dots, m\}.$$

- *Dimostrazione.* Anzitutto, una soluzione ottima esiste: il vettore $(d_1, d_2, \dots, d_m)$ è ammissibile e, per ciascuna delle 2^m scelte dell'insieme dei periodi con setup, il problema che resta è un problema LP su un poliedro limitato (poiché $s_m = 0$, ogni x_t è compreso fra 0 e $D_{1,m}$), quindi ha una soluzione ottima ogni volta che è ammissibile.

- Fra tutte le soluzioni ottime, consideriamone una con il **minimo numero di periodi con produzione positiva**, e supponiamo, per assurdo, che esista un periodo t con $x_t > 0$ e $s_{t-1} > 0$.

- Poiché $s_0 = 0$ e $s_{t-1} > 0$, qualcosa viene prodotto prima del periodo t: sia $j < t$ l'ultimo periodo prima di t con $x_j > 0$. Nei periodi $j+1, \dots, t-1$ non si produce alcuna unità, quindi la scorta può solo diminuire dal periodo j al periodo $t-1$:

$$s_k \geq s_{t-1} > 0 \qquad \forall k \in \{j, j+1, \dots, t-1\}.$$

- Per un numero ε, spostiamo ε unità di produzione dal periodo j al periodo t: sostituiamo x_j con $x_j - \varepsilon$, x_t con $x_t + \varepsilon$ e s_k con $s_k - \varepsilon$ per ogni $k \in \{j, \dots, t-1\}$. Tutti i vincoli di bilanciamento restano soddisfatti, e il nuovo piano è ammissibile per ogni ε con

$$-x_t \leq \varepsilon \leq \min\{x_j,\ s_{t-1}\}.$$

- Finché x_j e x_t restano positivi, i setup non cambiano e il costo varia di $\varepsilon\,\Delta$, dove

$$\Delta = vc_t - vc_j - \sum_{k=j}^{t-1} hc_k$$

è il costo di produrre un'unità nel periodo t meno il costo di produrla nel periodo j e tenerla a scorta fino al periodo t.

- Se $\Delta < 0$, un $\varepsilon > 0$ piccolo dà una soluzione ammissibile di costo minore, il che contraddice l'ottimalità.

- Se $\Delta \geq 0$, scegliamo $\varepsilon = -x_t$: tutta la produzione del periodo t viene spostata nel periodo j, il costo varia di $-x_t\,\Delta \leq 0$, e si risparmia il costo di setup $sc_t \geq 0$ del periodo t. La nuova soluzione è ottima e ha un periodo con produzione positiva in meno, il che contraddice la scelta della soluzione.

- Quindi un tale periodo t non esiste. □

In una soluzione ottima con la proprietà zero-inventory ordering, ogni periodo j con produzione positiva produce esattamente le domande dei periodi $j, j+1, \dots, t$, dove $t+1$ è il periodo successivo con produzione positiva (oppure $t = m$ se non ce n'è uno):

$$x_j = D_{j,t}.$$

- Infatti $s_{j-1} = 0$ (zero-inventory ordering nel periodo j), $s_t = 0$ (zero-inventory ordering nel periodo $t+1$, oppure $s_m = 0$), e nei periodi $j+1, \dots, t$ non si produce nulla; sommando i vincoli di bilanciamento dei periodi $j, \dots, t$ si ottiene $x_j = D_{j,t}$.

- Il periodo 1 produce sempre, perché $s_0 = 0$ e $d_1 \geq 1$. Una soluzione con la proprietà zero-inventory ordering è quindi determinata dall'insieme dei suoi periodi di produzione, che contiene il periodo 1 e un qualsiasi sottoinsieme di $\{2, 3, \dots, m\}$.

Il numero di soluzioni dell'ULS con la proprietà zero-inventory ordering è 2^{m-1}, perché ogni periodo $t \in \{2, 3, \dots, m\}$ può, indipendentemente dagli altri, produrre o no.

- Il numero di queste soluzioni cresce esponenzialmente con il numero m di periodi:

$$2^{m-1} = O(2^m).$$

Quindi enumerarle diventa rapidamente impraticabile al crescere di m.

Esempio 10.3: Enumerazione delle soluzioni con la proprietà zero-inventory ordering

Consideriamo l'istanza dell'ULS dell'Esempio 10.1:

t	1	2	3	4	5
d_t	20	10	30	40	10
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	1	1	1	1	1

Ci sono $2^{5-1} = 16$ soluzioni con la proprietà zero-inventory ordering. Per ciascuna di esse, la tabella riporta i periodi di produzione, il vettore soluzione, le scorte e il costo totale, scritto come costo di setup più costo di mantenimento a scorta. I costi ottimi sono evidenziati.

Etichetta	Periodi di produzione	Vettore soluzione	Scorte $(s_1, \ldots, s_5)$	Costo totale
(1)	$\{1\}$	$(110, 0, 0, 0, 0)$	$(90, 80, 50, 10, 0)$	$50 + 230 = 280$
(2)	$\{1, 5\}$	$(100, 0, 0, 0, 10)$	$(80, 70, 40, 0, 0)$	$100 + 190 = 290$
(3)	$\{1, 4\}$	$(60, 0, 0, 50, 0)$	$(40, 30, 0, 10, 0)$	$100 + 80 = 180$
(4)	$\{1, 4, 5\}$	$(60, 0, 0, 40, 10)$	$(40, 30, 0, 0, 0)$	$150 + 70 = 220$
(5)	$\{1, 3\}$	$(30, 0, 80, 0, 0)$	$(10, 0, 50, 10, 0)$	$100 + 70 = 170$
(6)	$\{1, 3, 5\}$	$(30, 0, 70, 0, 10)$	$(10, 0, 40, 0, 0)$	$150 + 50 = 200$
(7)	$\{1, 3, 4\}$	$(30, 0, 30, 50, 0)$	$(10, 0, 0, 10, 0)$	$150 + 20 = 170$
(8)	$\{1, 3, 4, 5\}$	$(30, 0, 30, 40, 10)$	$(10, 0, 0, 0, 0)$	$200 + 10 = 210$
(9)	$\{1, 2\}$	$(20, 90, 0, 0, 0)$	$(0, 80, 50, 10, 0)$	$100 + 140 = 240$
(10)	$\{1, 2, 5\}$	$(20, 80, 0, 0, 10)$	$(0, 70, 40, 0, 0)$	$150 + 110 = 260$
(11)	$\{1, 2, 4\}$	$(20, 40, 0, 50, 0)$	$(0, 30, 0, 10, 0)$	$150 + 40 = 190$
(12)	$\{1, 2, 4, 5\}$	$(20, 40, 0, 40, 10)$	$(0, 30, 0, 0, 0)$	$200 + 30 = 230$
(13)	$\{1, 2, 3\}$	$(20, 10, 80, 0, 0)$	$(0, 0, 50, 10, 0)$	$150 + 60 = 210$
(14)	$\{1, 2, 3, 5\}$	$(20, 10, 70, 0, 10)$	$(0, 0, 40, 0, 0)$	$200 + 40 = 240$
(15)	$\{1, 2, 3, 4\}$	$(20, 10, 30, 50, 0)$	$(0, 0, 0, 10, 0)$	$200 + 10 = 210$
(16)	$\{1, 2, 3, 4, 5\}$	$(20, 10, 30, 40, 10)$	$(0, 0, 0, 0, 0)$	$250 + 0 = 250$

Il valore ottimo dell'istanza dell'ULS è 170, raggiunto da 2 soluzioni: (5) e (7).

10.2 Un algoritmo di PD per l'ULS

10.2.1 Il costo di un lotto di produzione

- Per la proprietà zero-inventory ordering, una soluzione ottima è una sequenza di **lotti di produzione**: un lotto produce in un periodo j le domande dei periodi $j, \ldots, t$, e il lotto successivo inizia nel periodo $t+1$ con la scorta vuota.

- Per ogni $j, t \in T$ con $j \leq t$, sia $c_{j,t}$ il costo del lotto di produzione che produce nel periodo j le domande dei periodi $j, \ldots, t$:

$$c_{j,t} = sc_j + vc_j\, D_{j,t} + \sum_{k=j}^{t-1} hc_k\, D_{k+1,t}. \tag{10.2}$$

- Il primo termine è il costo di setup del periodo j, il secondo è il costo variabile delle $D_{j,t}$ unità prodotte, e l'ultimo è il costo di mantenimento a scorta: alla fine di ogni periodo $k \in \{j, \ldots, t-1\}$ la scorta contiene le domande $D_{k+1,t}$ dei periodi successivi del lotto.

- Per un periodo j fissato, i costi $c_{j,j}, c_{j,j+1}, \ldots, c_{j,m}$ si possono calcolare uno dopo l'altro. Un'unità usata nel periodo t e prodotta nel periodo j costa

$$u_{j,t} = vc_j + \sum_{k=j}^{t-1} hc_k,$$

quindi $u_{j,j} = vc_j$, $u_{j,t} = u_{j,t-1} + hc_{t-1}$, e aggiungere il periodo t al lotto aggiunge d_t unità di costo $u_{j,t}$:

$$c_{j,j} = sc_j + vc_j\, d_j, \qquad c_{j,t} = c_{j,t-1} + u_{j,t}\, d_t \quad \text{se } t > j.$$

Questo è lo pseudocodice che calcola la matrice dei costi dei lotti di produzione:

Algoritmo

Algorithm: CostMatrix

Input: Un'istanza dell'ULS data dal numero m di periodi e dai vettori $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$

Output: La matrice dei costi dei lotti di produzione

```
1  Function CostMatrix(m, d, sc, vc, hc)
2      c := new array[1..m][1..m]
3      for j := 1 to m do
4          u := vc[j]
5          g := sc[j] + vc[j] · d[j]
6          c[j][j] ← g
7          for t := j + 1 to m do
8              u ← u + hc[t − 1]
9              g ← g + u · d[t]
10             c[j][t] ← g
11         end
12     end
13     return c
```

- La variabile u è il costo unitario $u_{j,t}$ e la variabile g è il costo $c_{j,t}$ del lotto corrente; si usano solo le celle $c[j][t]$ con $j \leq t$.

Esempio 10.4: Calcolo dei costi dei lotti di produzione

Consideriamo l'istanza dell'ULS dell'Esempio 10.1:

t	1	2	3	4	5
d_t	20	10	30	40	10
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	1	1	1	1	1

Per ogni passo, la tabella riporta i valori di j e t, il valore corrente di u, la cella $c[j][t]$ e l'operazione eseguita; la cella $c[j][t]$ è evidenziata in blu quando viene assegnata.

j	t	u	$c[j][t]$	Azione
1	1	0	50	$u \leftarrow vc[1] = 0$, $g \leftarrow sc[1]+vc[1] \cdot d[1] = 50+0 \cdot 20 = 50$, $c[1][1] \leftarrow 50$
1	2	1	60	$u \leftarrow u+hc[1] = 0+1 = 1$, $g \leftarrow g+u \cdot d[2] = 50+1 \cdot 10 = 60$, $c[1][2] \leftarrow 60$
1	3	2	120	$u \leftarrow u+hc[2] = 1+1 = 2$, $g \leftarrow g+u \cdot d[3] = 60+2 \cdot 30 = 120$, $c[1][3] \leftarrow 120$
1	4	3	240	$u \leftarrow u+hc[3] = 2+1 = 3$, $g \leftarrow g+u \cdot d[4] = 120+3 \cdot 40 = 240$, $c[1][4] \leftarrow 240$
1	5	4	280	$u \leftarrow u+hc[4] = 3+1 = 4$, $g \leftarrow g+u \cdot d[5] = 240+4 \cdot 10 = 280$, $c[1][5] \leftarrow 280$
2	2	0	50	$u \leftarrow vc[2] = 0$, $g \leftarrow sc[2]+vc[2] \cdot d[2] = 50+0 \cdot 10 = 50$, $c[2][2] \leftarrow 50$
2	3	1	80	$u \leftarrow u+hc[2] = 0+1 = 1$, $g \leftarrow g+u \cdot d[3] = 50+1 \cdot 30 = 80$, $c[2][3] \leftarrow 80$
2	4	2	160	$u \leftarrow u+hc[3] = 1+1 = 2$, $g \leftarrow g+u \cdot d[4] = 80+2 \cdot 40 = 160$, $c[2][4] \leftarrow 160$
2	5	3	190	$u \leftarrow u+hc[4] = 2+1 = 3$, $g \leftarrow g+u \cdot d[5] = 160+3 \cdot 10 = 190$, $c[2][5] \leftarrow 190$
3	3	0	50	$u \leftarrow vc[3] = 0$, $g \leftarrow sc[3]+vc[3] \cdot d[3] = 50+0 \cdot 30 = 50$, $c[3][3] \leftarrow 50$
3	4	1	90	$u \leftarrow u+hc[3] = 0+1 = 1$, $g \leftarrow g+u \cdot d[4] = 50+1 \cdot 40 = 90$, $c[3][4] \leftarrow 90$
3	5	2	110	$u \leftarrow u+hc[4] = 1+1 = 2$, $g \leftarrow g+u \cdot d[5] = 90+2 \cdot 10 = 110$, $c[3][5] \leftarrow 110$
4	4	0	50	$u \leftarrow vc[4] = 0$, $g \leftarrow sc[4]+vc[4] \cdot d[4] = 50+0 \cdot 40 = 50$, $c[4][4] \leftarrow 50$
4	5	1	60	$u \leftarrow u+hc[4] = 0+1 = 1$, $g \leftarrow g+u \cdot d[5] = 50+1 \cdot 10 = 60$, $c[4][5] \leftarrow 60$
5	5	0	50	$u \leftarrow vc[5] = 0$, $g \leftarrow sc[5]+vc[5] \cdot d[5] = 50+0 \cdot 10 = 50$, $c[5][5] \leftarrow 50$

Questa è la matrice c dopo l'esecuzione dell'algoritmo (per esempio, $c_{1,3} = sc_1 + vc_1\, D_{1,3} + hc_1\, D_{2,3} + hc_2\, D_{3,3} = 50 + 0 \cdot 60 + 1 \cdot 40 + 1 \cdot 30 = 120$ per la formula (10.2)):

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

10.2.2 Passo 1 e Passo 2

- Passo 1: calcolo del valore ottimo

Per ogni $t \in \{1, 2, \ldots, m\}$, sia $v(t)$ il valore ottimo della sottoistanza dell'ULS data da:

1. un insieme $\{1, 2, \ldots, t\}$ di t periodi di tempo,
2. per ogni periodo $j \in \{1, 2, \ldots, t\}$, una domanda d_j,
3. per ogni periodo $j \in \{1, 2, \ldots, t\}$, un costo di setup sc_j, un costo variabile di produzione vc_j e un costo di mantenimento a scorta hc_j.

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

$$v(t) = \begin{cases} \min\left\{v(j-1) + c_{j,t} : j \in \{1, 2, \ldots, t\}\right\} & \text{se } 1 \leq t \leq m, \\ 0 & \text{se } t = 0. \end{cases} \tag{10.3}$$

- La sottoistanza con i periodi $\{1, \ldots, t\}$ è un'istanza dell'ULS dello stesso tipo: la sua scorta deve essere vuota alla fine del periodo t. Qui $v(0) = 0$, perché un insieme vuoto di periodi non ha domanda né costo.
- Il processo di decomposizione ricorsiva della formula (10.3) genera t **sottoproblemi indipendenti**, ciascuno corrispondente a una possibile scelta del periodo j dell'**ultimo lotto di produzione**, che produce le domande dei periodi $j, \ldots, t$.
- Per la proprietà zero-inventory ordering applicata alla sottoistanza, la scorta è vuota alla fine del periodo $j - 1$, e il sottoproblema residuo è la sottoistanza dell'ULS con i periodi $\{1, \ldots, j - 1\}$.
- Il valore ottimo $v(t)$ si ottiene provando tutti i possibili periodi dell'ultimo lotto di produzione e scegliendo quello che minimizza la somma $v(j - 1) + c_{j,t}$ del valore ottimo del sottoproblema residuo e del costo dell'ultimo lotto.
- Quindi una soluzione ottima per i periodi $\{1, \ldots, t\}$ si costruisce combinando l'ultimo lotto di produzione scelto nel periodo j con una soluzione ottima del sottoproblema residuo con i periodi $\{1, \ldots, j - 1\}$.

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

- Scelte diverse dell'ultimo lotto di produzione, fatte eventualmente per valori diversi di t, possono portare allo stesso insieme residuo di periodi, e quindi lo stesso valore $v(t)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (10.3).

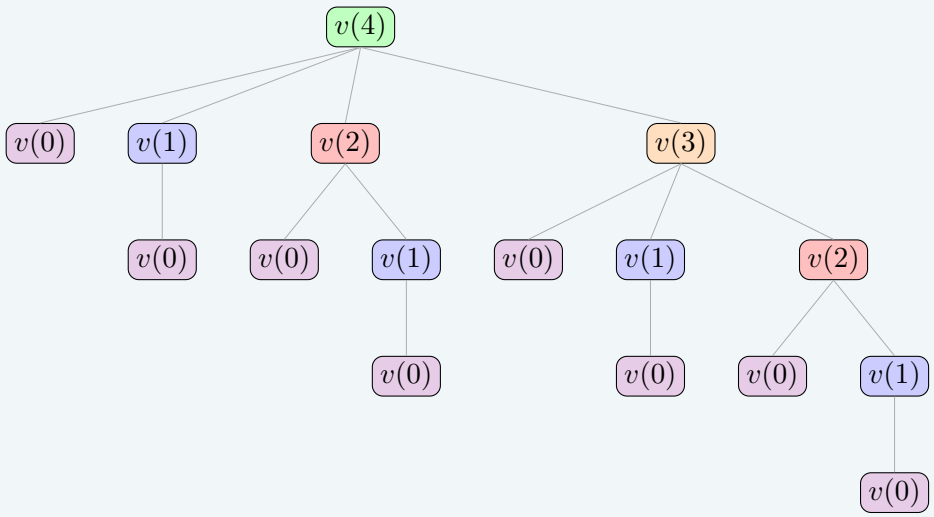
$\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

– Un **primo metodo** per calcolare il valore ottimo $v(m)$ dell'istanza dell'ULS è l'**approccio top-down**. Il metodo parte dalla chiamata con $t = m$ e valuta ricorsivamente la formula (10.3): ogni chiamata per i periodi $\{1, \dots, t\}$ genera altre chiamate per i periodi $\{1, \dots, j-1\}$, per ogni $j \in \{1, 2, \dots, t\}$, finché non si raggiunge il caso base $v(0)$.

– 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 effettuate da quella chiamata. Le foglie rappresentano i casi base.

Esempio 10.5: Albero di ricorsione

Consideriamo i primi quattro periodi dell'istanza dell'ULS dell'Esempio 10.1 (con cinque periodi l'albero ha 32 nodi). Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(4)$:



I figli di un nodo $v(t)$ sono ordinati secondo $j \in \{1, \dots, t\}$: il figlio $v(j-1)$ corrisponde a un ultimo lotto di produzione nel periodo j. La funzione v viene chiamata 16 volte, e gli stessi sottoproblemi vengono risolti più volte: $v(2)$ viene chiamata 2 volte, $v(1)$ viene chiamata 4 volte e $v(0)$ viene chiamata 8 volte. Tutte le altre chiamate vengono effettuate una sola volta.

– Il valore $v(t)$, per un periodo generico t, si può calcolare una volta noti i valori $v(j)$ per tutti i periodi minori $j < t$.

Esempio 10.6: Calcolo dei valori della funzione v

Consideriamo l'istanza dell'ULS dell'Esempio 10.1, con la matrice c dei costi dei lotti di produzione:

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

Il valore della funzione v si può calcolare partendo dal caso base $v(0) = 0$ e calcolando poi i valori di $v(1)$, $v(2)$, $v(3)$, $v(4)$ e $v(5)$ nell'ordine, come segue:

* Poiché conosciamo il valore di $v(0)$, possiamo calcolare $v(1)$ come segue:

$$v(1) = v(0) + c_{1,1} = 0 + 50 = 50$$

* Calcolati $v(1)$ e $v(0)$, possiamo calcolare $v(2)$ come segue:

$$\begin{aligned} v(2) &= \min\{v(0) + c_{1,2}, v(1) + c_{2,2}\} \\ &= \min\{0 + 60, 50 + 50\} = 60 \end{aligned}$$

* Calcolati $v(2)$, $v(1)$ e $v(0)$, possiamo calcolare $v(3)$ come segue:

$$\begin{aligned} v(3) &= \min\{v(0) + c_{1,3}, v(1) + c_{2,3}, v(2) + c_{3,3}\} \\ &= \min\{0 + 120, 50 + 80, 60 + 50\} = 110 \end{aligned}$$

* Calcolati $v(3)$, $v(2)$, $v(1)$ e $v(0)$, possiamo calcolare $v(4)$ come segue:

$$\begin{aligned} v(4) &= \min\{v(0) + c_{1,4}, v(1) + c_{2,4}, v(2) + c_{3,4}, v(3) + c_{4,4}\} \\ &= \min\{0 + 240, 50 + 160, 60 + 90, 110 + 50\} = 150 \end{aligned}$$

* Calcolati $v(4)$, $v(3)$, $v(2)$, $v(1)$ e $v(0)$, possiamo calcolare $v(5)$ come segue:

$$\begin{aligned} v(5) &= \min\{v(0) + c_{1,5}, v(1) + c_{2,5}, v(2) + c_{3,5}, v(3) + c_{4,5}, v(4) + c_{5,5}\} \\ &= \min\{0 + 280, 50 + 190, 60 + 110, 110 + 60, 150 + 50\} = 170 \end{aligned}$$

– Possiamo evitare di risolvere ripetutamente gli stessi sottoproblemi memorizzando il valore di ciascun sottoproblema appena viene calcolato. I valori sono memorizzati in una **tabella di lookup**, il vettore $v[0..m]$, la cui cella $v[t]$ contiene $v(t)$.

– Poiché la soluzione di un sottoproblema con valore $v(t)$ dipende solo dai sottoproblemi con valori $v(j-1)$, con $j-1 < t$, possiamo risolvere i sottoproblemi per valori crescenti di t, partendo dal caso base $v(0) = 0$ fino ad arrivare a $v(m)$: questa osservazione porta in modo naturale a un **metodo molto più efficiente** per calcolare il valore ottimo $v(m)$ dell'istanza dell'ULS.

– Un **secondo metodo** per calcolare il valore ottimo $v(m)$ dell'istanza dell'ULS è l'**approccio bottom-up**. Il metodo parte dal caso base $v(0) = 0$ e valuta la formula (10.3) per periodi crescenti $t \in \{1, 2, \ldots, m\}$: ogni valore $v(t)$ si calcola usando i valori già calcolati $v(0), v(1), \ldots, v(t-1)$, fino a ottenere il valore $v(m)$ dell'istanza dell'ULS.

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

Algoritmo

Algorithm: DP_WW

Input: Un'istanza dell'ULS data dal numero m di periodi e dai vettori $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$

Output: Il valore ottimo

```
1  Function DP_WW(m, d, sc, vc, hc)
2      c := CostMatrix(m, d, sc, vc, hc)
3      v := new array[0..m]
4      v[0] ← 0
5      for t := 1 to m do
6          b := +∞
7          for j := 1 to t do
8              if v[j − 1] + c[j][t] < b then
9                  b ← v[j − 1] + c[j][t]
10             end
11         end
12         v[t] ← b
13     end
14     return v[m]
```

* La matrice c contiene i costi dei lotti di produzione, calcolati dall'algoritmo CostMatrix.
* Il vettore $v[0..m]$ è la **tabella di lookup**: memorizza i valori ottimi di tutti i sottoproblemi, con il caso base $v[0] = 0$.
* Il ciclo esterno, su $t \in \{1, 2, \ldots, m\}$, calcola $v[t]$ in ordine crescente dell'ultimo periodo.
* Il ciclo interno, su $j \in \{1, 2, \ldots, t\}$, valuta tutti i possibili periodi dell'ultimo lotto di produzione e conserva in b il valore minimo $v[j-1] + c[j][t]$; la variabile b parte da $+\infty$, quindi il primo confronto la aggiorna sempre.
* Al termine del ciclo interno, $v[t] \leftarrow b$ memorizza il valore ottimo $v[t]$.
* Infine, viene restituito il valore ottimo $v[m]$ dell'istanza dell'ULS.

Esempio 10.7: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'ULS dell'Esempio 10.1, con la matrice c dei costi dei lotti di produzione:

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di t e j, il valore corrente di b e la cella $v[t]$. La cella di b è evidenziata in rosso quando viene aggiornata; la cella di $v[t]$ è evidenziata in blu quando viene assegnata.

t	j	$v[t]$	b	Azione
–	–	0	–	$v[0] \leftarrow 0$
1	–	–	$+\infty$	$b \leftarrow +\infty$
	1	–	50	$v[0]+c[1][1] = 0+50 = 50 < +\infty$: $b \leftarrow 50$
	–	50	50	$v[1] \leftarrow b = 50$
2	–	–	$+\infty$	$b \leftarrow +\infty$
	1	–	60	$v[0]+c[1][2] = 0+60 = 60 < +\infty$: $b \leftarrow 60$
	2	–	60	$v[1]+c[2][2] = 50+50 = 100 > 60$: nessun aggiornamento
	–	60	60	$v[2] \leftarrow b = 60$
3	–	–	$+\infty$	$b \leftarrow +\infty$
	1	–	120	$v[0]+c[1][3] = 0+120 = 120 < +\infty$: $b \leftarrow 120$
	2	–	120	$v[1]+c[2][3] = 50+80 = 130 > 120$: nessun aggiornamento
	3	–	110	$v[2]+c[3][3] = 60+50 = 110 < 120$: $b \leftarrow 110$
	–	110	110	$v[3] \leftarrow b = 110$
4	–	–	$+\infty$	$b \leftarrow +\infty$
	1	–	240	$v[0]+c[1][4] = 0+240 = 240 < +\infty$: $b \leftarrow 240$
	2	–	210	$v[1]+c[2][4] = 50+160 = 210 < 240$: $b \leftarrow 210$
	3	–	150	$v[2]+c[3][4] = 60+90 = 150 < 210$: $b \leftarrow 150$
	4	–	150	$v[3]+c[4][4] = 110+50 = 160 > 150$: nessun aggiornamento
	–	150	150	$v[4] \leftarrow b = 150$
5	–	–	$+\infty$	$b \leftarrow +\infty$
	1	–	280	$v[0]+c[1][5] = 0+280 = 280 < +\infty$: $b \leftarrow 280$
	2	–	240	$v[1]+c[2][5] = 50+190 = 240 < 280$: $b \leftarrow 240$
	3	–	170	$v[2]+c[3][5] = 60+110 = 170 < 240$: $b \leftarrow 170$
	4	–	170	$v[3]+c[4][5] = 110+60 = 170 = b$: nessun aggiornamento
	5	–	170	$v[4]+c[5][5] = 150+50 = 200 > 170$: nessun aggiornamento
	–	170	170	$v[5] \leftarrow b = 170$

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

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$v[t]$	0	50	60	110	150	170

  - Dopo l'esecuzione dell'algoritmo di PD, il vettore v contiene il valore ottimo $v[t]$ per ogni insieme di periodi $\{1, \dots, t\}$.

  - *Osservazione (cammino minimo).* La formula (10.3) calcola un cammino minimo in una rete con nodi $0, 1, \dots, m$ e un arco dal nodo $j-1$ al nodo t, di costo $c_{j,t}$, per ogni $j \leq t$. Il valore $v(t)$ è la lunghezza di un cammino minimo dal nodo 0 al nodo t. È la formulazione a cammino minimo del capitolo «Formulazioni dei problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*, senza le capacità: un arco è un lotto di produzione.

- Passo 2: calcolo di una soluzione ottima

  - Per calcolare una soluzione ottima è necessario memorizzare alcune informazioni aggiuntive.

  - Per ogni periodo t, oltre al valore ottimo $v[t]$, memorizziamo $r[t]$, il periodo dell'ultimo lotto di produzione in una soluzione ottima per i periodi $\{1, \dots, t\}$.

  - Partendo da $t = m$, il valore $r[t]$ dà il periodo dell'ultimo lotto di produzione, che produce $D_{r[t],t}$ unità; i periodi rimanenti sono $\{1, \dots, r[t]-1\}$, e la stessa regola si può applicare di nuovo, e così via, finché non resta alcun periodo.

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

Algoritmo

Algorithm: DP_WW_InfoSol

Input: Un'istanza dell'ULS data dal numero m di periodi e dai vettori $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$

Output: Il valore ottimo e il vettore delle informazioni $r[1..m]$

```
1  Function DP_WW_InfoSol(m, d, sc, vc, hc)
2      c := CostMatrix(m, d, sc, vc, hc)
3      v := new array[0..m]
4      r := new array[1..m]
5      v[0] ← 0
6      for t := 1 to m do
7          b := +∞
8          a := 0
9          for j := 1 to t do
10             if v[j − 1] + c[j][t] < b then
11                 b ← v[j − 1] + c[j][t]
12                 a ← j
13             end
14         end
15         v[t] ← b
16         r[t] ← a
17     end
18     return v[m], r
```

* L'algoritmo di PD usa la variabile temporanea aggiuntiva a per tenere traccia della decisione presa quando b viene aggiornato. Se viene scelto l'ultimo lotto di produzione nel periodo j, $a \leftarrow j$ memorizza questa decisione. Al termine del ciclo interno, $r[t] \leftarrow a$ memorizza il periodo dell'ultimo lotto di produzione scelto, insieme al valore ottimo $v[t] \leftarrow b$.
* A causa del confronto stretto, in caso di pareggio si memorizza il periodo j minimo.

Esempio 10.8: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza dell'ULS dell'Esempio 10.1, con la matrice c dei costi dei lotti di produzione:

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di t e j, la cella $r[t]$, il valore corrente di b e il valore corrente di a. La cella di b è evidenziata in

rosso quando viene aggiornata; la cella di a e la cella di $r[t]$ sono evidenziate in blu quando vengono aggiornate.

t	j	$r[t]$	b	a	Azione
–	–	–	–	–	Stato iniziale
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
1	1	–	50	1	$v[0]+c[1][1] = 0+50 = 50 < +\infty$: $b \leftarrow 50$, $a \leftarrow 1$
	–	1	50	1	$v[1] \leftarrow b = 50$, $r[1] \leftarrow a = 1$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
2	1	–	60	1	$v[0]+c[1][2] = 0+60 = 60 < +\infty$: $b \leftarrow 60$, $a \leftarrow 1$
	2	–	60	1	$v[1]+c[2][2] = 50+50 = 100 > 60$: nessun aggiornamento
	–	1	60	1	$v[2] \leftarrow b = 60$, $r[2] \leftarrow a = 1$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
3	1	–	120	1	$v[0]+c[1][3] = 0+120 = 120 < +\infty$: $b \leftarrow 120$, $a \leftarrow 1$
	2	–	120	1	$v[1]+c[2][3] = 50+80 = 130 > 120$: nessun aggiornamento
	3	–	110	3	$v[2]+c[3][3] = 60+50 = 110 < 120$: $b \leftarrow 110$, $a \leftarrow 3$
	–	3	110	3	$v[3] \leftarrow b = 110$, $r[3] \leftarrow a = 3$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
4	1	–	240	1	$v[0]+c[1][4] = 0+240 = 240 < +\infty$: $b \leftarrow 240$, $a \leftarrow 1$
	2	–	210	2	$v[1]+c[2][4] = 50+160 = 210 < 240$: $b \leftarrow 210$, $a \leftarrow 2$
	3	–	150	3	$v[2]+c[3][4] = 60+90 = 150 < 210$: $b \leftarrow 150$, $a \leftarrow 3$
	4	–	150	3	$v[3]+c[4][4] = 110+50 = 160 > 150$: nessun aggiornamento
	–	3	150	3	$v[4] \leftarrow b = 150$, $r[4] \leftarrow a = 3$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
5	1	–	280	1	$v[0]+c[1][5] = 0+280 = 280 < +\infty$: $b \leftarrow 280$, $a \leftarrow 1$
	2	–	240	2	$v[1]+c[2][5] = 50+190 = 240 < 280$: $b \leftarrow 240$, $a \leftarrow 2$
	3	–	170	3	$v[2]+c[3][5] = 60+110 = 170 < 240$: $b \leftarrow 170$, $a \leftarrow 3$
	4	–	170	3	$v[3]+c[4][5] = 110+60 = 170 = b$: nessun aggiornamento
	5	–	170	3	$v[4]+c[5][5] = 150+50 = 200 > 170$: nessun aggiornamento
	–	3	170	3	$v[5] \leftarrow b = 170$, $r[5] \leftarrow a = 3$

Questo è il vettore r dopo l'esecuzione dell'algoritmo di PD:

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$r[t]$	1	1	3	3	3

– Calcolato il vettore r, si può calcolare una soluzione ottima dell'istanza dell'ULS applicando ripetutamente la regola seguente, partendo da $t = m$ e da $x_j = 0$ per ogni $j \in \{1, 2, \ldots, m\}$:

$$x_{r[t]} \leftarrow D_{r[t],t}, \qquad t \leftarrow r[t] - 1,$$

finché $t = 0$. Si ottiene così il vettore $(x_1, x_2, \ldots, x_m)$, in cui ogni componente x_j è il numero di unità prodotte nel periodo j in una soluzione ottima.

– Lo pseudocodice seguente implementa il calcolo di una soluzione ottima, dato il vettore r.

Algoritmo

Algorithm: DP_WW_SolComputation

Input: Il numero m di periodi e le domande $d[1..m]$ dell'istanza dell'ULS, e il vettore $r[1..m]$

Output: Una soluzione ottima dell'istanza dell'ULS

```
1  Function DP_WW_SolComputation(m, d, r)
2      x := new array[1..m]
3      for j := 1 to m do
4          x[j] ← 0
5      end
6      t := m
7      while t > 0 do
8          x[r[t]] ← Σ_{k=r[t]}^{t} d[k]
9          t ← r[t] − 1
10     end
11     return x
```

Esempio 10.9: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 10.1, abbiamo il seguente vettore r:

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$r[t]$	1	1	3	3	3

Possiamo calcolare una soluzione ottima per $m=5$ periodi come segue:

Iterazione	t	$r[t]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$t \leftarrow r[t]-1$
–	–	–	0	0	0	0	0	$t=5$
1	5	3	0	0	80	0	0	$t=3-1=2$
2	2	1	30	0	80	0	0	$t=1-1=0$

* $t=5$, $r[5]=3$: l'ultimo lotto di produzione dei periodi $1,\ldots,5$ è nel periodo 3 e copre le domande dei periodi $3,\ldots,5$, quindi $x[3] \leftarrow d_3+d_4+d_5 = 30+40+10 = 80$

* $t=2$, $r[2]=1$: l'ultimo lotto di produzione dei periodi $1,\ldots,2$ è nel periodo 1 e copre le domande dei periodi $1,\ldots,2$, quindi $x[1] \leftarrow d_1+d_2 = 20+10 = 30$

Una soluzione ottima è il vettore:

$$(30,0,80,0,0)$$

Il valore ottimo corrispondente è $v[5]=170$, lo stesso valore ottenuto dall'enumerazione dell'Esempio 10.3 e da Gurobi sul modello ILP_{ULS}.

– *Osservazione (pareggi).* Nel calcolo di $v[5]$, il lotto nel periodo 4 dà lo stesso valore del lotto nel periodo 3, ma il confronto stretto conserva $j = 3$. Scegliendo invece $j = 4$ si ottiene l'altra soluzione ottima $(30, 0, 30, 50, 0)$ dell'Esempio 10.3. L'algoritmo di PD restituisce una soluzione ottima, non tutte.

10.2.3 Numero di passi

- Indichiamo con $S_c(m)$ il numero di passi eseguiti da CostMatrix nel caso peggiore, dove m è il numero di periodi. Il numero di esecuzioni delle sue righe è:
  - La riga 2 viene eseguita 1 volta.
  - Le righe 3, 4, 5 e 6 vengono eseguite m volte ciascuna.
  - Le righe 7, 8, 9 e 10 vengono eseguite $\sum_{j=1}^{m}(m-j) = \frac{m(m-1)}{2}$ volte ciascuna.
  - La riga 13 viene eseguita 1 volta.

  Quindi
  $$S_c(m) = 2 + 4m + 4 \cdot \frac{m(m-1)}{2} = 2m^2 + 2m + 2.$$
- Indichiamo con $S(m)$ il numero di passi eseguiti da DP_WW nel caso peggiore. La riga 2 esegue CostMatrix, cioè $S_c(m)$ passi. Nel caso peggiore, la condizione alla riga 8 è vera a ogni iterazione del ciclo interno, e quindi la riga 9 viene eseguita tante volte quanto la riga 8. Il numero di esecuzioni delle altre righe è:
  - La riga 3 viene eseguita 1 volta.
  - La riga 4 viene eseguita 1 volta.
  - La riga 5 viene eseguita m volte.
  - La riga 6 viene eseguita m volte.
  - La riga 7 viene eseguita $\sum_{t=1}^{m} t$ volte.
  - La riga 8 viene eseguita $\sum_{t=1}^{m} t$ volte.
  - La riga 9 viene eseguita $\sum_{t=1}^{m} t$ volte.
  - La riga 12 viene eseguita m volte.
  - La riga 14 viene eseguita 1 volta.

  Quindi il numero totale di passi nel caso peggiore è
  $$S(m) = S_c(m)+3+3m+3\sum_{t=1}^{m} t = 2m^2+2m+2+3+3m+3\cdot\frac{m(m+1)}{2} = \frac{7m^2+13m+10}{2}.$$

  Di conseguenza,
  $$S(m) = \Theta(m^2).$$

Infatti,

$$\lim_{m\to\infty} \frac{\frac{7m^2+13m+10}{2}}{m^2} = \lim_{m\to\infty} \frac{1}{2}\left(7+\frac{13}{m}+\frac{10}{m^2}\right) = \frac{7}{2} \in \mathbb{R}_{>0}.$$

L'algoritmo di PD esegue

$$S(m) = \Theta(m^2)$$

passi nel caso peggiore.

- DP_WW_InfoSol aggiunge solo un numero costante di operazioni negli stessi cicli, e quindi il suo numero di passi ha lo stesso ordine di crescita.

10.3 Errori tipici

Attenzione

- **Applicare la proprietà zero-inventory ordering con le capacità.** La dimostrazione sposta la produzione fra due periodi senza limiti: con le capacità un periodo può produrre mentre ci sono unità a scorta.
- **Calcolare da zero il costo di un lotto.** Calcolare ogni $c_{j,t}$ dalla sua definizione costa $O(m^3)$ passi dentro l'algoritmo di PD; il calcolo incrementale di CostMatrix mantiene $\Theta(m^2)$.
- **Dimenticare la scorta finale vuota.** La sottoistanza con i periodi $\{1,\dots,t\}$ termina con la scorta vuota: per questo il lotto nel periodo j produce esattamente $D_{j,t}$ unità.
- **Prendere un piano ottimo per l'unico.** L'Esempio 10.1 ha due piani ottimi: l'algoritmo di PD restituisce quello con il periodo j minimo, perché aggiorna solo con un valore strettamente minore.

10.4 In sintesi

In sintesi

- Se tutti i costi sono non negativi, il problema senza capacità ha un piano ottimo in cui un periodo produce solo quando la scorta è nulla (Wagner e Whitin), quindi $x_j = D_{j,t}$ per ogni lotto di produzione.
- Il costo $c_{j,t}$ di ogni lotto si calcola in $\Theta(m^2)$ passi, e la ricorrenza $v(t) = \min_j\{v(j-1) + c_{j,t}\}$ dà il valore ottimo in $\Theta(m^2)$ passi; è un cammino minimo.
- La proprietà zero-inventory ordering si basa su una produzione senza limiti: con le capacità non vale.

Quadro riassuntivo dell'esempio.

Istanza	Lower bound	Upper bound	Origine
Esempio 10.1 (senza capacità, $m = 5$)	170	170	algoritmo di PD di Wagner e Whitin

Capitolo 11

PD per il problema del lot sizing capacitato

Prerequisiti

- Da questo modulo, il capitolo «PD per il problema del lot sizing senza capacità»: le domande totali $D_{j,t}$, la codifica di un piano di produzione e la sua rete, la proprietà zero-inventory ordering, e l'istanza del suo Esempio 10.1.
- Dal modulo *Problemi di ottimizzazione combinatoria*, i capitoli «Problemi di lot sizing» e «Formulazioni dei problemi di lot sizing»: il problema capacitato, la sua istanza di riferimento, e il suo modello PLI con le variabili x_t, s_t, y_t.
- Da questo modulo, il capitolo «Principi della programmazione dinamica»: ricorrenza, casi base, tabella di lookup e ricostruzione di una soluzione.
- Dal modulo *Algoritmi*: lo pseudocodice (Capitolo 2), e la dimensione dell'input (Capitolo 11).

Dove eravamo

- **Come problema e come modello**: il problema di lot sizing capacitato e le sue formulazioni PLI (capitoli «Problemi di lot sizing» e «Formulazioni dei problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*).
- **Come tabella, senza capacità**: l'algoritmo di PD di Wagner e Whitin del Capitolo 10 «PD per il problema del lot sizing senza capacità», sui lotti di produzione, in $\Theta(m^2)$ passi.
- **Ora, con le capacità**: un algoritmo di PD sulle coppie (periodo, scorta), che è pseudo-polinomiale.

Obiettivi

Al termine di questo capitolo saremo in grado di:

- spiegare su un esempio perché la proprietà zero-inventory ordering non vale con le capacità;
- scrivere la ricorrenza $v(t, k)$ sulle coppie (periodo, scorta finale), con i suoi casi base;

- eseguire a mano l'algoritmo di PD, riempire le matrici v e r, e ricostruire un piano ottimo;
- contare i passi dell'algoritmo di PD, e spiegare perché è pseudo-polinomiale;
- enunciare la proprietà più debole che vale quando la capacità è la stessa in tutti i periodi.

Il problema di questo capitolo è il problema di lot sizing capacitato (CLS): la sua definizione e le sue ipotesi sono nella sua pagina.

11.1 Notazione, modello e soluzioni ammissibili

- Come nel Capitolo 10 «PD per il problema del lot sizing senza capacità», per ogni $j, t \in T$ indichiamo con
$$D_{j,t} = \sum_{k=j}^{t} d_k$$
la domanda totale dei periodi $j, j+1, \ldots, t$, con $D_{j,t} = 0$ se $j > t$.

- Il modello PLI è il modello ILP_{CLSP} del capitolo «Formulazioni dei problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*, con il valore di big-M $\min\{cap_t, D_{t,m}\}$ del periodo t: è il modello ILP_{ULS} con i vincoli aggiuntivi $x_t \leq cap_t$, per ogni $t \in T$.

- Una soluzione ammissibile si codifica con lo stesso vettore $(x_1, x_2, \ldots, x_m)$ dell'ULS, con le scorte $s_t = \sum_{j=1}^{t} x_j - D_{1,t}$, e con il requisito aggiuntivo $x_t \leq cap_t$ per ogni $t \in T$.

- La **proprietà zero-inventory ordering non vale** per il CLS: un periodo può dover produrre mentre ci sono ancora unità in scorta, perché la capacità non permette di produrre tutto nel periodo in cui sarebbe più economico.

- La proprietà, e l'algoritmo di PD sui lotti di produzione che si basa su di essa, sono quelli del Capitolo 10 «PD per il problema del lot sizing senza capacità».

Esempio 11.1: lot sizing capacitato

Consideriamo l'istanza senza capacità dell'Esempio 10.1 del Capitolo 10 «PD per il problema del lot sizing senza capacità», misurata in unità di 10 prodotti (quindi il costo di mantenimento è 10 per unità), con una capacità costante di 4 unità per periodo:

t	1	2	3	4	5
d_t	2	1	3	4	1
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	10	10	10	10	10
cap_t	4	4	4	4	4

Chiamiamo questa istanza del CLS Esempio 11.1: è l'istanza di riferimento del capitolo «Problemi di lot sizing» del modulo *Problemi di ottimizzazione combinatoria*. La soluzione ottima dell'istanza senza capacità, in unità di 10, è il vettore $(3, 0, 8, 0, 0)$, di costo 170: non è ammissibile per l'Esempio 11.1, poiché il periodo 3 produce più di 4 unità.

11.2 Un algoritmo di PD per il CLS

11.2.1 Passo 1 e Passo 2

- Passo 1: calcolo del valore ottimo
- Per ogni periodo $t \in \{1, \ldots, m\}$, la **funzione di costo di produzione** del periodo t è

$$f_t(x) = \begin{cases} 0 & \text{se } x = 0, \\ sc_t + vc_t\, x & \text{se } x \in \{1, 2, \ldots, cap_t\}. \end{cases}$$

La scorta alla fine del periodo t non supera mai la domanda $D_{t+1,m}$ dei periodi successivi, poiché la scorta deve essere vuota alla fine del periodo m.

Per ogni $t \in \{1, 2, \ldots, m\}$ e $k \in \{0, 1, \ldots, D_{t+1,m}\}$, sia $v(t, k)$ il valore ottimo della sottoistanza del CLS data da:

1. un insieme $\{1, 2, \ldots, t\}$ di t periodi di tempo,
2. per ogni periodo $j \in \{1, 2, \ldots, t\}$, una domanda d_j,
3. per ogni periodo $j \in \{1, 2, \ldots, t\}$, un costo di setup sc_j, un costo variabile di produzione vc_j e un costo di mantenimento a scorta hc_j,
4. per ogni periodo $j \in \{1, 2, \ldots, t\}$, una capacità cap_j,
5. una scorta uguale a k alla fine del periodo t.

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

$$v(t,k) = \begin{cases} \min\left\{v(t-1, k+d_t-x) + f_t(x) : x \in X_t(k)\right\} & \text{se } 1 \le t \le m, \\ \quad + hc_t\, k & \\ 0 & \text{se } t = 0 \text{ e } k = 0, \\ +\infty & \text{se } t = 0 \text{ e } k > 0, \end{cases} \tag{11.1}$$

dove $X_t(k) = \{0, 1, \ldots, \min\{cap_t,\, k + d_t\}\}$ è l'insieme delle possibili produzioni del periodo t.

- La coppia (t, k) è lo **stato**: l'ultimo periodo della sottoistanza e la scorta alla sua fine. Per $t = m$ e $k = 0$ otteniamo l'istanza del CLS stessa, quindi il suo valore ottimo è $v(m, 0)$.
- Il processo di decomposizione ricorsiva della formula (11.1) genera al più $cap_t + 1$ **sottoproblemi indipendenti**, ciascuno corrispondente a una possibile scelta della produzione x dell'ultimo periodo t.
- Se si producono x unità nel periodo t e alla sua fine restano k unità, la scorta alla fine del periodo $t-1$ è $k+d_t-x$: il sottoproblema residuo è la sottoistanza del CLS con i periodi $\{1, \ldots, t-1\}$ e scorta finale $k + d_t - x$.
- La produzione x è al più cap_t, e al più $k + d_t$ perché la scorta alla fine del periodo $t-1$ non può essere negativa. Inoltre $k + d_t - x \le D_{t+1,m} + d_t = D_{t,m}$, quindi il sottoproblema residuo è uno stato.
- Il caso base è $v(0, 0) = 0$: nessun periodo, nessuna scorta, nessun costo.
- Per $k > 0$ poniamo $v(0, k) = +\infty$: la scorta iniziale è vuota, quindi nessuna soluzione ammissibile ha una scorta positiva prima del periodo 1. Il valore $+\infty$ esclude questo caso, e si propaga a ogni stato che non si può raggiungere entro le capacità, con la convenzione $+\infty + a = +\infty$.
- Il valore ottimo $v(t, k)$ si ottiene provando tutte le possibili produzioni x del periodo t e scegliendo quella che minimizza la somma $v(t-1, k+d_t-x) + f_t(x)$ del valore ottimo del sottoproblema residuo e del costo di produzione del periodo t; il costo di mantenimento $hc_t\, k$ della scorta finale non dipende da x.
- Quindi, una soluzione ottima della sottoistanza dello stato (t, k) si costruisce combinando la produzione scelta del periodo t con una soluzione ottima del sottoproblema residuo dello stato $(t-1, k+d_t-x)$.

  $\rightsquigarrow$ **Sottostruttura ottima verificata!**
- Sequenze diverse di decisioni di produzione possono portare allo stesso stato. Per esempio, nell'Esempio 11.1, partendo dallo stato $(5, 0)$, le decisioni $x_5 = 0$ e $x_4 = 4$ portano allo stato $(3, 1)$, e le decisioni $x_5 = 1$ e $x_4 = 3$ portano allo stesso stato $(3, 1)$.

– Quindi lo stesso valore $v(t,k)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (11.1).

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

– Un **primo metodo** per calcolare il valore ottimo $v(m,0)$ dell'istanza del CLS è l'**approccio top-down**. Il metodo parte dalla chiamata con $(t,k) = (m,0)$ e valuta ricorsivamente la formula (11.1): ogni chiamata per uno stato (t,k) genera ulteriori chiamate per gli stati $(t-1, k+d_t-x)$, per ogni $x \in \{0,\ldots,\min\{cap_t, k+d_t\}\}$, finché non si raggiunge uno dei casi base $v(0,k)$.

– Il valore $v(t,k)$ si può calcolare una volta noti i valori $v(t-1,k')$ per ogni $k' \in \{0,\ldots,D_{t,m}\}$.

– Possiamo evitare di risolvere ripetutamente gli stessi sottoproblemi memorizzando il valore di ciascun sottoproblema non appena viene calcolato, in una **tabella di lookup**: la matrice $v[0..m][0..D_{1,m}]$, la cui cella $v[t][k]$ contiene $v(t,k)$ per $k \leq D_{t+1,m}$ (le altre celle non si usano).

– Un **secondo metodo** per calcolare il valore ottimo $v(m,0)$ dell'istanza del CLS è l'**approccio bottom-up**. Il metodo parte dai casi base $v(0,0) = 0$ e $v(0,k) = +\infty$ per ogni $k \in \{1,\ldots,D_{1,m}\}$.

– Valuta la formula (11.1) per valori crescenti di $t \in \{1,\ldots,m\}$ e, per ogni t fissato, per $k \in \{0,1,\ldots,D_{t+1,m}\}$: ogni valore $v(t,k)$ usa valori già calcolati della riga $t-1$, finché non si ottiene $v(m,0)$.

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

Algoritmo

Algorithm: DP_CLS

Input: Un'istanza del CLS data dal numero m di periodi e dai vettori $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$, $cap[1..m]$

Output: Il valore ottimo

```
1  Function DP_CLS(m, d, sc, vc, hc, cap)
2      R := new array[0..m]
3      R[m] ← 0
4      for t := m − 1 downto 0 do
5          R[t] ← R[t + 1] + d[t + 1]
6      end
7      v := new array[0..m][0..R[0]]
8      v[0][0] ← 0
9      for k := 1 to R[0] do
10         v[0][k] ← +∞
11     end
12     for t := 1 to m do
13         for k := 0 to R[t] do
14             b := +∞
15             for x := 0 to min{cap[t], k + d[t]} do
16                 if v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k < b then
17                     b ← v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k
18                 end
19             end
20             v[t][k] ← b
21         end
22     end
23     return v[m][0]
```

* Il vettore $R[0..m]$ contiene le domande residue $R[t] = D_{t+1,m}$, calcolate all'indietro a partire da $R[m] = 0$: gli stati del periodo t sono le coppie (t, k) con $k \leq R[t]$.
* La matrice v è la **tabella di lookup**. L'inizializzazione pone $v[0][0] = 0$ e $v[0][k] = +\infty$ per ogni $k \in \{1, \ldots, R[0]\}$, poiché la scorta iniziale è vuota.
* Il ciclo esterno, su $t \in \{1, 2, \ldots, m\}$, considera i periodi in ordine crescente, e il secondo ciclo, su $k \in \{0, \ldots, R[t]\}$, considera tutte le scorte finali del periodo t.
* Il ciclo interno, sulla produzione x del periodo t, mantiene in b il valore minimo $v[t-1][k+d[t]-x]+f_t(x)+hc[t]\cdot k$; il valore $f_t(x)$ è 0 per $x = 0$ e $sc[t]+vc[t]\cdot x$ altrimenti. Se tutti i candidati valgono $+\infty$, il confronto non è mai vero e $v[t][k] = +\infty$: lo stato (t, k) non si può raggiungere entro le capacità.
* Infine, viene restituito il valore ottimo $v[m][0]$ dell'istanza del CLS.

Esempio 11.2: esecuzione dell'algoritmo di PD

Consideriamo l'istanza del CLS dell'Esempio 11.1:

t	1	2	3	4	5
d_t	2	1	3	4	1
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	10	10	10	10	10
cap_t	4	4	4	4	4

Le domande residue sono $(R[0], \ldots, R[5]) = (11, 9, 8, 5, 1, 0)$.

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di t e k, la cella $v[t][k]$ e l'operazione eseguita. I candidati del minimo sono elencati per $x = 0, 1, \ldots, \min\{cap_t, k + d_t\}$.

t	k	$v[t][k]$	Azione
0	0	0	$v[0][0] \leftarrow 0$
0	1	$+\infty$	$v[0][1] \leftarrow +\infty$
0	2	$+\infty$	$v[0][2] \leftarrow +\infty$
0	3	$+\infty$	$v[0][3] \leftarrow +\infty$
0	4	$+\infty$	$v[0][4] \leftarrow +\infty$
0	5	$+\infty$	$v[0][5] \leftarrow +\infty$
0	6	$+\infty$	$v[0][6] \leftarrow +\infty$
0	7	$+\infty$	$v[0][7] \leftarrow +\infty$
0	8	$+\infty$	$v[0][8] \leftarrow +\infty$
0	9	$+\infty$	$v[0][9] \leftarrow +\infty$
0	10	$+\infty$	$v[0][10] \leftarrow +\infty$
0	11	$+\infty$	$v[0][11] \leftarrow +\infty$
1	0	50	$\min\{v[0][2]+f_1(0), v[0][1]+f_1(1), v[0][0]+f_1(2)\}+hc_1 \cdot 0 = \min\{+\infty, +\infty, 0+50\}+10 \cdot 0 = 50$
1	1	60	$\min\{v[0][3]+f_1(0), v[0][2]+f_1(1), v[0][1]+f_1(2), v[0][0]+f_1(3)\}+hc_1 \cdot 1 = \min\{+\infty, +\infty, +\infty, 0+50\}+10 \cdot 1 = 60$
1	2	70	$\min\{v[0][4]+f_1(0), v[0][3]+f_1(1), v[0][2]+f_1(2), v[0][1]+f_1(3), v[0][0]+f_1(4)\}+hc_1 \cdot 2 = \min\{+\infty, +\infty, +\infty, +\infty, 0+50\}+10 \cdot 2 = 70$
1	3	$+\infty$	i candidati usano le celle $v[0][1], \ldots, v[0][5]$, che valgono tutte $+\infty$: $v[1][3] \leftarrow +\infty$
1	4	$+\infty$	i candidati usano le celle $v[0][2], \ldots, v[0][6]$, che valgono tutte $+\infty$: $v[1][4] \leftarrow +\infty$
1	5	$+\infty$	i candidati usano le celle $v[0][3], \ldots, v[0][7]$, che valgono tutte $+\infty$: $v[1][5] \leftarrow +\infty$
1	6	$+\infty$	i candidati usano le celle $v[0][4], \ldots, v[0][8]$, che valgono tutte $+\infty$: $v[1][6] \leftarrow +\infty$
1	7	$+\infty$	i candidati usano le celle $v[0][5], \ldots, v[0][9]$, che valgono tutte $+\infty$: $v[1][7] \leftarrow +\infty$
1	8	$+\infty$	i candidati usano le celle $v[0][6], \ldots, v[0][10]$, che valgono tutte $+\infty$: $v[1][8] \leftarrow +\infty$
1	9	$+\infty$	i candidati usano le celle $v[0][7], \ldots, v[0][11]$, che valgono tutte $+\infty$: $v[1][9] \leftarrow +\infty$
2	0	60	$\min\{v[1][1]+f_2(0), v[1][0]+f_2(1)\}+hc_2 \cdot 0 = \min\{60+0, 50+50\}+10 \cdot 0 = 60$
2	1	80	$\min\{v[1][2]+f_2(0), v[1][1]+f_2(1), v[1][0]+f_2(2)\}+hc_2 \cdot 1 = \min\{70+0, 60+50, 50+50\}+10 \cdot 1 = 80$
2	2	120	$\min\{v[1][3]+f_2(0), v[1][2]+f_2(1), v[1][1]+f_2(2), v[1][0]+f_2(3)\}+hc_2 \cdot 2 = \min\{+\infty, 70+50, 60+50, 50+50\}+10 \cdot 2 = 120$
2	3	130	$\min\{v[1][4]+f_2(0), v[1][3]+f_2(1), v[1][2]+f_2(2), v[1][1]+f_2(3), v[1][0]+f_2(4)\}+hc_2 \cdot 3 = \min\{+\infty, +\infty, 70+50, 60+50, 50+50\}+10 \cdot 3 = 130$
2	4	150	$\min\{v[1][5]+f_2(0), v[1][4]+f_2(1), v[1][3]+f_2(2), v[1][2]+f_2(3), v[1][1]+f_2(4)\}+hc_2 \cdot 4 = \min\{+\infty, +\infty, +\infty, 70+50, 60+50\}+10 \cdot 4 = 150$
2	5	170	$\min\{v[1][6]+f_2(0), v[1][5]+f_2(1), v[1][4]+f_2(2), v[1][3]+f_2(3), v[1][2]+f_2(4)\}+hc_2 \cdot 5 = \min\{+\infty, +\infty, +\infty, +\infty, 70+50\}+10 \cdot 5 = 170$
2	6	$+\infty$	i candidati usano le celle $v[1][3], \ldots, v[1][7]$, che valgono tutte $+\infty$: $v[2][6] \leftarrow +\infty$
2	7	$+\infty$	i candidati usano le celle $v[1][4], \ldots, v[1][8]$, che valgono tutte $+\infty$: $v[2][7] \leftarrow +\infty$
2	8	$+\infty$	i candidati usano le celle $v[1][5], \ldots, v[1][9]$, che valgono tutte $+\infty$: $v[2][8] \leftarrow +\infty$
3	0	110	$\min\{v[2][3]+f_3(0), v[2][2]+f_3(1), v[2][1]+f_3(2), v[2][0]+f_3(3)\}+hc_3 \cdot 0 = \min\{130+0, 120+50, 80+50, 60+50\}+10 \cdot 0 = 110$

3	1	120	$\min\{v[2][4]+f_3(0), v[2][3]+f_3(1), v[2][2]+f_3(2), v[2][1]+f_3(3), v[2][0]+f_3(4)\}+hc_3 \cdot 1 = \min\{150+0, 130+50, 120+50, 80+50, 60+50\}+10 \cdot 1 = 120$
3	2	150	$\min\{v[2][5]+f_3(0), v[2][4]+f_3(1), v[2][3]+f_3(2), v[2][2]+f_3(3), v[2][1]+f_3(4)\}+hc_3 \cdot 2 = \min\{170+0, 150+50, 130+50, 120+50, 80+50\}+10 \cdot 2 = 150$
3	3	200	$\min\{v[2][6]+f_3(0), v[2][5]+f_3(1), v[2][4]+f_3(2), v[2][3]+f_3(3), v[2][2]+f_3(4)\}+hc_3 \cdot 3 = \min\{+\infty, 170+50, 150+50, 130+50, 120+50\}+10 \cdot 3 = 200$
3	4	220	$\min\{v[2][7]+f_3(0), v[2][6]+f_3(1), v[2][5]+f_3(2), v[2][4]+f_3(3), v[2][3]+f_3(4)\}+hc_3 \cdot 4 = \min\{+\infty, +\infty, 170+50, 150+50, 130+50\}+10 \cdot 4 = 220$
3	5	250	$\min\{v[2][8]+f_3(0), v[2][7]+f_3(1), v[2][6]+f_3(2), v[2][5]+f_3(3), v[2][4]+f_3(4)\}+hc_3 \cdot 5 = \min\{+\infty, +\infty, +\infty, 170+50, 150+50\}+10 \cdot 5 = 250$
4	0	160	$\min\{v[3][4]+f_4(0), v[3][3]+f_4(1), v[3][2]+f_4(2), v[3][1]+f_4(3), v[3][0]+f_4(4)\}+hc_4 \cdot 0 = \min\{220+0, 200+50, 150+50, 120+50, 110+50\}+10 \cdot 0 = 160$
4	1	180	$\min\{v[3][5]+f_4(0), v[3][4]+f_4(1), v[3][3]+f_4(2), v[3][2]+f_4(3), v[3][1]+f_4(4)\}+hc_4 \cdot 1 = \min\{250+0, 220+50, 200+50, 150+50, 120+50\}+10 \cdot 1 = 180$
5	0	180	$\min\{v[4][1]+f_5(0), v[4][0]+f_5(1)\}+hc_5 \cdot 0 = \min\{180+0, 160+50\}+10 \cdot 0 = 180$

Questa è la matrice v dopo l'esecuzione dell'algoritmo di PD (un punto indica una coppia (t, k) con $k > D_{t+1,m}$, che non è uno stato):

$v[t][k]$	0	1	2	3	4	5	6	7	8	9	10	11
$t=0$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$t=1$	50	60	70	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	·	·
$t=2$	60	80	120	130	150	170	$+\infty$	$+\infty$	$+\infty$	·	·	·
$t=3$	110	120	150	200	220	250	·	·	·	·	·	·
$t=4$	160	180	·	·	·	·	·	·	·	·	·	·
$t=5$	180	·	·	·	·	·	·	·	·	·	·	·

Il valore ottimo dell'istanza del CLS è $v[5][0] = 180$.

  - Dopo l'esecuzione dell'algoritmo di PD, la matrice v contiene il valore ottimo $v[t][k]$ di ogni stato (t, k).

- Passo 2: calcolo di una soluzione ottima
  - Per calcolare una soluzione ottima è necessario memorizzare alcune informazioni aggiuntive.
  - Per ogni stato (t, k), oltre al valore ottimo $v[t][k]$, memorizziamo $r[t][k]$, la produzione del periodo t in una soluzione ottima della sottoistanza dello stato (t, k); per gli stati con valore $+\infty$ memorizziamo 0.

– Partendo da $(t, k) = (m, 0)$, il valore $r[t][k]$ dà la produzione del periodo t; la scorta alla fine del periodo $t-1$ è allora $k + d_t - r[t][k]$, e la stessa regola si può applicare di nuovo al periodo $t-1$, e così via, fino a $t = 0$.

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

Algoritmo

Algorithm: DP_CLS_InfoSol

Input: Un'istanza del CLS data dal numero m di periodi e dai vettori $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$, $cap[1..m]$

Output: Il valore ottimo e la matrice delle informazioni $r[0..m][0..D_{1,m}]$

```
1  Function DP_CLS_InfoSol(m, d, sc, vc, hc, cap)
2      R := new array[0..m]
3      R[m] ← 0
4      for t := m − 1 downto 0 do
5          R[t] ← R[t + 1] + d[t + 1]
6      end
7      v := new array[0..m][0..R[0]]
8      r := new array[0..m][0..R[0]]
9      v[0][0] ← 0
10     r[0][0] ← 0
11     for k := 1 to R[0] do
12         v[0][k] ← +∞
13         r[0][k] ← 0
14     end
15     for t := 1 to m do
16         for k := 0 to R[t] do
17             b := +∞
18             a := 0
19             for x := 0 to min{cap[t], k + d[t]} do
20                 if v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k < b then
21                     b ← v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k
22                     a ← x
23                 end
24             end
25             v[t][k] ← b
26             r[t][k] ← a
27         end
28     end
29     return v[m][0], r
```

∗ L'algoritmo di PD usa la variabile temporanea aggiuntiva a per tenere traccia della decisione presa quando b viene aggiornato: se viene scelta la produzione x, $a \leftarrow x$ memorizza questa decisione, e alla fine del ciclo interno $r[t][k] \leftarrow a$ la

memorizza, insieme al valore ottimo $v[t][k] \leftarrow b$. A causa del confronto stretto, in caso di pareggio viene memorizzata la produzione più piccola.

Esempio 11.3: calcolo delle informazioni aggiuntive

Consideriamo l'istanza del CLS dell'Esempio 11.1:

t	1	2	3	4	5
d_t	2	1	3	4	1
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	10	10	10	10	10
cap_t	4	4	4	4	4

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di t e k, la cella $r[t][k]$ e la produzione x per cui il minimo della riga corrispondente dell'Esempio 11.2 viene raggiunto per primo.

t	k	$r[t][k]$	Azione
0	0	0	$r[0][0] \leftarrow 0$
0	1	0	$r[0][1] \leftarrow 0$
0	2	0	$r[0][2] \leftarrow 0$
0	3	0	$r[0][3] \leftarrow 0$
0	4	0	$r[0][4] \leftarrow 0$
0	5	0	$r[0][5] \leftarrow 0$
0	6	0	$r[0][6] \leftarrow 0$
0	7	0	$r[0][7] \leftarrow 0$
0	8	0	$r[0][8] \leftarrow 0$
0	9	0	$r[0][9] \leftarrow 0$
0	10	0	$r[0][10] \leftarrow 0$
0	11	0	$r[0][11] \leftarrow 0$
1	0	2	primo minimo in $x = 2$, ossia $v[0][0]+f_1(2)+hc_1 \cdot 0 = 0+50+0 = 50$: $r[1][0] \leftarrow 2$
1	1	3	primo minimo in $x = 3$, ossia $v[0][0]+f_1(3)+hc_1 \cdot 1 = 0+50+10 = 60$: $r[1][1] \leftarrow 3$
1	2	4	primo minimo in $x = 4$, ossia $v[0][0]+f_1(4)+hc_1 \cdot 2 = 0+50+20 = 70$: $r[1][2] \leftarrow 4$
1	3	0	tutti i candidati valgono $+\infty$: $r[1][3] \leftarrow 0$
1	4	0	tutti i candidati valgono $+\infty$: $r[1][4] \leftarrow 0$
1	5	0	tutti i candidati valgono $+\infty$: $r[1][5] \leftarrow 0$
1	6	0	tutti i candidati valgono $+\infty$: $r[1][6] \leftarrow 0$
1	7	0	tutti i candidati valgono $+\infty$: $r[1][7] \leftarrow 0$
1	8	0	tutti i candidati valgono $+\infty$: $r[1][8] \leftarrow 0$
1	9	0	tutti i candidati valgono $+\infty$: $r[1][9] \leftarrow 0$
2	0	0	primo minimo in $x = 0$, ossia $v[1][1]+f_2(0)+hc_2 \cdot 0 = 60+0+0 = 60$: $r[2][0] \leftarrow 0$
2	1	0	primo minimo in $x = 0$, ossia $v[1][2]+f_2(0)+hc_2 \cdot 1 = 70+0+10 = 80$: $r[2][1] \leftarrow 0$
2	2	3	primo minimo in $x = 3$, ossia $v[1][0]+f_2(3)+hc_2 \cdot 2 = 50+50+20 = 120$: $r[2][2] \leftarrow 3$
2	3	4	primo minimo in $x = 4$, ossia $v[1][0]+f_2(4)+hc_2 \cdot 3 = 50+50+30 = 130$: $r[2][3] \leftarrow 4$

2	4	4	primo minimo in $x = 4$, ossia $v[1][1]+f_2(4)+hc_2 \cdot 4 = 60+50+40 = 150$: $r[2][4] \leftarrow 4$
2	5	4	primo minimo in $x = 4$, ossia $v[1][2]+f_2(4)+hc_2 \cdot 5 = 70+50+50 = 170$: $r[2][5] \leftarrow 4$
2	6	0	tutti i candidati valgono $+\infty$: $r[2][6] \leftarrow 0$
2	7	0	tutti i candidati valgono $+\infty$: $r[2][7] \leftarrow 0$
2	8	0	tutti i candidati valgono $+\infty$: $r[2][8] \leftarrow 0$
3	0	3	primo minimo in $x = 3$, ossia $v[2][0]+f_3(3)+hc_3 \cdot 0 = 60+50+0 = 110$: $r[3][0] \leftarrow 3$
3	1	4	primo minimo in $x = 4$, ossia $v[2][0]+f_3(4)+hc_3 \cdot 1 = 60+50+10 = 120$: $r[3][1] \leftarrow 4$
3	2	4	primo minimo in $x = 4$, ossia $v[2][1]+f_3(4)+hc_3 \cdot 2 = 80+50+20 = 150$: $r[3][2] \leftarrow 4$
3	3	4	primo minimo in $x = 4$, ossia $v[2][2]+f_3(4)+hc_3 \cdot 3 = 120+50+30 = 200$: $r[3][3] \leftarrow 4$
3	4	4	primo minimo in $x = 4$, ossia $v[2][3]+f_3(4)+hc_3 \cdot 4 = 130+50+40 = 220$: $r[3][4] \leftarrow 4$
3	5	4	primo minimo in $x = 4$, ossia $v[2][4]+f_3(4)+hc_3 \cdot 5 = 150+50+50 = 250$: $r[3][5] \leftarrow 4$
4	0	4	primo minimo in $x = 4$, ossia $v[3][0]+f_4(4)+hc_4 \cdot 0 = 110+50+0 = 160$: $r[4][0] \leftarrow 4$
4	1	4	primo minimo in $x = 4$, ossia $v[3][1]+f_4(4)+hc_4 \cdot 1 = 120+50+10 = 180$: $r[4][1] \leftarrow 4$
5	0	0	primo minimo in $x = 0$, ossia $v[4][1]+f_5(0)+hc_5 \cdot 0 = 180+0+0 = 180$: $r[5][0] \leftarrow 0$

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

$r[t][k]$	0	1	2	3	4	5	6	7	8	9	10	11
$t = 0$	0	0	0	0	0	0	0	0	0	0	0	0
$t = 1$	2	3	4	0	0	0	0	0	0	0	·	·
$t = 2$	0	0	3	4	4	4	0	0	0	·	·	·
$t = 3$	3	4	4	4	4	4	·	·	·	·	·	·
$t = 4$	4	4	·	·	·	·	·	·	·	·	·	·
$t = 5$	0	·	·	·	·	·	·	·	·	·	·	·

– Una volta calcolata la matrice r, si può calcolare una soluzione ottima dell'istanza del CLS applicando la regola seguente per $t = m, m-1, \ldots, 1$, a partire da $k = 0$:

$$x_t \leftarrow r[t][k], \qquad k \leftarrow k + d_t - x_t.$$

Si ottiene il vettore $(x_1, x_2, \ldots, x_m)$, in cui ogni componente x_t è il numero di unità prodotte nel periodo t in una soluzione ottima.

– Il seguente pseudocodice implementa il calcolo di una soluzione ottima, data la matrice r.

Algoritmo

Algorithm: DP_CLS_SolComputation

Input: Il numero m di periodi e le domande $d[1..m]$ dell'istanza del CLS, e la matrice $r[0..m][0..D_{1,m}]$

Output: Una soluzione ottima dell'istanza del CLS

```
1 Function DP_CLS_SolComputation(m, d, r)
2     x := new array[1..m]
3     k := 0
4     for t := m downto 1 do
5         x[t] ← r[t][k]
6         k ← k + d[t] − x[t]
7     end
8     return x
```

Esempio 11.4: calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 11.1, abbiamo la matrice r dell'Esempio 11.3. Possiamo calcolare una soluzione ottima come segue:

Iterazione	t	k	$r[t][k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$k \leftarrow k + d[t] - x[t]$
–	–	–	–	0	0	0	0	0	$t = 5,\ k = 0$
1	5	0	0	0	0	0	0	0	$k = 0 + 1 - 0 = 1$
2	4	1	4	0	0	0	4	0	$k = 1 + 4 - 4 = 1$
3	3	1	4	0	0	4	4	0	$k = 1 + 3 - 4 = 0$
4	2	0	0	0	0	4	4	0	$k = 0 + 1 - 0 = 1$
5	1	1	3	3	0	4	4	0	$k = 1 + 2 - 3 = 0$

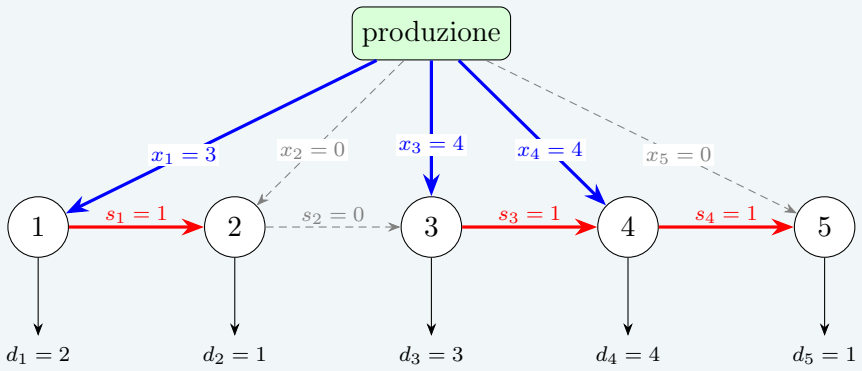
Una soluzione ottima è il vettore:

$$(3, 0, 4, 4, 0)$$

I setup sono nei periodi 1, 3, 4, le scorte sono $(s_1, \ldots, s_5) = (1, 0, 1, 1, 0)$, e il costo totale è (non c'è costo variabile di produzione)

$$\underbrace{50 + 50 + 50}_{\text{setup}} + \underbrace{10 \cdot 1 + 10 \cdot 0 + 10 \cdot 1 + 10 \cdot 1 + 10 \cdot 0}_{\text{mantenimento}} = 150 + 30 = 180.$$

Il valore ottimo corrispondente è $v[5][0] = 180$, lo stesso valore dato da Gurobi sul modello ILP_{CLSP}; la capacità costa $180 - 170$ rispetto all'istanza senza capacità. La rete della soluzione è:



Il periodo 4 produce anche se alla fine del periodo 3 c'è un'unità in scorta: la soluzione non ha la proprietà zero-inventory ordering (ed è l'unica soluzione ottima con quantità intere).

11.2.2 Numero di passi

- Indichiamo con $S(m, D, C)$ il numero di passi eseguiti da DP_CLS nel caso peggiore, dove m è il numero di periodi, $D = D_{1,m}$ è la domanda totale e $C = \max_{t \in T} cap_t$ è la capacità più grande. Per ogni $t \geq 1$ abbiamo $R[t] + 1 \leq D$ (poiché $d_1 \geq 1$), e il ciclo interno ha al più $C + 1$ iterazioni. Nel caso peggiore, la condizione della riga 16 è sempre vera. Il numero di esecuzioni delle righe è:
  - Le righe 2, 3, 7 e 8 vengono eseguite 1 volta ciascuna.
  - Le righe 4 e 5 vengono eseguite m volte ciascuna.
  - Le righe 9 e 10 vengono eseguite D volte ciascuna.
  - La riga 12 viene eseguita m volte.
  - Le righe 13, 14 e 20 vengono eseguite $\sum_{t=1}^{m}(R[t] + 1) \leq mD$ volte ciascuna.
  - Le righe 15, 16 e 17 vengono eseguite al più $mD(C + 1)$ volte ciascuna.
  - La riga 23 viene eseguita 1 volta.

  Quindi,

$$S(m, D, C) \leq 5 + 3m + 2D + 3mD + 3mD(C + 1) = 5 + 3m + 2D + 6mD + 3mDC,$$

  e di conseguenza

$$S(m, D, C) = O(mDC).$$

L'algoritmo di PD esegue

$$S(m, D, C) = O(m\,D\,C)$$

passi nel caso peggiore. Il numero di passi dipende dai valori D e C, non solo dal numero m di periodi: l'algoritmo di PD è **pseudo-polinomiale**.

- DP_CLS_InfoSol aggiunge solo un numero costante di operazioni dentro gli stessi cicli, quindi il suo numero di passi ha lo stesso ordine di crescita.

- *Esempio.* Consideriamo l'istanza capacitata con 6 periodi, domande $(50, 30, 10, 10, 40, 10)$, costo di setup 40, costo di mantenimento 1 e capacità 80. Misurata in unità di 10, ha 35 stati con $t \geq 1$.

- L'algoritmo di PD dà il valore ottimo 160, con il piano di produzione $(50, 50, 0, 0, 50, 0)$ (nelle unità originali): lo stesso valore dato da Gurobi sul modello ILP_{CLSP}.

11.2.3 Il caso a capacità costante

- Con capacità cap_t generali il CLS è NP-difficile, quindi non ci si aspetta un algoritmo polinomiale e l'algoritmo di PD pseudo-polinomiale è un metodo esatto naturale per piccoli valori di D e C.

- Quando la capacità è la stessa in tutti i periodi, $cap_t = C$ per ogni $t \in T$, il CLS si può risolvere in tempo polinomiale, perché la proprietà zero-inventory ordering è sostituita da una più debole:

Se $cap_t = C$ per ogni $t \in T$, esiste una soluzione ottima in cui, fra due periodi consecutivi con scorta vuota, al più un periodo produce una quantità strettamente compresa fra 0 e C; tutti gli altri periodi producono 0 o C.

- Il CLS a capacità costante ha la sua pagina nel catalogo dei problemi.

- L'Esempio 11.1 ha capacità costante $C = 4$. La sua soluzione ottima $(3, 0, 4, 4, 0)$ ha scorte $(1, 0, 1, 1, 0)$, vuote alla fine dei periodi 0, 2 e 5.

- Nei periodi 1–2 solo il periodo 1 produce una quantità strettamente compresa fra 0 e 4, e nei periodi 3–5 ogni periodo produce 0 o 4, come afferma la proprietà.

11.3 Errori tipici

Attenzione

- **Applicare la proprietà zero-inventory ordering con le capacità.** Con le capacità un periodo può produrre mentre ci sono unità in scorta: nell'Esempio 11.1 l'unico piano ottimo viola la proprietà.

- **Dimenticare i casi base con valore $+\infty$.** La scorta iniziale è vuota, quindi $v(0, k) = +\infty$ per $k > 0$; questo valore marca ogni stato che non si può raggiungere entro le capacità.

- **Ammettere una scorta troppo grande.** Alla fine del periodo t la scorta è al più $D_{t+1,m}$, poiché deve essere vuota alla fine del periodo m: le altre coppie (t, k) non sono stati.

- **Mettere il costo di mantenimento dentro la scelta.** Il costo $hc_t\, k$ della scorta

finale si paga nello stato (t, k) qualunque sia la produzione x: si aggiunge fuori dal minimo.

- **Chiamare polinomiale $O(mDC)$.** Il tempo di calcolo contiene la domanda totale D e la capacità più grande C, che sono numeri dell'input: l'algoritmo di PD è pseudo-polinomiale.

11.4 In sintesi

In sintesi

- Con le capacità la proprietà zero-inventory ordering non vale: un periodo può produrre mentre ci sono unità in scorta, e l'algoritmo di PD sui lotti di produzione non si applica.
- Lo stato è la coppia (t, k) del periodo e della scorta finale, con $k \leq D_{t+1,m}$; la ricorrenza $v(t, k)$ sceglie la produzione x del periodo t, e il valore ottimo è $v(m, 0)$.
- La matrice r delle produzioni scelte, letta all'indietro a partire dallo stato $(m, 0)$, dà un piano ottimo.
- L'algoritmo di PD richiede $O(mDC)$ passi, quindi è pseudo-polinomiale.
- Con una capacità costante vale una proprietà più debole, e il problema è polinomiale; con capacità generali è NP-difficile.

Quadro riassuntivo dell'esempio.

Istanza	Lower bound	Upper bound	Origine
Esempio 11.1 (capacitata, $C = 4$)	180	180	PD su (periodo, scorta)

Problemi misti

Capitolo 12

PD per il problema del cambio monete

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 (Capitolo 2) e la dimensione dell'input (Capitolo 11).

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.

Il problema di questo capitolo è il problema del cambio monete (CCP): la sua definizione è nella sua pagina.

12.1 Istanze e soluzioni ammissibili

Esempio 12.1: 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 Esempio 12.1.

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.

12.1.1 Codifica delle 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.

- 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:

  $$x_j = \text{numero di monete del taglio } j \text{ usate}, \qquad \forall j \in \{1, 2, \ldots, n\}, \quad x_j \in \mathbb{Z}_{\geq 0}.$$

  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

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

- 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, ed enumerarle tutte per trovare quella con meno monete diventa presto impraticabile.

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

$$v(k) = \begin{cases} 0 & \text{se } k = 0, \\ +\infty & \text{se } k \geq 1 \text{ e } w_j > k \text{ per ogni } j \in \{1, 2, \ldots, n\}, \\ \min\limits_{j\,:\,w_j \leq k} \{1 + v(k - w_j)\} & \text{se } k \geq 1 \text{ e } w_j \leq k \text{ per qualche } j \in \{1, 2, \ldots, n\}. \end{cases} \tag{12.1}$$

- Il processo di decomposizione ricorsiva della formula (12.1) 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, e $v(k) = +\infty$ se nessun taglio è abbastanza piccolo da essere usato, perché in questo caso il resto k non si può restituire affatto.
- 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!**

- 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 formula (12.1).

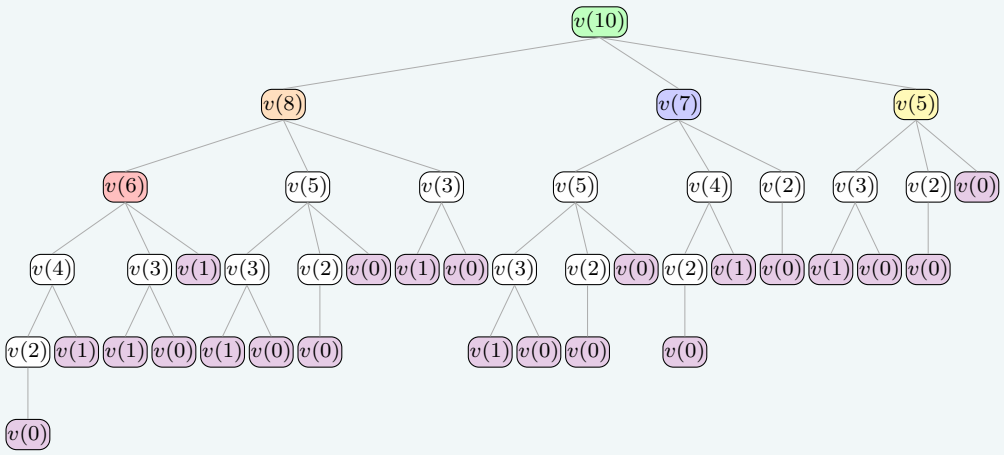
  $\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

– 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 formula (12.1): 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 12.2: Albero di ricorsione

Consideriamo l'istanza del CCP dell'Esempio 12.1. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(10)$:



La funzione v viene chiamata 42 volte, e gli stessi sottoproblemi vengono risolti più volte: $v(0)$ viene chiamata 14 volte, $v(1)$ 8 volte, $v(2)$ 6 volte, $v(3)$ 5 volte, $v(4)$ 2 volte e $v(5)$ 3 volte. Tutte le altre chiamate vengono fatte una sola volta.

– 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 12.3: Calcolo dei valori della funzione v

Consideriamo l'istanza del CCP dell'Esempio 12.1, 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 $v(2)$:

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

* 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 $v(5)$:

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

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

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

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

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

  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, $w_1 = 2$.

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

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

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

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

– 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.

– 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 formula (12.1) 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.

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

Algoritmo

Algorithm: CCP_DP

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
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]
```

* 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 12.4: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza del CCP dell'Esempio 12.1, 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.

k	j	$v[k]$	m	Azione
–	–	0	–	$v[0] \leftarrow 0$
1	–	–	$+\infty$	$m := +\infty$
	1	–	$+\infty$	$w[1] = 2 > 1$: si salta
	2	–	$+\infty$	$w[2] = 3 > 1$: si salta
	3	–	$+\infty$	$w[3] = 5 > 1$: si salta
	–	$+\infty$	$+\infty$	$v[1] \leftarrow m = +\infty$
2	–	–	$+\infty$	$m := +\infty$
	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	$v[2] \leftarrow m = 1$
3	–	–	$+\infty$	$m := +\infty$
	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	$v[3] \leftarrow m = 1$
4	–	–	$+\infty$	$m := +\infty$
	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	$v[4] \leftarrow m = 2$
5	–	–	$+\infty$	$m := +\infty$
	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	$v[5] \leftarrow m = 1$
6	–	–	$+\infty$	$m := +\infty$
	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	$v[6] \leftarrow m = 2$
7	–	–	$+\infty$	$m := +\infty$
	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	$v[7] \leftarrow m = 2$
8	–	–	$+\infty$	$m := +\infty$
	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	$v[8] \leftarrow m = 2$
9	–	–	$+\infty$	$m := +\infty$
	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	$v[9] \leftarrow m = 3$
10	–	–	$+\infty$	$m := +\infty$
	1	–		- 3
	2	–		nessun aggiornamento
	3	–		- 2
	–	2	2	$v[10] \leftarrow m = 2$

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 $k \in \{0, 1, \ldots, c\}$, dove $v[c]$ è il numero minimo di monete necessarie per restituire il resto c.

- 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 formula (12.1).
  - Più precisamente, $s[k]$ è uguale all'indice j del taglio w_j che minimizza il membro destro della formula (12.1) nel calcolo di $v[k]$.

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

Algoritmo

Algorithm: CCP_DP_InfoSol

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
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
```

  * 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 12.5: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza del CCP dell'Esempio 12.1, 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 formula (12.1) 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.

– 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 $v[c] < +\infty$ 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.

Algoritmo

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
```

Esempio 12.6: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 12.1, 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	$k = 5 - 5 = 0$

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 $v(10) = 2$.

12.2.1 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. 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.

  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,

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

  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 $\Theta(nc)$.

12.3 Errori tipici

Attenzione

- **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 $v[c] < +\infty$.
- **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.

12.4 In sintesi

In sintesi

- 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 $v(k) = \min_{j:\, w_j \le k}\{1 + v(k - w_j)\}$ 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 $v[c] < +\infty$.

Quadro riassuntivo dell'esempio del CCP. Esempio 12.1 ($c = 10$, $w = (2, 3, 5)$):

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

Capitolo 13

PD per il problema del taglio delle aste

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.
- Da questo modulo, il capitolo «PD per il problema dello zaino 0/1»: le coppie (oggetto, capacità) e la ricostruzione di una soluzione, che ritornano qui.
- Dal modulo *Algoritmi*: come leggere lo pseudocodice (Capitolo 2) e come contare i passi di un algoritmo (Capitolo 12 «Tempo di esecuzione»).

Obiettivi

Alla fine di questo capitolo saremo in grado di:

- enunciare l'RCP, codificare una soluzione come vettore di numeri di pezzi e spiegare perché schemi di taglio e vettori sono oggetti diversi;
- scrivere la ricorrenza $v(\ell)$ e spiegarla, caso base compreso;
- eseguire a mano l'algoritmo di PD con l'approccio top-down e con quello bottom-up, e con il miglior valore corrente m;
- ricostruire una soluzione ottima dal vettore s, per l'asta e per ogni asta più corta;
- scrivere la seconda ricorrenza $v(j, \ell)$ sulle coppie (lunghezza massima dei pezzi, lunghezza dell'asta) e contare i passi dell'algoritmo di PD.

Il problema di questo capitolo è il problema del taglio delle aste (RCP): la sua definizione è nella sua pagina.

13.1 Istanze e soluzioni ammissibili

Esempio 13.1: Problema del taglio delle aste

Consideriamo un'asta di lunghezza $n = 4$ e un insieme $\{1, 2, 3, 4\}$ di 4 possibili lunghezze dei pezzi con i seguenti profitti:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

Chiamiamo questa istanza dell'RCP Esempio 13.1.

Il numero di possibili soluzioni dell'RCP è 2^{n-1}, perché possiamo tagliare un'asta di lunghezza n in 2^{n-1} modi diversi, con una scelta indipendente di tagliare (o no) in ogni possibile posizione di taglio.

- Il numero di possibili soluzioni dell'RCP cresce esponenzialmente con la lunghezza n dell'asta:
$$2^{n-1} = O(2^n).$$
Quindi enumerare tutte le possibili soluzioni diventa presto impraticabile al crescere di n.

13.1.1 Codifica delle soluzioni ammissibili

Una *soluzione ammissibile* dell'RCP è un modo di tagliare l'asta in pezzi, tale che la lunghezza totale dei pezzi sia uguale alla lunghezza dell'asta.

- Una soluzione ammissibile dell'RCP si può descrivere con un vettore soluzione intero
$$(x_1, x_2, \ldots, x_n)$$
dove ogni componente ha il seguente significato:
$$x_\ell = \text{numero di pezzi di lunghezza } \ell, \qquad \forall \ell \in \{1, 2, \ldots, n\};$$
Il numero di pezzi è
$$\sum_{\ell=1}^{n} x_\ell.$$
Poiché la soluzione è ammissibile, la lunghezza totale dei pezzi deve essere uguale alla lunghezza dell'asta:
$$\sum_{\ell=1}^{n} \ell\, x_\ell = n.$$
Il profitto totale di una soluzione ammissibile è
$$\sum_{\ell=1}^{n} p_\ell x_\ell.$$

- Due o più schemi di taglio possono avere la stessa rappresentazione vettoriale: il vettore registra quanti pezzi di ogni lunghezza si ottengono, ma non specifica l'ordine in cui i pezzi vengono tagliati.

- Schemi di taglio con la stessa rappresentazione vettoriale hanno lo stesso profitto totale, perché l'ordine in cui i pezzi vengono tagliati non cambia i pezzi ottenuti e quindi non cambia il profitto totale.

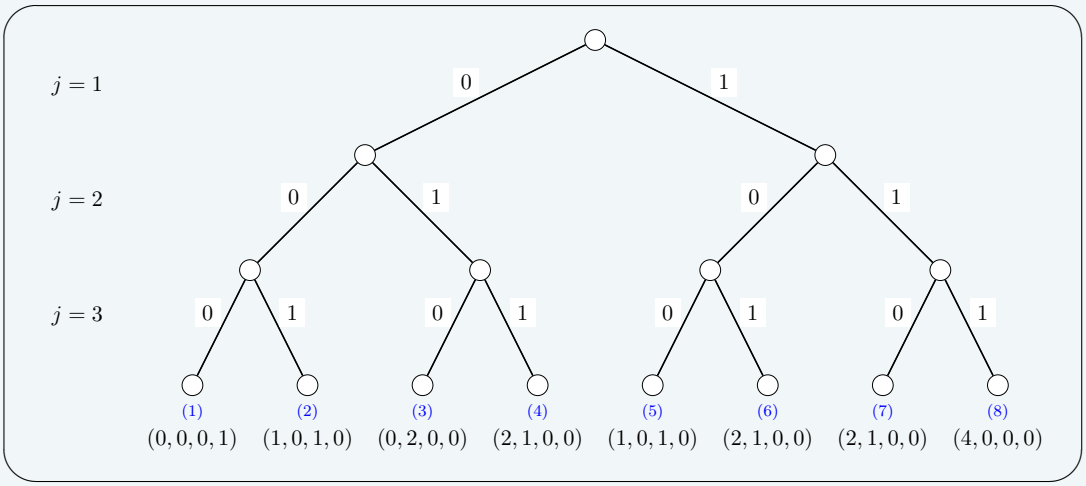
Esempio 13.2: Enumerazione di tutte le possibili soluzioni dell'RCP

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n = 4$; i profitti associati alle possibili lunghezze dei pezzi sono:

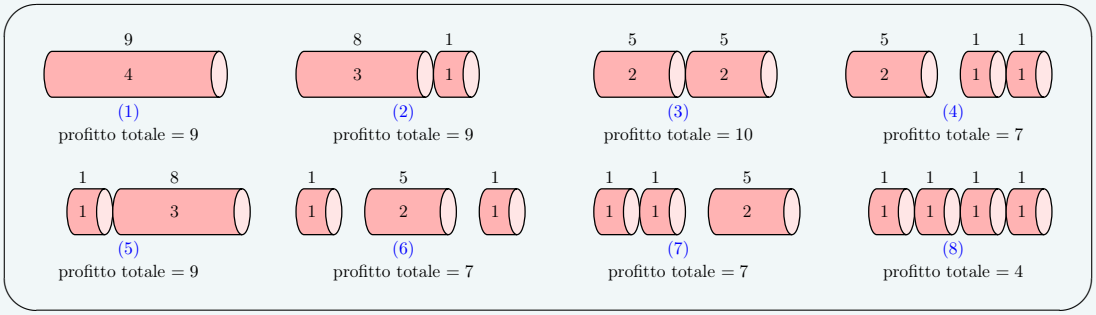
	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

Ci sono $2^{4-1} = 8$ soluzioni ammissibili, corrispondenti ai diversi modi di tagliare l'asta.

Nel seguente albero binario ogni livello corrisponde a una possibile posizione di taglio: un arco con etichetta 1 indica che l'asta viene tagliata a quella lunghezza, mentre un arco con etichetta 0 indica che non viene tagliata a quella lunghezza. Ogni foglia rappresenta uno schema di taglio completo e riporta il vettore corrispondente.



La figura seguente mostra tutti i possibili schemi di taglio associati alle foglie dell'albero precedente, tramite le etichette $(1), (2), \ldots, (8)$. Per ogni pezzo ottenuto, il numero scritto dentro il pezzo è la sua lunghezza, mentre il numero sopra è il suo profitto.



Etichetta	Vettore soluzione	Numero di pezzi	Profitto totale
(1)	$(0,0,0,1)$	1	9
(2)	$(1,0,1,0)$	2	$8+1=9$
(3)	$(0,2,0,0)$	2	$5+5=10$
(4)	$(2,1,0,0)$	3	$5+1+1=7$
(5)	$(1,0,1,0)$	2	$1+8=9$
(6)	$(2,1,0,0)$	3	$1+5+1=7$
(7)	$(2,1,0,0)$	3	$1+1+5=7$
(8)	$(4,0,0,0)$	4	$1+1+1+1=4$

13.2 Un primo algoritmo di PD

- Passo 1: calcolo del valore ottimo

Per ogni $\ell \in \{0,1,\ldots,n\}$, sia $v(\ell)$ il valore ottimo della sottoistanza dell'RCP data da:

1. un'asta di lunghezza ℓ,
2. un insieme $\{1,2,\ldots,\ell\}$ di ℓ possibili lunghezze dei pezzi,
3. per ogni lunghezza dei pezzi $j \in \{1,2,\ldots,\ell\}$, un profitto p_j.

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

$$v(\ell) = \begin{cases} \max\left\{p_j + v(\ell-j) : j \in \{1,2,\ldots,\ell\}\right\} & \text{se } 1 \le \ell \le n, \\ 0 & \text{se } \ell = 0. \end{cases} \tag{13.1}$$

  - Il processo di decomposizione ricorsiva della formula (13.1) genera ℓ **sottoproblemi indipendenti**, ciascuno corrispondente a una possibile scelta di tagliare un primo pezzo di lunghezza j da un'asta di lunghezza ℓ. Qui $v(0) = 0$, perché un'asta di lunghezza 0 non si può tagliare e ha profitto 0.
  - Il valore ottimo $v(\ell)$ si ottiene provando tutte le possibili lunghezze del primo pezzo e scegliendo quella che massimizza la somma $p_j + v(\ell-j)$ del profitto del primo pezzo e del valore ottimo del sottoproblema residuo.
  - Quindi una soluzione ottima per un'asta di lunghezza ℓ si costruisce combinando il primo pezzo scelto, di lunghezza j, con una soluzione ottima del sottoproblema residuo di lunghezza $\ell - j$.

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

– Scelte diverse della lunghezza del primo pezzo, eventualmente fatte per valori diversi di ℓ, possono portare alla stessa lunghezza residua; quindi lo stesso valore $v(\ell)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (13.1).

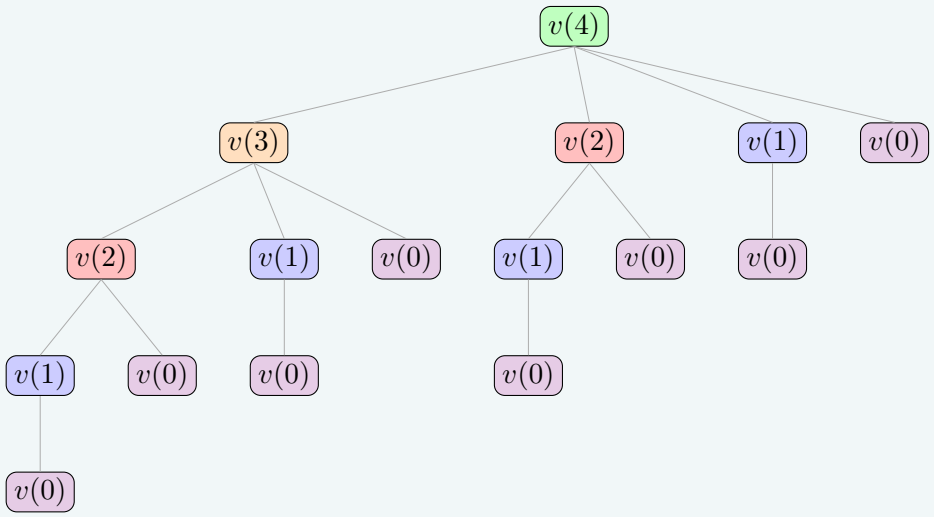
$\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

– Un **primo metodo** per calcolare il valore ottimo $v(n)$ dell'istanza dell'RCP è l'**approccio top-down**. Il metodo parte dalla chiamata con $\ell = n$ e valuta ricorsivamente la formula (13.1): ogni chiamata per un'asta di lunghezza ℓ genera altre chiamate per aste di lunghezza $\ell - j$, per ogni $j \in \{1, 2, \ldots, \ell\}$, fino a raggiungere il caso base $v(0)$.

– 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 13.3: Albero di ricorsione

Consideriamo l'istanza dell'RCP dell'Esempio 13.1. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(4)$:



La funzione v viene chiamata 16 volte, e gli stessi sottoproblemi vengono risolti più volte: $v(2)$ viene chiamata 2 volte, $v(1)$ viene chiamata 4 volte e $v(0)$ viene chiamata 8 volte. Tutte le altre chiamate vengono fatte una sola volta.

– Il valore $v(\ell)$, per una generica lunghezza dell'asta ℓ, si può calcolare una volta noti i valori $v(j)$ per tutte le lunghezze minori $j < \ell$.

Esempio 13.4: Calcolo dei valori della funzione v

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n = 4$; i profitti associati alle possibili lunghezze dei pezzi sono:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

Il valore della funzione v si può calcolare partendo dal caso base $v(0) = 0$ e calcolando poi, in ordine, i valori di $v(1)$, $v(2)$, $v(3)$ e $v(4)$, come segue:

* Poiché conosciamo il valore di $v(0)$, possiamo calcolare $v(1)$ come segue:

$$v(1) = p_1 + v(0) = 1 + 0 = 1$$

* Calcolato $v(1)$, possiamo calcolare $v(2)$ come segue:

$$v(2) = \max\{p_1 + v(1), p_2 + v(0)\} = \max\{1 + 1, 5 + 0\} = 5$$

* Calcolati $v(2)$ e $v(1)$, possiamo calcolare $v(3)$ come segue:

$$v(3) = \max\{p_1 + v(2), p_2 + v(1), p_3 + v(0)\} = \max\{1 + 5, 5 + 1, 8 + 0\} = 8$$

* Calcolati $v(3)$, $v(2)$ e $v(1)$, possiamo calcolare $v(4)$ come segue:

$$\begin{aligned} v(4) &= \max\{p_1 + v(3), p_2 + v(2), p_3 + v(1), p_4 + v(0)\} \\ &= \max\{1 + 8, 5 + 5, 8 + 1, 9 + 0\} = 10 \end{aligned}$$

– 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 un sottoproblema di valore $v(\ell)$ dipende solo dai sottoproblemi di valore $v(j)$, con $j < \ell$, possiamo risolvere i sottoproblemi per valori crescenti di ℓ, partendo dal caso base $v(0) = 0$ fino ad arrivare a $v(n)$: questa osservazione porta in modo naturale a un **metodo molto più efficiente** per calcolare il valore ottimo $v(n)$ dell'istanza dell'RCP.

– Un **secondo metodo** per calcolare il valore ottimo $v(n)$ dell'istanza dell'RCP è l'**approccio bottom-up**. Il metodo parte dal caso base $v(0) = 0$ e valuta la formula (13.1) per lunghezze crescenti dell'asta $\ell \in \{1, 2, \ldots, n\}$: ogni valore $v(\ell)$ si calcola con i valori già calcolati $v(0), v(1), \ldots, v(\ell - 1)$, fino a ottenere il valore $v(n)$ dell'istanza dell'RCP.

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

Algoritmo

Algorithm: DP1

Input: Un'istanza dell'RCP data dalla lunghezza n dell'asta e da un array $p[1..n]$ dei profitti dei pezzi

Output: Il valore ottimo dell'istanza dell'RCP

```
1  Function DP1(n, p)
2      v := new array[0..n]
3      v[0] ← 0
4      for ℓ := 1 to n do
5          m := 0
6          for j := 1 to ℓ do
7              if m < p[j] + v[ℓ − j] then
8                  m ← p[j] + v[ℓ − j]
9              end
10         end
11         v[ℓ] ← m
12     end
13     return v[n]
```

* Il vettore $v[0..n]$ è la **tabella di lookup**: memorizza i valori ottimi di tutti i sottoproblemi, con il caso base $v[0] = 0$.
* Il ciclo esterno, su $\ell \in \{1, 2, \ldots, n\}$, calcola $v[\ell]$ in ordine crescente di lunghezza dell'asta.
* Il ciclo interno, su $j \in \{1, 2, \ldots, \ell\}$, valuta tutte le possibili lunghezze del primo pezzo e tiene in m il valore massimo di $p[j] + v[\ell - j]$.
* Alla fine del ciclo interno, $v[\ell] \leftarrow m$ memorizza il valore ottimo $v[\ell]$.
* Infine, l'algoritmo restituisce il valore ottimo $v[n]$ dell'istanza dell'RCP.

Esempio 13.5: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n = 4$; i profitti associati alle possibili lunghezze dei pezzi sono:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

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

ℓ	j	$v[\ell]$	m	Azione
–	–	0	–	$v[0] \leftarrow 0$
1	–	–	0	$m \leftarrow 0$
	1	–	1	$p[1]+v[0] = 1+0 = 1 > 0$: $m \leftarrow 1$
	–	1	1	$v[1] \leftarrow m = 1$
2	–	–	0	$m \leftarrow 0$
	1	–	2	$p[1]+v[1] = 1+1 = 2 > 0$: $m \leftarrow 2$
	2	–	5	$p[2]+v[0] = 5+0 = 5 > 2$: $m \leftarrow 5$
	–	5	5	$v[2] \leftarrow m = 5$
3	–	–	0	$m \leftarrow 0$
	1	–	6	$p[1]+v[2] = 1+5 = 6 > 0$: $m \leftarrow 6$
	2	–	6	$p[2]+v[1] = 5+1 = 6 = m$: nessun aggiornamento
	3	–	8	$p[3]+v[0] = 8+0 = 8 > 6$: $m \leftarrow 8$
	–	8	8	$v[3] \leftarrow m = 8$
4	–	–	0	$m \leftarrow 0$
	1	–	9	$p[1]+v[3] = 1+8 = 9 > 0$: $m \leftarrow 9$
	2	–	10	$p[2]+v[2] = 5+5 = 10 > 9$: $m \leftarrow 10$
	3	–	10	$p[3]+v[1] = 8+1 = 9 < 10$: nessun aggiornamento
	4	–	10	$p[4]+v[0] = 9+0 = 9 < 10$: nessun aggiornamento
	–	10	10	$v[4] \leftarrow m = 10$

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

	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$v[\ell]$	0	1	5	8	10

  - Dopo l'esecuzione dell'algoritmo di PD, il vettore v contiene il valore ottimo $v[\ell]$ per ogni lunghezza dell'asta.

- Passo 2: calcolo di una soluzione ottima
  - Per calcolare una soluzione ottima è necessario memorizzare alcune informazioni aggiuntive.
  - Per ogni lunghezza dell'asta ℓ, oltre al valore ottimo $v[\ell]$, memorizziamo $s[\ell]$, la lunghezza del primo pezzo da tagliare in una soluzione ottima per un'asta di lunghezza ℓ.
  - Partendo da $\ell = n$, il valore $s[\ell]$ dà la lunghezza del primo pezzo da tagliare; l'asta rimanente ha lunghezza $\ell - s[\ell]$, e la stessa regola si può applicare di nuovo, e così via, finché la lunghezza residua arriva a 0.

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

Algoritmo

Algorithm: DP1_InfoSol

Input: Un'istanza dell'RCP data dalla lunghezza n dell'asta e da un array $p[1..n]$ dei profitti dei pezzi

Output: Il valore ottimo dell'istanza dell'RCP e il vettore di informazioni $s[1..n]$

```
1  Function DP1_InfoSol(n, p)
2      v := new array[0..n]
3      s := new array[1..n]
4      v[0] ← 0
5      for ℓ := 1 to n do
6          m := 0
7          t := 0
8          for j := 1 to ℓ do
9              if m < p[j] + v[ℓ − j] then
10                 m ← p[j] + v[ℓ − j]
11                 t ← j
12             end
13         end
14         v[ℓ] ← m
15         s[ℓ] ← t
16     end
17     return v[n], s
```

∗ L'algoritmo di PD usa la variabile temporanea aggiuntiva t per tenere traccia della decisione presa quando m viene aggiornato. Se si sceglie un primo pezzo di lunghezza j, $t \leftarrow j$ memorizza questa decisione. Alla fine del ciclo interno, $s[\ell] \leftarrow t$ memorizza la lunghezza del primo pezzo scelto, insieme al valore ottimo $v[\ell] \leftarrow m$.

Esempio 13.6: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n = 4$; i profitti associati alle possibili lunghezze dei pezzi sono:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

Per ogni passo dell'algoritmo di PD, la tabella riporta i valori di ℓ e j, la cella $s[\ell]$, il valore corrente di m e il valore corrente di t. La cella di m è evidenziata in rosso quando viene aggiornata; la cella di t e la cella di $s[\ell]$ sono evidenziate in blu quando vengono aggiornate.

ℓ	j	$s[\ell]$	m	t	Azione
–	–	–	–	–	Stato iniziale
	–	–	0	0	$m \leftarrow 0$, $t \leftarrow 0$
1	1	–	1	1	$1{+}0 = 1 > 0$: $m \leftarrow 1$, $t \leftarrow 1$
	–	1	1	1	$v[1] \leftarrow m = 1$, $s[1] \leftarrow t = 1$
	–	–	0	0	$m \leftarrow 0$, $t \leftarrow 0$
2	1	–	2	1	$1{+}1 = 2 > 0$: $m \leftarrow 2$, $t \leftarrow 1$
	2	–	5	2	$5{+}0 = 5 > 2$: $m \leftarrow 5$, $t \leftarrow 2$
	–	2	5	2	$v[2] \leftarrow m = 5$, $s[2] \leftarrow t = 2$
	–	–	0	0	$m \leftarrow 0$, $t \leftarrow 0$
3	1	–	6	1	$1{+}5 = 6 > 0$: $m \leftarrow 6$, $t \leftarrow 1$
	2	–	6	1	$5{+}1 = 6 = m$: nessun aggiornamento
	3	–	8	3	$8{+}0 = 8 > 6$: $m \leftarrow 8$, $t \leftarrow 3$
	–	3	8	3	$v[3] \leftarrow m = 8$, $s[3] \leftarrow t = 3$
	–	–	0	0	$m \leftarrow 0$, $t \leftarrow 0$
4	1	–	9	1	$1{+}8 = 9 > 0$: $m \leftarrow 9$, $t \leftarrow 1$
	2	–	10	2	$5{+}5 = 10 > 9$: $m \leftarrow 10$, $t \leftarrow 2$
	3	–	10	2	$8{+}1 = 9 < 10$: nessun aggiornamento
	4	–	10	2	$9{+}0 = 9 < 10$: nessun aggiornamento
	–	2	10	2	$v[4] \leftarrow m = 10$, $s[4] \leftarrow t = 2$

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

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s[\ell]$	1	2	3	2

– Calcolato il vettore s, si può calcolare una soluzione ottima dell'istanza dell'RCP applicando ripetutamente la seguente regola, partendo da $j = n$ e da $x_\ell = 0$ per ogni $\ell \in \{1, 2, \ldots, n\}$:

$$x_{s[j]} \leftarrow x_{s[j]} + 1, \qquad j \leftarrow j - s[j],$$

finché $j = 0$. Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, dove ogni componente x_ℓ è il numero di pezzi di lunghezza ℓ in una soluzione ottima.

– Il seguente pseudocodice implementa il calcolo di una soluzione ottima, dato il vettore s.

Algoritmo

Algorithm: DP1_SolComputation

Input: La lunghezza n dell'asta dell'istanza dell'RCP e il vettore $s[1..n]$
Output: Una soluzione ottima dell'istanza dell'RCP

```
1  Function DP1_SolComputation(n, s)
2      x := new array[1..n]
3      for ℓ := 1 to n do
4          x[ℓ] ← 0
5      end
6      j := n
7      while j > 0 do
8          x[s[j]] ← x[s[j]] + 1
9          j ← j − s[j]
10     end
11     return x
```

Esempio 13.7: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 13.1, abbiamo il seguente vettore s:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s[\ell]$	1	2	3	2

Possiamo calcolare una soluzione ottima per un'asta di lunghezza $n = 4$ come segue:

Iterazione	j	$s[j]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$j \leftarrow j - s[j]$
–	–	–	0	0	0	0	$j = 4$
1	4	2	0	1	0	0	$j = 4 - 2 = 2$
2	2	2	0	2	0	0	$j = 2 - 2 = 0$

Una soluzione ottima è il vettore:

$$(0, 2, 0, 0)$$

– La stessa procedura di calcolo, basata sul vettore s, si può applicare partendo da qualunque lunghezza dell'asta ℓ.

Esempio 13.8: Calcolo di soluzioni ottime per le lunghezze dell'asta 3 e 2

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 13.1, abbiamo il seguente vettore s:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s[\ell]$	1	2	3	2

Possiamo calcolare una soluzione ottima per un'asta di lunghezza 3 come segue:

Iterazione	j	$s[j]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$j \leftarrow j - s[j]$
–	–	–	0	0	0	0	$j = 3$
1	3	3	0	0	1	0	$j = 3 - 3 = 0$

Una soluzione ottima è il vettore:

$$(0, 0, 1, 0)$$

Possiamo calcolare una soluzione ottima per un'asta di lunghezza 2 come segue:

Iterazione	j	$s[j]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$j \leftarrow j - s[j]$
–	–	–	0	0	0	0	$j = 2$
1	2	2	0	1	0	0	$j = 2 - 2 = 0$

Una soluzione ottima è il vettore:

$$(0, 1, 0, 0)$$

13.2.1 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da `DP1` nel caso peggiore, dove n è la lunghezza dell'asta. Calcoliamo $S(n)$ contando quante volte vengono eseguite le Righe 2–Riga 13.

- 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. Il numero di esecuzioni delle righe rilevanti è:

  - Riga 2 viene eseguita 1 volta.
  - Riga 3 viene eseguita 1 volta.
  - Riga 4 viene eseguita n volte.
  - Riga 5 viene eseguita n volte.
  - Riga 6 viene eseguita $\sum_{\ell=1}^{n} \ell$ volte.
  - Riga 7 viene eseguita $\sum_{\ell=1}^{n} \ell$ volte.
  - Riga 8 viene eseguita $\sum_{\ell=1}^{n} \ell$ volte.

– Riga 11 viene eseguita n volte.
– Riga 13 viene eseguita 1 volta.

Quindi il numero totale di passi nel caso peggiore è

$$1+1+n+n+\sum_{\ell=1}^{n}\ell+\sum_{\ell=1}^{n}\ell+\sum_{\ell=1}^{n}\ell+n+1=3+3n+3\sum_{\ell=1}^{n}\ell.$$

Usando $\sum_{\ell=1}^{n}\ell=\dfrac{n(n+1)}{2}$, otteniamo

$$S(n)=3+3n+3\cdot\frac{n(n+1)}{2}=\frac{3n^2+9n+6}{2}.$$

Quindi

$$S(n)=\Theta(n^2).$$

Infatti,

$$\lim_{n\to\infty}\frac{\frac{3n^2+9n+6}{2}}{n^2}=\lim_{n\to\infty}\frac{1}{2}\left(3+\frac{9}{n}+\frac{6}{n^2}\right)=\frac{3}{2}\in\mathbb{R}_{>0}.$$

L'algoritmo di PD esegue

$$S(n)=\Theta(n^2)$$

passi nel caso peggiore.

13.3 Un secondo algoritmo di PD

- Passo 1: calcolo del valore ottimo

Per ogni $j\in\{0,1,\ldots,n\}$ e $\ell\in\{0,1,\ldots,n\}$, sia $v(j,\ell)$ il valore ottimo della sottoistanza dell'RCP data da:

1. un'asta di lunghezza ℓ,
2. un insieme $\{1,2,\ldots,j\}$ di j possibili lunghezze dei pezzi,
3. per ogni lunghezza dei pezzi $k\in\{1,2,\ldots,j\}$, un profitto p_k.

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

$$v(j,\ell)=\begin{cases}\max\left\{v(j-1,\ell),\ p_j+v(j,\ell-j)\right\} & \text{se } 1\le j\le\ell,\\ v(j-1,\ell) & \text{se } 1\le\ell<j,\\ 0 & \text{se } \ell=0,\\ -\infty & \text{se } j=0 \text{ e } \ell>0.\end{cases}\tag{13.2}$$

- Il processo di decomposizione ricorsiva della formula (13.2) genera al più 2 **sottoproblemi indipendenti**, corrispondenti alla scelta di tagliare o no un pezzo di lunghezza j da un'asta di lunghezza ℓ. Se $j > \ell$, un pezzo di lunghezza j non si può tagliare dall'asta, e quindi l'unica opzione ammissibile è non tagliarlo.

- Qui $v(j, 0) = 0$, perché un'asta di lunghezza 0 non si può tagliare e ha profitto 0.

- Inoltre, $v(0, \ell) = -\infty$ per $\ell > 0$: in questo caso non è disponibile alcuna lunghezza positiva dei pezzi, quindi non esiste una soluzione ammissibile per un'asta di lunghezza positiva. Il valore $-\infty$ serve a escludere questo caso inammissibile.

- Se $1 \leq j \leq \ell$, il valore ottimo $v(j, \ell)$ si ottiene provando le due opzioni possibili, tagliare o non tagliare un primo pezzo di lunghezza j, e scegliendo quella che massimizza il profitto. I valori corrispondenti sono rispettivamente $p_j + v(j, \ell - j)$ e $v(j-1, \ell)$. Se $1 \leq \ell < j$, la lunghezza j non si può usare, e quindi $v(j, \ell) = v(j-1, \ell)$.

- Quindi una soluzione ottima per un'asta di lunghezza ℓ, con lunghezza massima disponibile dei pezzi j, si costruisce combinando la decisione presa sulla lunghezza j con una soluzione ottima del sottoproblema residuo corrispondente.

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

- Sequenze diverse di decisioni sulle lunghezze dei pezzi, eventualmente prese per valori diversi di j e ℓ, possono portare allo stesso sottoproblema; quindi lo stesso valore $v(j, \ell)$ può dover essere calcolato più volte durante la valutazione ricorsiva della formula (13.2).

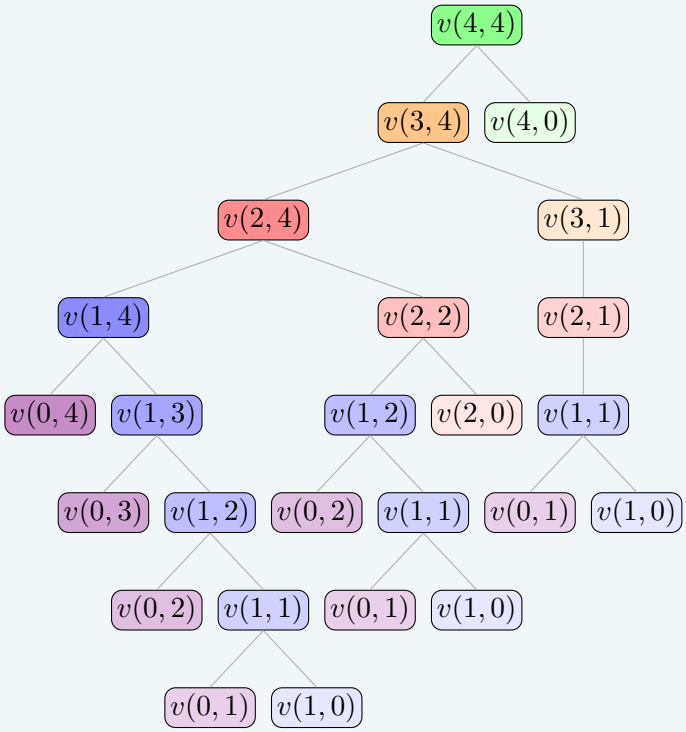
  $\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

- Un **primo metodo** per calcolare il valore ottimo $v(n, n)$ dell'istanza dell'RCP è l'**approccio top-down**. Il metodo parte dalla chiamata con $(j, \ell) = (n, n)$ e valuta ricorsivamente la formula (13.2): ogni chiamata per un'asta di lunghezza ℓ e lunghezza massima disponibile dei pezzi j genera altre chiamate, corrispondenti alle decisioni ammissibili sulla lunghezza j, fino a raggiungere uno dei casi base $v(j, 0)$ o $v(0, \ell)$.

- 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 13.9: Albero di ricorsione

Consideriamo l'istanza dell'RCP dell'Esempio 13.1. Questo è l'albero di ricorsione dell'approccio top-down per calcolare $v(4,4)$:



La funzione v viene chiamata 25 volte, e gli stessi sottoproblemi vengono risolti più volte: $v(1,2)$ viene chiamata 2 volte, $v(0,2)$ viene chiamata 2 volte, $v(1,1)$ viene chiamata 3 volte, $v(0,1)$ viene chiamata 3 volte e $v(1,0)$ viene chiamata 3 volte. Tutte le altre chiamate vengono fatte una sola volta.

– Il valore $v(j,\ell)$ si può calcolare una volta noto $v(j-1,\ell)$ e, quando $j \leq \ell$, una volta noto anche $v(j,\ell-j)$. Quindi, per un valore fissato di j, tutti i valori $v(j,\ell)$ si possono calcolare considerando i valori di ℓ in ordine crescente.

– Partendo dai casi base $v(j,0)=0$ e $v(0,\ell)=-\infty$, i valori della funzione v si possono calcolare riga per riga, passando al valore successivo di j solo dopo aver calcolato tutti i valori $v(j,\ell)$ per il valore corrente di j.

Esempio 13.10: Calcolo dei valori della funzione v

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n=4$; i profitti associati alle possibili lunghezze dei pezzi sono:

	$\ell=1$	$\ell=2$	$\ell=3$	$\ell=4$
p_ℓ	1	5	8	9

I valori della funzione v si calcolano come segue:

∗ Per $j = 1$ si possono usare solo pezzi di lunghezza 1. Quindi:

$$v(1,0) = 0,$$

$$v(1,1) = \max\{v(0,1), p_1 + v(1,0)\} = \max\{-\infty, 1+0\} = 1,$$
$$v(1,2) = \max\{v(0,2), p_1 + v(1,1)\} = \max\{-\infty, 1+1\} = 2,$$
$$v(1,3) = \max\{v(0,3), p_1 + v(1,2)\} = \max\{-\infty, 1+2\} = 3,$$
$$v(1,4) = \max\{v(0,4), p_1 + v(1,3)\} = \max\{-\infty, 1+3\} = 4.$$

∗ Per $j = 2$ si possono usare pezzi di lunghezza 1 e 2. Otteniamo:

$$v(2,0) = 0, \qquad v(2,1) = v(1,1) = 1,$$

$$v(2,2) = \max\{v(1,2), p_2 + v(2,0)\} = \max\{2, 5+0\} = 5,$$
$$v(2,3) = \max\{v(1,3), p_2 + v(2,1)\} = \max\{3, 5+1\} = 6,$$
$$v(2,4) = \max\{v(1,4), p_2 + v(2,2)\} = \max\{4, 5+5\} = 10.$$

∗ Per $j = 3$ si possono usare pezzi di lunghezza 1, 2 e 3. Otteniamo:

$$v(3,0) = 0, \qquad v(3,1) = v(2,1) = 1, \qquad v(3,2) = v(2,2) = 5,$$

$$v(3,3) = \max\{v(2,3), p_3 + v(3,0)\} = \max\{6, 8+0\} = 8,$$
$$v(3,4) = \max\{v(2,4), p_3 + v(3,1)\} = \max\{10, 8+1\} = 10.$$

∗ Per $j = 4$ sono disponibili tutte le lunghezze dei pezzi. Otteniamo:

$$v(4,0) = 0, \qquad v(4,1) = v(3,1) = 1,$$
$$v(4,2) = v(3,2) = 5, \qquad v(4,3) = v(3,3) = 8,$$

$$v(4,4) = \max\{v(3,4), p_4 + v(4,0)\} = \max\{10, 9+0\} = 10.$$

– 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 valore minore di j, o con lo stesso valore di j e una lunghezza dell'asta minore, possiamo risolvere i sottoproblemi in ordine crescente di j e, per ogni valore fissato di j, in ordine crescente di lunghezza dell'asta, partendo dai casi base $v(0,0) = 0$ e $v(0,\ell) = -\infty$ per ogni $\ell \in \{1, \ldots, n\}$ fino ad arrivare a $v(n,n)$.

– Questa osservazione porta in modo naturale a un **metodo molto più efficiente** per calcolare il valore ottimo $v(n,n)$ dell'istanza dell'RCP.

– Un **secondo metodo** per calcolare il valore ottimo $v(n,n)$ dell'istanza dell'RCP è l'**approccio bottom-up**. Il metodo parte dai casi base $v(0,0) = 0$ e $v(0,\ell) =$

$-\infty$ per ogni $\ell \in \{1, \ldots, n\}$ e valuta la formula (13.2) per valori crescenti di $j \in \{1, 2, \ldots, n\}$ e $\ell \in \{0, 1, \ldots, n\}$: ogni valore $v(j, \ell)$ si calcola con valori già calcolati, fino a ottenere il valore $v(n, n)$ dell'istanza dell'RCP.

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

Algoritmo

Algorithm: DP2

Input: Un'istanza dell'RCP data dalla lunghezza n dell'asta e da un array $p[1..n]$ dei profitti dei pezzi

Output: Il valore ottimo dell'istanza dell'RCP

```
1  Function DP2(n, p)
2      v := new array[0..n][0..n]
3      v[0][0] ← 0
4      for ℓ := 1 to n do
5          v[0][ℓ] ← −∞
6      end
7      for j := 1 to n do
8          for ℓ := 0 to n do
9              if j > ℓ then
10                 v[j][ℓ] ← v[j − 1][ℓ]
11             else
12                 if p[j] + v[j][ℓ − j] > v[j − 1][ℓ] then
13                     v[j][ℓ] ← p[j] + v[j][ℓ − j]
14                 else
15                     v[j][ℓ] ← v[j − 1][ℓ]
16                 end
17             end
18         end
19     end
20     return v[n][n]
```

* La matrice $v[0..n][0..n]$ è la **tabella di lookup**: memorizza i valori ottimi di tutti i sottoproblemi.
* L'inizializzazione pone $v[0][0] = 0$ e $v[0][\ell] = -\infty$ per ogni lunghezza dell'asta $\ell \in \{1, \ldots, n\}$, perché una lunghezza positiva non si può riempire se non è disponibile alcuna lunghezza dei pezzi.
* Il ciclo esterno, su $j \in \{1, 2, \ldots, n\}$, considera le lunghezze dei pezzi disponibili in ordine crescente.
* Il ciclo interno, su $\ell \in \{0, 1, \ldots, n\}$, calcola il valore migliore per ogni lunghezza dell'asta usando solo pezzi di lunghezza al più j.

* Se $j > \ell$, la lunghezza j non si può usare, e quindi $v[j][\ell] \leftarrow v[j-1][\ell]$.
* Altrimenti, l'algoritmo di PD confronta il valore ottenuto senza usare la lunghezza j, cioè $v[j-1][\ell]$, con il valore ottenuto usando un pezzo di lunghezza j, cioè $p[j] + v[j][\ell - j]$.
* Infine, l'algoritmo restituisce il valore ottimo dell'istanza dell'RCP, $v[n][n]$.

Esempio 13.11: Esecuzione dell'algoritmo di PD

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n = 4$; i profitti associati alle possibili lunghezze dei pezzi sono:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

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

j	ℓ	$v[j][\ell]$	Azione
0	0	0	$v[0][0] \leftarrow 0$
0	1	$-\infty$	$v[0][1] \leftarrow -\infty$
0	2	$-\infty$	$v[0][2] \leftarrow -\infty$
0	3	$-\infty$	$v[0][3] \leftarrow -\infty$
0	4	$-\infty$	$v[0][4] \leftarrow -\infty$
1	0	0	$v[1][0] \leftarrow v[0][0] = 0$
1	1	1	$\max\{v[0][1], p[1] + v[1][0]\} = \max\{-\infty, 1+0\} = 1$
1	2	2	$\max\{v[0][2], p[1] + v[1][1]\} = \max\{-\infty, 1+1\} = 2$
1	3	3	$\max\{v[0][3], p[1] + v[1][2]\} = \max\{-\infty, 1+2\} = 3$
1	4	4	$\max\{v[0][4], p[1] + v[1][3]\} = \max\{-\infty, 1+3\} = 4$
2	0	0	$v[2][0] \leftarrow v[1][0] = 0$
2	1	1	$2 > 1$: $v[2][1] \leftarrow v[1][1] = 1$
2	2	5	$\max\{v[1][2], p[2] + v[2][0]\} = \max\{2, 5+0\} = 5$
2	3	6	$\max\{v[1][3], p[2] + v[2][1]\} = \max\{3, 5+1\} = 6$
2	4	10	$\max\{v[1][4], p[2] + v[2][2]\} = \max\{4, 5+5\} = 10$
3	0	0	$v[3][0] \leftarrow v[2][0] = 0$
3	1	1	$3 > 1$: $v[3][1] \leftarrow v[2][1] = 1$
3	2	5	$3 > 2$: $v[3][2] \leftarrow v[2][2] = 5$
3	3	8	$\max\{v[2][3], p[3] + v[3][0]\} = \max\{6, 8+0\} = 8$
3	4	10	$\max\{v[2][4], p[3] + v[3][1]\} = \max\{10, 8+1\} = 10$
4	0	0	$v[4][0] \leftarrow v[3][0] = 0$
4	1	1	$4 > 1$: $v[4][1] \leftarrow v[3][1] = 1$
4	2	5	$4 > 2$: $v[4][2] \leftarrow v[3][2] = 5$
4	3	8	$4 > 3$: $v[4][3] \leftarrow v[3][3] = 8$
4	4	10	$\max\{v[3][4], p[4] + v[4][0]\} = \max\{10, 9+0\} = 10$

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

$v[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$j = 1$	0	1	2	3	4
$j = 2$	0	1	5	6	10
$j = 3$	0	1	5	8	10
$j = 4$	0	1	5	8	10

  - Dopo l'esecuzione dell'algoritmo di PD, la matrice v contiene il valore ottimo $v[j][\ell]$ per ogni sottoproblema (j, ℓ).

- Passo 2: calcolo di una soluzione ottima
  - Per calcolare una soluzione ottima è necessario memorizzare alcune informazioni aggiuntive.
  - Per ogni valore j della lunghezza massima disponibile dei pezzi e per ogni lunghezza dell'asta ℓ, oltre al valore ottimo $v[j][\ell]$, memorizziamo $s[j][\ell]$, che vale 1 se si usa un pezzo di lunghezza j e 0 altrimenti.
  - Partendo da $j = n$ e $\ell = n$, se $s[j][\ell] = 0$ la lunghezza j non viene usata e j diminuisce di 1. Se $s[j][\ell] = 1$, si usa un pezzo di lunghezza j e la lunghezza residua dell'asta diventa $\ell - j$. La procedura si ferma quando la lunghezza residua arriva a 0.

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

Algoritmo

Algorithm: DP2_InfoSol

Input: Un'istanza dell'RCP data dalla lunghezza n dell'asta e da un array $p[1..n]$ dei profitti dei pezzi

Output: Il valore ottimo dell'istanza dell'RCP e la matrice di informazioni binarie $s[0..n][0..n]$

```
1  Function DP2_InfoSol(n, p)
2      v := new array[0..n][0..n]
3      s := new array[0..n][0..n]
4      v[0][0] ← 0
5      s[0][0] ← 0
6      for ℓ := 1 to n do
7          v[0][ℓ] ← −∞
8          s[0][ℓ] ← 0
9      end
10     for j := 1 to n do
11         for ℓ := 0 to n do
12             if j > ℓ then
13                 v[j][ℓ] ← v[j − 1][ℓ]
14                 s[j][ℓ] ← 0
15             else
16                 if p[j] + v[j][ℓ − j] > v[j − 1][ℓ] then
17                     v[j][ℓ] ← p[j] + v[j][ℓ − j]
18                     s[j][ℓ] ← 1
19                 else
20                     v[j][ℓ] ← v[j − 1][ℓ]
21                     s[j][ℓ] ← 0
22                 end
23             end
24         end
25     end
26     return v[n][n], s
```

∗ L'algoritmo di PD usa la matrice aggiuntiva s, che tiene traccia della decisione presa quando si calcola $v[j][\ell]$. Se si sceglie un pezzo di lunghezza j, $s[j][\ell] \leftarrow 1$ memorizza questa decisione; altrimenti $s[j][\ell] \leftarrow 0$ registra che la lunghezza j non viene usata.

Esempio 13.12: Calcolo delle informazioni aggiuntive

Consideriamo l'istanza dell'RCP dell'Esempio 13.1, con un'asta di lunghezza $n = 4$; i profitti associati alle possibili lunghezze dei pezzi sono:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

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

j	ℓ	$s[j][\ell]$	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	$s[0][4] \leftarrow 0$
1	0	0	$1 > 0$: $s[1][0] \leftarrow 0$
1	1	1	$p[1] + v[1][0] = 1 + 0 > v[0][1] = -\infty$: $s[1][1] \leftarrow 1$
1	2	1	$p[1] + v[1][1] = 1 + 1 > v[0][2] = -\infty$: $s[1][2] \leftarrow 1$
1	3	1	$p[1] + v[1][2] = 1 + 2 > v[0][3] = -\infty$: $s[1][3] \leftarrow 1$
1	4	1	$p[1] + v[1][3] = 1 + 3 > v[0][4] = -\infty$: $s[1][4] \leftarrow 1$
2	0	0	$2 > 0$: $s[2][0] \leftarrow 0$
2	1	0	$2 > 1$: $s[2][1] \leftarrow 0$
2	2	1	$p[2] + v[2][0] = 5 + 0 > v[1][2] = 2$: $s[2][2] \leftarrow 1$
2	3	1	$p[2] + v[2][1] = 5 + 1 > v[1][3] = 3$: $s[2][3] \leftarrow 1$
2	4	1	$p[2] + v[2][2] = 5 + 5 > v[1][4] = 4$: $s[2][4] \leftarrow 1$
3	0	0	$3 > 0$: $s[3][0] \leftarrow 0$
3	1	0	$3 > 1$: $s[3][1] \leftarrow 0$
3	2	0	$3 > 2$: $s[3][2] \leftarrow 0$
3	3	1	$p[3] + v[3][0] = 8 + 0 > v[2][3] = 6$: $s[3][3] \leftarrow 1$
3	4	0	$p[3] + v[3][1] = 8 + 1 \leq v[2][4] = 10$: $s[3][4] \leftarrow 0$
4	0	0	$4 > 0$: $s[4][0] \leftarrow 0$
4	1	0	$4 > 1$: $s[4][1] \leftarrow 0$
4	2	0	$4 > 2$: $s[4][2] \leftarrow 0$
4	3	0	$4 > 3$: $s[4][3] \leftarrow 0$
4	4	0	$p[4] + v[4][0] = 9 + 0 \leq v[3][4] = 10$: $s[4][4] \leftarrow 0$

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

$s[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	0	0	0	0
$j = 1$	0	1	1	1	1
$j = 2$	0	0	1	1	1
$j = 3$	0	0	0	1	0
$j = 4$	0	0	0	0	0

– Calcolata la matrice s, si può calcolare una soluzione ottima per un'asta di lunghezza n applicando ripetutamente la seguente regola, partendo da $(j, \ell) = (n, n)$ e da $x_h = 0$ per ogni $h \in \{1, 2, \ldots, n\}$:

$$j \leftarrow j - 1 \qquad \text{se } s[j][\ell] = 0,$$
$$x_j \leftarrow x_j + 1, \quad \ell \leftarrow \ell - j \qquad \text{se } s[j][\ell] = 1.$$

La procedura si ferma quando $\ell = 0$. Si ottiene così il vettore $(x_1, x_2, \ldots, x_n)$, dove ogni componente x_h è il numero di pezzi di lunghezza h in una soluzione ottima.

– Il seguente pseudocodice implementa il calcolo di una soluzione ottima, data la matrice s.

Algoritmo

Algorithm: DP2_SolComputation

Input: La lunghezza n dell'asta e la matrice $s[0..n][0..n]$
Output: Una soluzione ottima dell'istanza dell'RCP

```
1  Function DP2_SolComputation(n, s)
2      x := new array[1..n]
3      for h := 1 to n do
4          x[h] ← 0
5      end
6      j := n
7      ℓ := n
8      while ℓ > 0 do
9          if s[j][ℓ] = 0 then
10             j ← j − 1
11         else
12             x[j] ← x[j] + 1
13             ℓ ← ℓ − j
14         end
15     end
16     return x
```

Esempio 13.13: Calcolo di una soluzione ottima

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 13.1, abbiamo la seguente matrice s:

$s[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	0	0	0	0
$j = 1$	0	1	1	1	1
$j = 2$	0	0	1	1	1
$j = 3$	0	0	0	1	0
$j = 4$	0	0	0	0	0

Possiamo calcolare una soluzione ottima per un'asta di lunghezza $n = 4$ come segue:

Iterazione	j	ℓ	$s[j][\ell]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Azione
1	4	4	0	0	0	0	0	$j \leftarrow 3$
2	3	4	0	0	0	0	0	$j \leftarrow 2$
3	2	4	1	0	1	0	0	$x[2] \leftarrow 1$, $\ell \leftarrow 4-2=2$
4	2	2	1	0	2	0	0	$x[2] \leftarrow 2$, $\ell \leftarrow 2-2=0$

Una soluzione ottima è il vettore:

$$(0, 2, 0, 0)$$

– La stessa procedura di calcolo, basata sulla matrice s, si può applicare a qualunque lunghezza dell'asta $\ell \leq n$ partendo da $j = n$.

Esempio 13.14: Calcolo di soluzioni ottime

Dopo l'esecuzione dell'algoritmo di PD sull'istanza dell'Esempio 13.1, abbiamo la seguente matrice s:

$s[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	0	0	0	0
$j = 1$	0	1	1	1	1
$j = 2$	0	0	1	1	1
$j = 3$	0	0	0	1	0
$j = 4$	0	0	0	0	0

Possiamo calcolare una soluzione ottima per un'asta di lunghezza $\ell = 3$ come segue:

Iterazione	j	ℓ	$s[j][\ell]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Azione
1	4	3	0	0	0	0	0	$j \leftarrow 3$
2	3	3	1	0	0	1	0	$x[3] \leftarrow 1$, $\ell \leftarrow 3-3=0$

Una soluzione ottima è il vettore:

$$(0, 0, 1, 0)$$

Possiamo calcolare una soluzione ottima per un'asta di lunghezza $\ell = 2$ come segue:

Iterazione	j	ℓ	$s[j][\ell]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Azione
1	4	2	0	0	0	0	0	$j \leftarrow 3$
2	3	2	0	0	0	0	0	$j \leftarrow 2$
3	2	2	1	0	1	0	0	$x[2] \leftarrow 1$, $\ell \leftarrow 2-2=0$

Una soluzione ottima è il vettore:

$$(0, 1, 0, 0)$$

13.3.1 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da `DP2` nel caso peggiore, dove n è la lunghezza dell'asta. Calcoliamo $S(n)$ contando quante volte vengono eseguite le righe dello pseudocodice di `DP2`.

- Il numero di volte in cui viene eseguita ogni riga di codice è:
  - Riga 2 viene eseguita 1 volta.
  - Riga 3 viene eseguita 1 volta.
  - Riga 4 viene eseguita n volte.
  - Riga 5 viene eseguita n volte.
  - Riga 7 viene eseguita n volte.
  - Riga 8 viene eseguita $n(n+1)$ volte.
  - Riga 9 viene eseguita $n(n+1)$ volte.
  - Riga 12 viene controllata solo quando $j \leq \ell$, cioè
  $$\sum_{j=1}^{n}(n-j+1) = \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$
  volte.
  - Per ogni coppia (j, ℓ) viene eseguita una fra la Riga 10, la Riga 13 e la Riga 15, e quindi queste righe di assegnamento vengono eseguite $n(n+1)$ volte in totale.
  - Riga 20 viene eseguita 1 volta.

  Quindi il numero totale di passi nel caso peggiore è
  $$1+1+n+n+n+n(n+1)+n(n+1)+\frac{n(n+1)}{2}+n(n+1)+1.$$
  Quindi
  $$S(n) = 3+3n+3n(n+1)+\frac{n(n+1)}{2} = \frac{7n^2+13n+6}{2}.$$
  Quindi
  $$S(n) = \Theta(n^2).$$
  Infatti,
  $$\lim_{n\to\infty} \frac{\frac{7n^2+13n+6}{2}}{n^2} = \lim_{n\to\infty} \frac{1}{2}\left(7+\frac{13}{n}+\frac{6}{n^2}\right) = \frac{7}{2} \in \mathbb{R}_{>0}.$$

L'algoritmo di PD esegue
$$S(n) = \Theta(n^2)$$
passi nel caso peggiore.

13.4 Errori tipici

Attenzione

- **Contare i vettori invece degli schemi di taglio.** I 2^{n-1} schemi di taglio di un'asta di lunghezza n sono più dei vettori soluzione quando $n \geq 3$, perché schemi diversi possono dare lo stesso vettore.
- **Dimenticare il caso base.** $v(0) = 0$ si usa quando il primo pezzo ha la lunghezza dell'intera asta: $v(\ell)$ contiene il termine $p_\ell + v(0)$.
- **Usare la riga $j-1$ nella seconda ricorrenza.** Il termine $p_j + v(j, \ell - j)$ resta nella riga j: altrimenti ogni lunghezza dei pezzi si potrebbe tagliare una sola volta.
- **Fermare la ricostruzione troppo presto.** La regola $x_{s[\ell]} \leftarrow x_{s[\ell]} + 1$, $\ell \leftarrow \ell - s[\ell]$ si ripete finché $\ell = 0$; l'ultimo pezzo completa l'asta.

13.5 In sintesi

In sintesi

- L'RCP chiede come tagliare un'asta di lunghezza n in pezzi, con ogni lunghezza disponibile un numero qualsiasi di volte, per massimizzare il profitto totale.
- La ricorrenza $v(\ell) = \max_{j \leq \ell}\{p_j + v(\ell - j)\}$ ha sottostruttura ottima e sottoproblemi sovrapposti, e $v(0) = 0$.
- Il vettore v si riempie bottom-up in $\Theta(n^2)$ passi, un numero polinomiale perché l'input contiene n profitti.
- Il vettore s dei primi pezzi dà una soluzione ottima per ogni lunghezza dell'asta senza costi aggiuntivi.

Quadro riassuntivo dell'esempio dell'RCP. Esempio 13.1 (asta di lunghezza 4):

Metodo	Lower bound	Upper bound	Origine
Algoritmo di PD	10	10	valore ottimo, due pezzi di lunghezza 2