PD per il problema dello zaino illimitato

Indice

Istanze e soluzioni ammissibili
    Codifica delle soluzioni ammissibili

Un primo algoritmo di PD
    Numero di passi del primo algoritmo di PD

Un secondo algoritmo di PD
    Numero di passi del secondo algoritmo di PD

Errori tipici

In sintesi

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.

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 e la dimensione dell'input.

Il problema di questo capitolo

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

Istanze e soluzioni ammissibili

Introduzione: istanze e soluzioni ammissibili

- Introduciamo l'istanza dell'UKP che useremo in tutto il capitolo
- Limitiamo il numero di soluzioni ammissibili di un'istanza dell'UKP e vediamo perché enumerarle diventa presto impraticabile
- Descriviamo come si codifica una soluzione ammissibile e la differenza principale rispetto al KP
- È il punto di partenza degli algoritmi di PD delle prossime sezioni

Esempio: 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 istanza di riferimento.

Quante soluzioni ammissibili?

- 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.

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 rispetto al KP

- 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.

Riepilogo

Istanze e soluzioni ammissibili:

- Un'istanza dell'UKP: una capacità c e n tipi di oggetto, ciascuno con un profitto p_j e un peso w_j, disponibili in un numero qualsiasi di copie
- Una soluzione ammissibile usa al più $\lfloor c/w_j \rfloor$ copie del tipo di oggetto j: al più $\prod_{j=1}^{n} \left(\lfloor c/w_j \rfloor + 1 \right)$ soluzioni ammissibili, troppe da enumerare

Codifica:

- Un vettore intero $(x_1, x_2, \ldots, x_n)$ con $\sum_{j=1}^{n} w_j x_j \leq c$ e profitto $\sum_{j=1}^{n} p_j x_j$
- A differenza del KP, le variabili sono intere non negative, non binarie

Un primo algoritmo di PD

Introduzione: un primo algoritmo di PD

- Nell'UKP un tipo di oggetto si può usare un numero qualsiasi di volte, quindi il valore ottimo dipende solo dalla capacità residua
- Scriviamo una ricorrenza $v(k)$ sulle sole capacità e verifichiamo la sottostruttura ottima e i sottoproblemi sovrapposti
- Calcoliamo il valore ottimo top-down e bottom-up, e poi ricostruiamo una soluzione ottima
- Infine contiamo i passi dell'algoritmo di PD

Passo 1: calcolo del valore ottimo

Per ogni $k \in \{0, 1, \ldots, c\}$, sia $v(k)$ il valore ottimo della sottoistanza 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 ricorrenza per $v(k)$

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

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

Sottoproblemi indipendenti e caso base

- Il processo di decomposizione ricorsiva della formula ricorsiva per $v(k)$ genera al più n **sottoproblemi indipendenti**, ciascuno corrispondente a una possibile scelta di un tipo di oggetto j, fra quelli con $w_j \leq k$, 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 $v(k) = 0$.

Sottostruttura ottima

- Il valore ottimo $v(k)$ si ottiene provando ogni tipo di oggetto j con $w_j \leq k$ 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!**

Sottoproblemi sovrapposti

- 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 ricorsiva per $v(k)$.

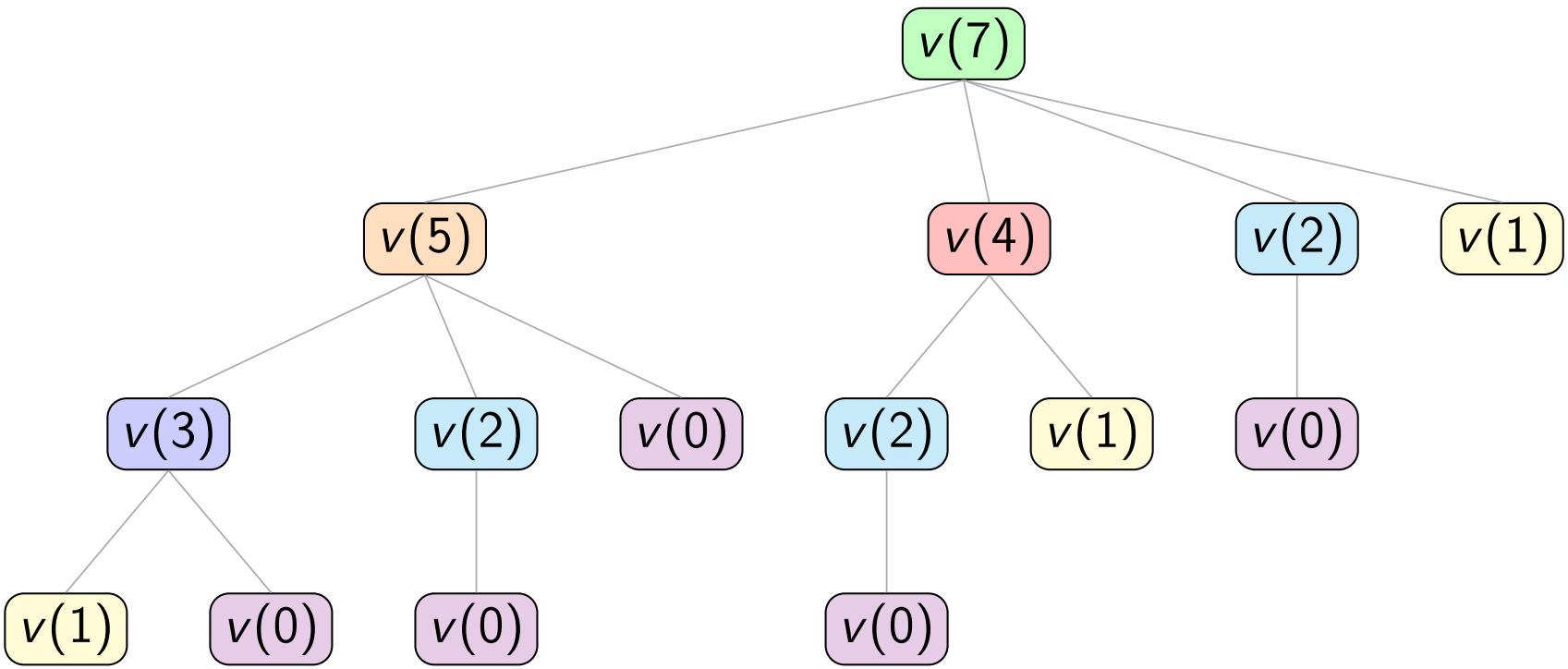
$\rightsquigarrow$ **Sottoproblemi sovrapposti verificati!**

L'approccio top-down

- 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 ricorsiva per $v(k)$: 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: albero di ricorsione (1/2)

Consideriamo l'istanza di riferimento, 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 $v(7)$:



Esempio: albero di ricorsione (2/2)

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,
- $v(0)$ viene chiamata 5 volte.

Tutte le altre chiamate avvengono una sola volta.

Il calcolo dei valori per capacità crescenti

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

Consideriamo l'istanza di riferimento, 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:

$$v(1) = 0.$$

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

- 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:

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

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

- 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:

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

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

- 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:

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

La lookup table

- 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.

L'approccio bottom-up

- 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 ricorsiva per $v(k)$ 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.

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

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

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
```

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

```
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]
16 end
```

Come funziona l'algoritmo di PD

- 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: esecuzione dell'algoritmo di PD (1/6)

Consideriamo l'istanza di riferimento, 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.

Esempio: esecuzione dell'algoritmo di PD (2/6)

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	$v[1] \leftarrow m = 0$

Esempio: esecuzione dell'algoritmo di PD (3/6)

k	j	$v[k]$	m	Azione
	–	–	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	$v[3] \leftarrow m = 5$

Esempio: esecuzione dell'algoritmo di PD (4/6)

k	j	$v[k]$	m	Azione
	–	–	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	$v[5] \leftarrow m = 10$

Esempio: esecuzione dell'algoritmo di PD (5/6)

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	$v[7] \leftarrow m = 14$

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

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.

Le informazioni per una soluzione ottima (1/2)

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

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
```

Le informazioni per una soluzione ottima (2/2)

```
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
16  end
```

Come si memorizzano le informazioni

- 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: calcolo delle informazioni aggiuntive (1/6)

Consideriamo l'istanza di riferimento, 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.

Esempio: calcolo delle informazioni aggiuntive (2/6)

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	$v[1] \leftarrow m = 0$, $s[1] \leftarrow t = 0$

Esempio: calcolo delle informazioni aggiuntive (3/6)

k	j	$s[k]$	m	t	Azione
	–	–	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	$v[3] \leftarrow m = 5,\ s[3] \leftarrow t = 2$

Esempio: calcolo delle informazioni aggiuntive (4/6)

k	j	$s[k]$	m	t	Azione
	–	–	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	$v[5] \leftarrow m = 10$, $s[5] \leftarrow t = 3$

Esempio: calcolo delle informazioni aggiuntive (5/6)

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	$v[7] \leftarrow m = 14,\ s[7] \leftarrow t = 1$

Esempio: calcolo delle informazioni aggiuntive (6/6)

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

Il calcolo di una soluzione ottima dal vettore s

- 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.

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
12 end
```

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

Dopo l'esecuzione dell'algoritmo di PD sull'istanza di riferimento, 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	$k = 5 - 5 = 0$

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

Una soluzione ottima è il vettore:

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

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

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.

Richiamiamo lo pseudocodice di UKP_DP1 (1/2)

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
```

Richiamiamo lo pseudocodice di UKP_DP1 (2/2)

```
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]
16  end
```

Numero di esecuzioni delle righe

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.

Il numero totale di passi

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

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

Il numero di passi del primo algoritmo di PD

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 $\Theta(nc)$.

Riepilogo (1/2)

La ricorrenza sulle capacità:

- $v(k)$ è il valore ottimo per uno zaino di capacità k con tutti gli n tipi di oggetto
- $v(0) = 0$; $v(k) = \max_{j:\, w_j \leq k}\{p_j + v(k - w_j)\}$, oppure 0 se nessun tipo di oggetto entra
- Al più n sottoproblemi indipendenti; sottostruttura ottima e sottoproblemi sovrapposti verificati

Top-down e bottom-up:

- L'albero di ricorsione per $v(7)$ ha 15 chiamate, con sottoproblemi ripetuti
- `UKP_DP1` riempie la lookup table $v[0..c]$ per k crescente; sull'istanza di riferimento $v[7] = 14$

Riepilogo (2/2)

Una soluzione ottima:

- $s[k]$ è il tipo di oggetto scelto per primo per la capacità k, oppure 0 se nessun tipo di oggetto entra
- Da $k = c$: una copia in più del tipo di oggetto $s[k]$, poi $k \leftarrow k - w_{s[k]}$, finché $k = 0$ o $s[k] = 0$
- Sull'istanza di riferimento: $(1, 0, 1, 0)$, con profitto 14

Numero di passi:

- $S(n, c) = 3 + 3c + 4cn = \Theta(nc)$, anche con il vettore s

Un secondo algoritmo di PD

Introduzione: un secondo algoritmo di PD

- Scriviamo una seconda ricorrenza, $v(j, k)$, sulle coppie (tipo di oggetto, capacità), come per il KP
- La differenza rispetto al KP: dopo aver inserito una copia del tipo di oggetto j, la ricorrenza resta nella stessa riga j, perché sono ancora disponibili altre copie
- Seguiamo lo stesso percorso: top-down, bottom-up, ricostruzione di una soluzione ottima
- Infine contiamo i passi e li confrontiamo con quelli del 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 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 ricorrenza per $v(j, k)$

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

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

Sottoproblemi indipendenti e caso base

- Il processo di decomposizione ricorsiva della formula ricorsiva per $v(j, k)$ 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.

Le due opzioni

- 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.

Sottostruttura ottima e sottoproblemi sovrapposti

- 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.

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

- 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 ricorsiva per $v(j, k)$.

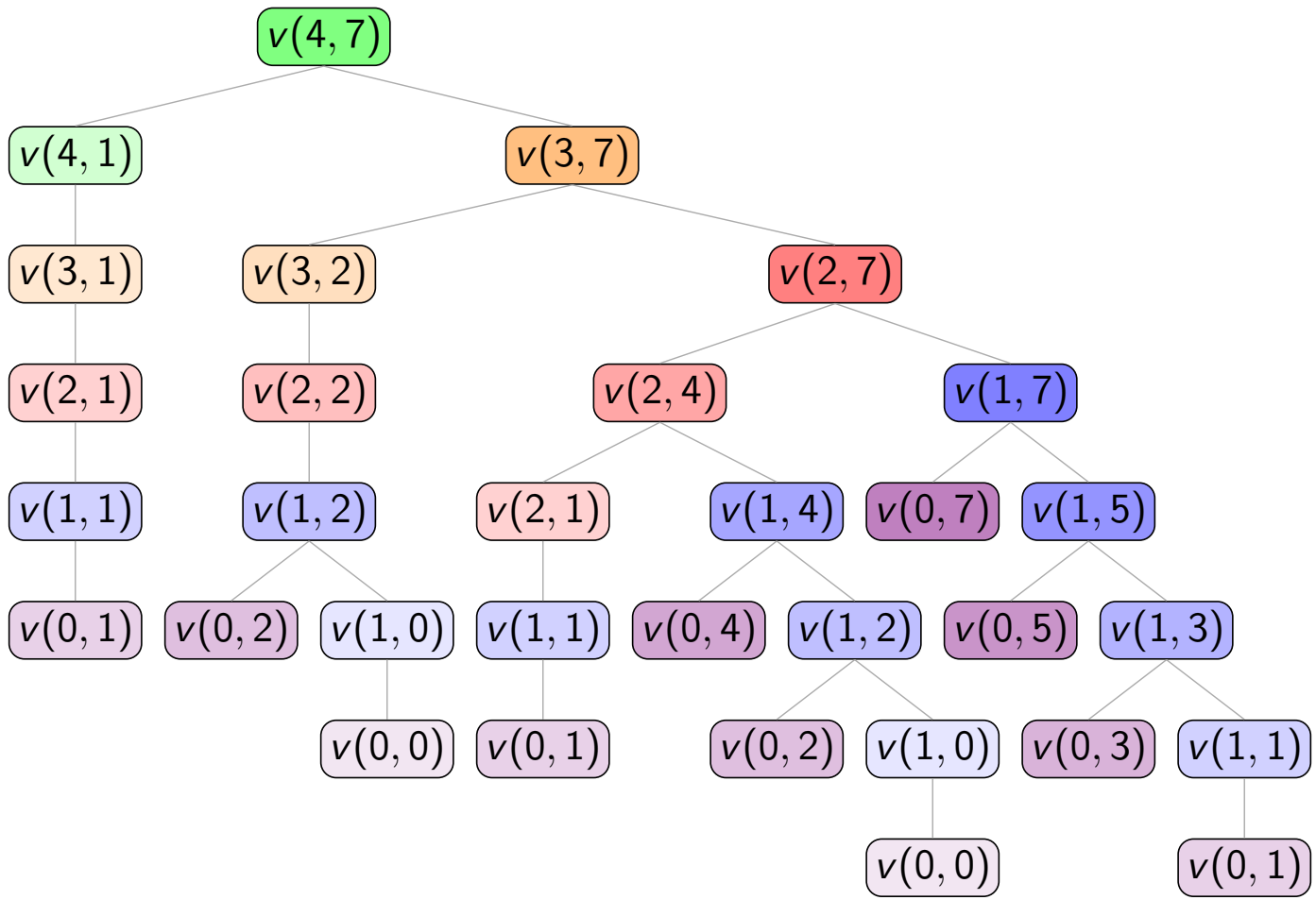
$$\rightsquigarrow \textbf{Sottoproblemi sovrapposti verificati!}$$

L'approccio top-down

- 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 ricorsiva per $v(j, k)$: 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: albero di ricorsione (1/2)

Consideriamo l'istanza di riferimento, 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 $v(4, 7)$:



Esempio: albero di ricorsione (2/2)

La funzione v viene chiamata 32 volte e gli stessi sottoproblemi vengono risolti più volte:

- $v(1,1)$ viene chiamata 3 volte,
- $v(0,1)$ viene chiamata 3 volte,
- $v(1,2)$ viene chiamata 2 volte,
- $v(0,2)$ viene chiamata 2 volte,
- $v(1,0)$ viene chiamata 2 volte,
- $v(0,0)$ viene chiamata 2 volte,
- $v(2,1)$ viene chiamata 2 volte.

Tutte le altre chiamate avvengono una sola volta.

Il calcolo dei valori riga per riga

- 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: calcolo dei valori della funzione v (1/5)

Consideriamo l'istanza di riferimento, 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, un valore di j alla volta, partendo dal caso base $v(0, k) = 0$ per ogni k.

Esempio: calcolo dei valori della funzione v (2/5)

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

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

Esempio: calcolo dei valori della funzione v (3/5)

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

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

Esempio: calcolo dei valori della funzione v (4/5)

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

$$v(3,0) = 0, \qquad v(3,1) = v(2,1) = 0, \qquad v(3,2) = v(2,2) = 4,$$
$$v(3,3) = v(2,3) = 5, \qquad v(3,4) = v(2,4) = 8,$$
$$v(3,5) = \max\{v(2,5), p_3 + v(3,0)\} = \max\{9, 10+0\} = 10,$$
$$v(3,6) = \max\{v(2,6), p_3 + v(3,1)\} = \max\{12, 10+0\} = 12,$$
$$v(3,7) = \max\{v(2,7), p_3 + v(3,2)\} = \max\{13, 10+4\} = 14.$$

Esempio: calcolo dei valori della funzione v (5/5)

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

$$v(4,0) = 0, \quad v(4,1) = v(3,1) = 0, \quad v(4,2) = v(3,2) = 4,$$
$$v(4,3) = v(3,3) = 5, \quad v(4,4) = v(3,4) = 8,$$
$$v(4,5) = v(3,5) = 10,$$
$$v(4,6) = \max\{v(3,6), p_4 + v(4,0)\} = \max\{12, 11 + 0\} = 12,$$
$$v(4,7) = \max\{v(3,7), p_4 + v(4,1)\} = \max\{14, 11 + 0\} = 14.$$

La lookup table

- 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.

L'approccio bottom-up

- 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 ricorsiva per $v(j, k)$ 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.

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

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

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
```

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

```
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]
20  end
```

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

- La matrice $v[0..n][0..c]$ è la **lookup table**: memorizza i valori ottimi di tutti i sottoproblemi.
- 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 $1, 2, \ldots, j$.

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

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

Consideriamo l'istanza di riferimento, 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.

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

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	$v[0][7] \leftarrow 0$

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

j	k	$v[j][k]$	Azione
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	$\max\{v[0][7], p[1] + v[1][5]\} = \max\{0, 4 + 8\} = 12$

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

j	k	$v[j][k]$	Azione
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	$\max\{v[1][7], p[2] + v[2][4]\} = \max\{12, 5 + 8\} = 13$

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

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	$\max\{v[2][7], p[3] + v[3][2]\} = \max\{13, 10 + 4\} = 14$

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

j	k	$v[j][k]$	Azione
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	$\max\{v[3][7], p[4] + v[4][1]\} = \max\{14, 11 + 0\} = 14$

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

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.

Le informazioni per una soluzione ottima (1/2)

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

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
```

Le informazioni per una soluzione ottima (2/2)

```
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
20  end
```

Come si memorizzano le informazioni

- 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: calcolo delle informazioni aggiuntive (1/7)

Consideriamo l'istanza di riferimento, 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.

Esempio: calcolo delle informazioni aggiuntive (2/7)

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	$s[0][7] \leftarrow 0$

Esempio: calcolo delle informazioni aggiuntive (3/7)

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

Esempio: calcolo delle informazioni aggiuntive (4/7)

j	k	$s[j][k]$	Azione
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	$p[2] + v[2][4] = 5 + 8 > v[1][7] = 12$: $s[2][7] \leftarrow 1$

Esempio: calcolo delle informazioni aggiuntive (5/7)

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	$p[3] + v[3][2] = 10 + 4 > v[2][7] = 13$: $s[3][7] \leftarrow 1$

Esempio: calcolo delle informazioni aggiuntive (6/7)

j	k	$s[j][k]$	Azione
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	$p[4] + v[4][1] = 11 + 0 \leq v[3][7] = 14$: $s[4][7] \leftarrow 0$

Esempio: calcolo delle informazioni aggiuntive (7/7)

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

Il calcolo di una soluzione ottima dalla matrice s

- 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.

Il calcolo di una soluzione ottima: lo pseudocodice (1/2)

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
```

Il calcolo di una soluzione ottima: lo pseudocodice (2/2)

```
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
17  end
```

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

Dopo l'esecuzione dell'algoritmo di PD sull'istanza di riferimento, 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

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

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 $v(4, 7) = 14$.

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.

Richiamiamo lo pseudocodice di UKP_DP2 (1/2)

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
```

Richiamiamo lo pseudocodice di UKP_DP2 (2/2)

```
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]
20  end
```

Numero di esecuzioni delle righe

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.

Il numero totale di passi

Pertanto, il numero totale di passi nel caso peggiore è

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

Quindi

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

Dunque

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

Infatti

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

Il numero di passi del secondo algoritmo di PD

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 $\Theta(nc)$.

Riepilogo (1/2)

La ricorrenza sulle coppie (tipo di oggetto, capacità):

- $v(j, k)$ è il valore ottimo per uno zaino di capacità k con i tipi di oggetto $1, 2, \ldots, j$
- $v(0, k) = 0$; $v(j, k) = v(j-1, k)$ se $k < w_j$; altrimenti $\max\{v(j-1, k),\ p_j + v(j, k - w_j)\}$
- Il termine $p_j + v(j, k - w_j)$ resta nella stessa riga j: sono ancora disponibili altre copie del tipo di oggetto j

Top-down e bottom-up:

- L'albero di ricorsione per $v(4, 7)$ ha 32 chiamate, con sottoproblemi ripetuti
- `UKP_DP2` riempie la matrice $v[0..n][0..c]$ riga per riga; sull'istanza di riferimento $v[4][7] = 14$

Riepilogo (2/2)

Una soluzione ottima:

- $s[j][k] = 1$ se si usa una copia del tipo di oggetto j nel calcolo di $v[j][k]$, e 0 altrimenti
- Da $(j, k) = (n, c)$: se $s[j][k] = 0$ allora $j \leftarrow j - 1$, altrimenti una copia in più del tipo di oggetto j e $k \leftarrow k - w_j$, finché $k = 0$ o $j = 0$
- Sull'istanza di riferimento: $(1, 0, 1, 0)$, con profitto 14, come con il primo algoritmo di PD

Numero di passi:

- $S(n, c) = 4nc + 4n + 2c + 4 = \Theta(nc)$, anche con la matrice s: lo stesso ordine del primo algoritmo di PD

Errori tipici

Introduzione: errori tipici

- Elenchiamo gli errori più frequenti con gli algoritmi di PD per l'UKP
- Riguardano la seconda ricorrenza, la capacità inutilizzata, la ricostruzione di una soluzione e il caso base
- Conoscerli aiuta a eseguire correttamente a mano gli algoritmi di PD

Osservazione 1: Errori tipici (1/2)

- ***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'istanza di riferimento, $v(1) = 0$.*

Osservazione: Errori tipici (2/2)

- ***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*.*

Riepilogo

Errori tipici:

- Nella seconda ricorrenza le copie successive del tipo di oggetto j vengono dalla stessa riga j, non dalla riga $j-1$
- La capacità non deve per forza essere riempita: se nessun tipo di oggetto entra, il valore è 0
- La ricostruzione si ferma alla capacità residua 0, oppure a $s[k]=0$ (primo algoritmo di PD) o a $j=0$ (secondo)
- Il caso base $v(0)=0$: il vettore $v[0..c]$ ha $c+1$ celle

In sintesi

Introduzione: in sintesi

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

Riepilogo (1/2)

- Nell'UKP un tipo di oggetto si può usare un numero qualsiasi di volte, quindi il valore ottimo dipende solo dalla capacità residua: $v(k) = \max_{j:\, w_j \leq k}\{p_j + v(k - w_j)\}$.
- 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.

Riepilogo (2/2)

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

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