Esercizi programmazione dinamica

A.Marchetti Spaccamela

Calcola quanti percorsi

Supponete di avere una scacchiera con n × n caselle e una pedina. Quando posizionata sulla generica casella (i,j) per la pedina sono possibili al più due mosse:

- spostarsi verso il basso nella casella (i+1, j), se i < n-1;
- spostarsi verso destra nella casella (i, j+1), se j < n-1.

Progettare un algoritmo di programmazione dinamica che calcola il numero di percorsi possibili per spostare la pedina dalla casella (0, 0) in alto a sinistra alla casella (n-1, n-1) in basso a destra

ad esempio, in una scacchiera 3 × 3 i percorsi possibili sono 6. Valutare la complessità dell'algoritmo proposto.

Calcola quanti percorsi: soluzione

ContaPercorsi(n)

Osservazioni

- esiste un solo modo per raggiungere gli elementi della prima riga e colonna
- Posso raggiungere la posizione (i,j) solo dalle posizioni (i-1,j) o (i, j-1)

Calcola quanti percorsi: soluzione

```
ContaPercorsi(n) L = nuova matrice n x n;
// soluzione problemi elementari e memorizzazione nella matrice
// esiste un solo modo per raggiungere gli elementi della prima riga e colonna
for (i = 0; i < n; i++) L[i,0] = 1; for (j = 0; j < n; j++) L[0,j] = 1;
// riempimento della matrice
for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
        L[i,j] = L[i-1,j] + L[i,j-1];
return L[n-1,n-1];
```

Complessità: $T(n) = \Theta(n^2)$

Percorso pesato

Supponete di avere una scacchiera con n × n caselle e una pedina. Quando posizionata sulla generica casella (i,j) per la pedina sono possibili al più due mosse:

- spostarsi verso il basso nella casella (i+1, j), se i < n-1;
- spostarsi verso destra nella casella (i, j+1), se j < n-1.

Si assuma che ad ogni casella (i,j) sia associato un valore $C_{i,j,}$ e si progetti un algoritmo che trovi il percorso da (0,0) a (n-1,n-1) di valore totale (ovvero la somma del valori delle caselle calpestate) massimo. Suggerimento: si scriva prima una funzione che trova il valore massimo, e poi una che ne ricostruisca il percorso.

Percorso pesato: soluzione

Sottoproblemi: Si memorizza in T[i,j] il cammino di valore massimo dalla casella (0,0) alla casella (i,j).

Sottoproblemi elementari

- T[0,0] = c[0,0]
- prima riga ci si può muovere solo in orizzontale:
  T[0,j] = T[0,j-1] + c[0,j] j > 0
- prima colonna ci si può muovere solo in verticale:
  T[i,0] = T[i-1,0] + c[i,0] i > 0

Passo generale ??

Percorso pesato: soluzione

Sottoproblemi: Si memorizza in T[i,j] il cammino di valore massimo dalla casella (0,0) alla casella (i,j).

Sottoproblemi elementari

- T[0,0] = c[0,0]
- prima riga ci si può muovere solo in orizzontale:
  T[0,j] = T[0,j-1] + c[0,j] j > 0
- prima colonna ci si può muovere solo in verticale:
  T[i,0] = T[i-1,0] + c[i,0] i > 0

Passo generale

- La casella (i,j) si può raggiungere solo spostandosi da (i-1,j) o da (i, j-1)
  T[i,j] = c[i,j] + max {T[i-1,j], T[i,j-1]}

Percorso pesato: soluzione calcolo valore

```
//input C: matrice dei valori delle caselle
// T = nuova matrice n x n;
T[0,0] = c[0,0]
for (i = 1; i < n; i++) T[i,0] = T[i-1,0] + c[i,0]
for (j = 1; j < n; j++) T[0,j] = T[0,j-1] + c[0,j]
for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
        if (T[i-1,j] ≥ T[i,j-1])
        then T[i,j] = c[i,j] + T[i-1,j];
        else T[i,j] = c[i,j] + T[i,j-1];
```

Percorso pesato: ricostruzione del cammino

// il cammino da (0,0) a (n-1, n-1) consta di 2n-2 mosse

cammino = array di dimensione 2n–2

// calcolo il cammino a partire dall'ultima posizione (n-1, n-1)

i = n-1;
j = n-1;
for k = 2n-3; k ≥ 0; k--) //ogni iterazione decremento k e mi avvicino a (0,0)

if (T[i,j] == c[i,j] + T[i-1,j])

then {cammino[k] = ; i= i-1 }

else { cammino[k] = ; j= j-1 }

print(cammino)

Convenienza di acquisti/vendite

Dato un vettore P di n interi in cui P[i] rappresenta il prezzo di una certa merce nei prossimi i giorni, i=1,2,…n, vogliamosapere qual è il giorno i in cui conviene comprare quella merce ed il giorno j, con j > i, in cui conviene rivenderla in modo da massimizzare il profitto o in alternativa (se non è possibile guadagnarci) minimizzare la perdita.

In altre parole siamo interessati a conoscere la coppia (i, j) con i < j per cui risulta massimo il valore P [j] – P [i].

Descrivere un algoritmo che risolve il problema in O(n).

Nota Bene: ovviamente pensare di sapere il prezzo è irrealistico; ma i programmi di analisi finanziaria utilizzano una stima dei prezzi

Convenienza di acquisti/vendite

Dato un vettore P di n interi in cui P[i] rappresenta il prezzo di una certa merce nei prossimi i giorni, i=1,2,…n, vogliamo sapere qual è il giorno i in cui conviene comprare quella merce ed il giorno j, con j > i, in cui conviene rivenderla in modo da massimizzare il profitto o in alternativa (se non è possibile guadagnarci) minimizzare la perdita.

- E' facile trovare un algoritmo con costo $O(n^2)$: sia G[j] il miglior prezzo che ottengo quando vendo il giorno j
- Per vendere il giorno j devo acquistare uno dei giorni che precedono j
- Quindi G[j]= P[j] – (minimo prezzo di un giorno i che precede j)

  = P[j] – min(P[i], i < j)
- Quindi un programma con costo $O(n^2)$ - due cicli for annidati - è immediato

Convenienza di acquisti/vendite

Dato un vettore P di n interi in cui P[i] rappresenta il prezzo di una certa merce nei prossimi i giorni, i+1,2,…n, vogliamosapere qual è il giorno i in cui conviene comprare quella merce ed il giorno j, con j > i, in cui conviene rivenderla in modo da massimizzare il profitto o in alternativa (se non è possibile guadagnarci) minimizzare la perdita.

Programmazione dinamica: definire sottoproblemi

Sottoproblemi:

- Sia A[j] il costo di acquisto minimo per poter vendere il giorno j

  Quindi sappiamo che A[j] = min(P[i], i < j) = min(A[j-1], P[j])

- Per vendere il giorno j devo acquistare in uno dei gironi che precedono j

Convenienza di acquisti/vendite

Algoritmo lineare: Supponiamo di voler calcolare solo il massimo guadagno G (non i giorni di acquisto vendita)

- Sia A[j] il costo di acquisto minimo per poter vendere il giorno j
- Quindi il programma potrebbe essere

  G= P[2]- P[1] // inizializzo: se vendo il giorno 2 compro il giorno 1

  for j=3 to n

  // calcolo max guadagno quando vendo un giorno in {3,…j} calcola A[j]

  G= max (G, P[j] – A[j])

- Per avere un algoritmo di costo lineare devo calcolare A[j] in tempo costante. COME?

Convenienza di acquisti/vendite

G= P[2]- P[1] // inizializzo: se vendo il giorno 2 compro il giorno 1

for j=3 to n

// calcolo max guadagno quando vendo un giorno in {3,…j} calcola A[j]

G= max (G, P[j] – A[j])

Come calcolare A[j] in tempo costante?

- Per vendere il giorno j devo acquistare in un giorno i, precedente j i <j
- Quindi $A[j] = \min_{i<j} P[i] = \min_{i \le j-1} P[i]$
- Posso riscrivere quindi $A[j] = \min_{i<j} P_i = \min (A[j-2], P[j-1])$

Convenienza di acquisti/vendite

Algoritmo lineare: Supponiamo di voler calcolare solo il massimo guadagno G (non i giorni di acquisto vendita)

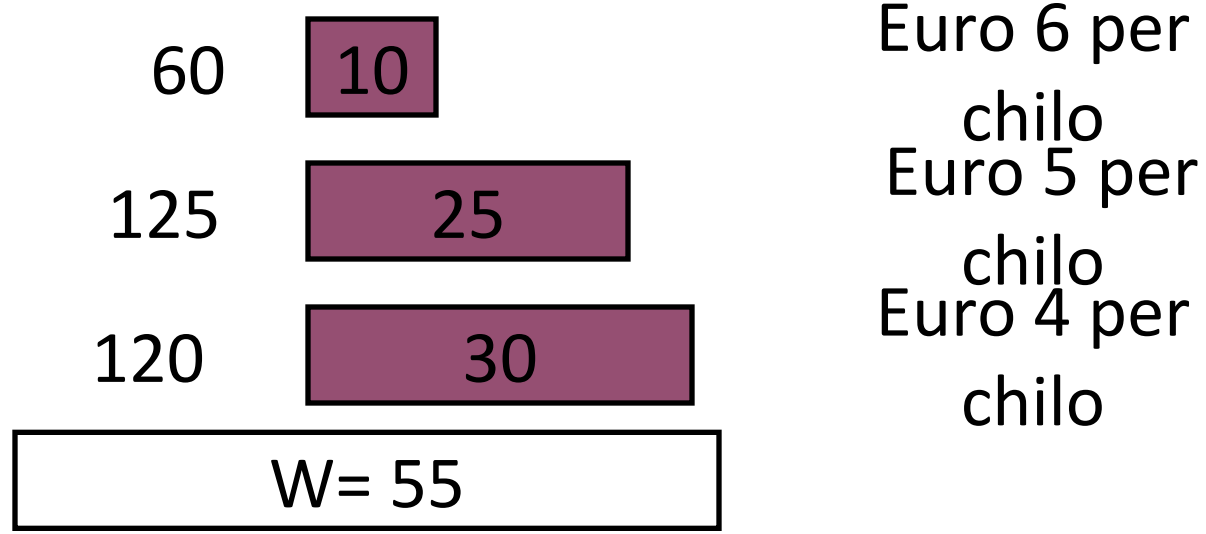
P[j] prezzo giorno j; A[j] il costo di acquisto minimo per vendere il giorno j

```
A[2]= P[1] // inizializzo: se vendo il giorno 2 devo comprare il giorno 1
G= P[2]- P[1]
for j=3 to n
  // calcolo max guadagno quando vendo un giorno in {3,…j}        A[j] = min (A[j-2], P[j-1])
   G=  max (G, P[j] – A[j])
print(G)
```

- Esercizio: aggiungere al programma il calcolo dei giorni in cui conviene vendere/comprare

Problema dello Zaino

- Ci sono n oggetti a disposizione.
- L'oggetto i-esimo pesa w_i chili e vale v_i euro
- Vogliamo caricare uno zaino con un carico di oggetti in modo da massimizzare il valore complessivo del carico ma di non superare un peso fissato di W chili.
- Si possono anche inserire *porzioni di oggetti* di meno di p_i chili. Ad esempio se un oggetto pesa 10 ne posso prendere una frazione f, $0<f<1$



Scegli per primo l'oggetto che ha il più alto valore

25 | 30 £125 + £120 = £245

Scegli per primo l'oggetto che ha il maggior peso

30 | 25 £120 + £125 = £245

Scegli per primo l'oggetto che ha il più alto valore per chilo

10 | 25 | (2/3)30

£60 + £125+ £80 = £265

Nota: del terzo oggetto si in-serisce una frazione pari a 2/3

Scelta greedy

Teorema: Il problema dello zaino frazionario soddisfa la *scelta greedy*.
Dimostrazione (cenni):
Possiamo dimostrare che: *nella soluzione ottima deve comparire la quantità massima dell'oggetto con il maggior rapporto profitto/peso* (v_i/w_i) (*scelta greedy*).
Successivamente si può dimostrare che: *la scelta greedy può sempre essere fatta per prima*.

Scelta greedy

Nella soluzione ottima deve comparire la quantità massima dell'oggetto con il maggior rapporto profitto/peso (v_i/w_i) (*scelta greedy*).

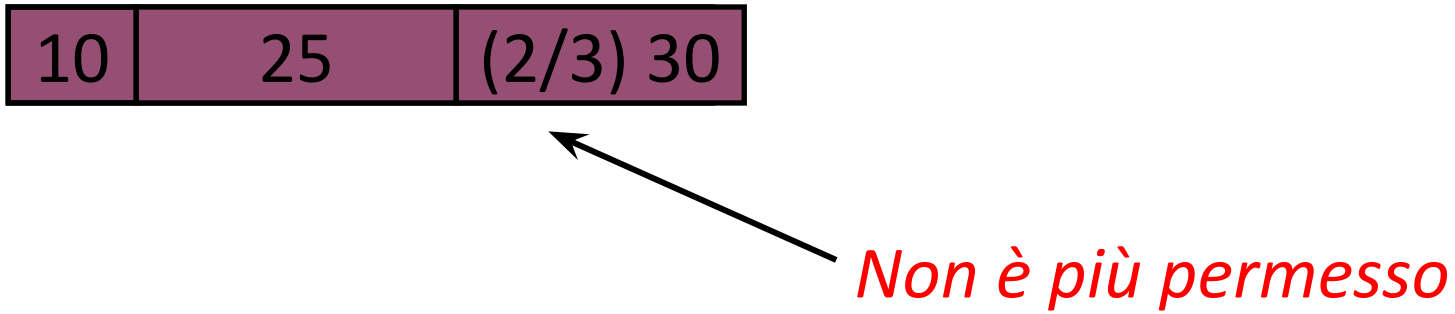
Siano v_h e w_h valore e peso disponibile dell'oggetto col massimo rapporto v_i/w_i e w_i il peso dell'*i*-esimo oggetto inserito nella soluzione.

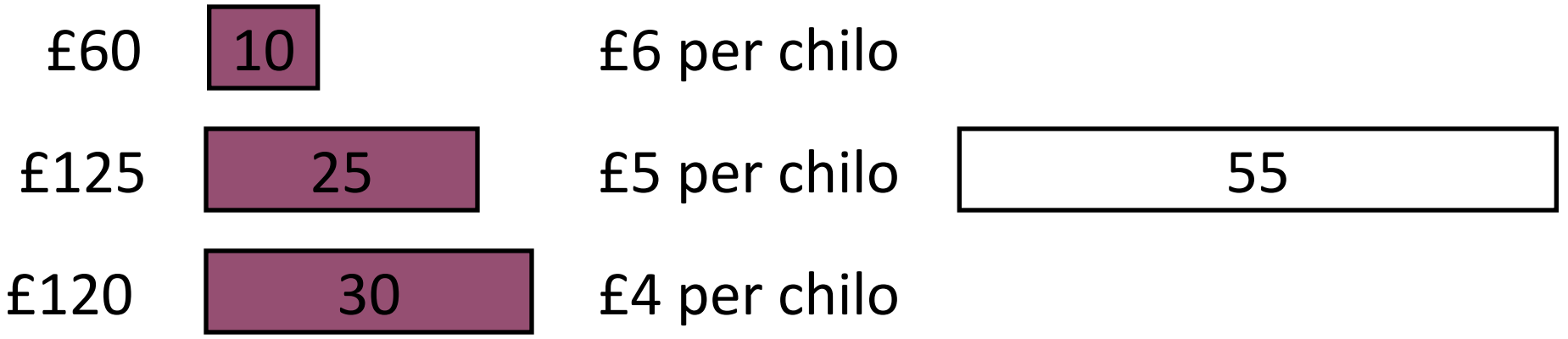
Se $w_j \neq 0$ e solo una parte dell'oggetto *h* è nella soluzione allora sostituendolo nella soluzione al posto di *j* otteniamo una soluzione migliore, infatti se togliamo un frazione x di j

- Perdiamo di $x\ v_j$ di profitto ma guadagniamo un peso $x\ w_j$
- Il peso guadagnato ci permette un profitto pari a $(x\ w_j)\ v_h / w_h$
- Complessivamente la differenza è $(x\ w_j)\ v_h / w_h - x\ v_j = x[(w_j\ v_h / w_h) - v_j]$
- Dato che $v_h / w_h > v_j / w_j$ il guadagno è positivo

Problema dello Zaino 0-1

- Identica formulazione del problema dello zaino eccettuato che
  - un oggetto deve essere preso tutto
  - oppure preso tutto

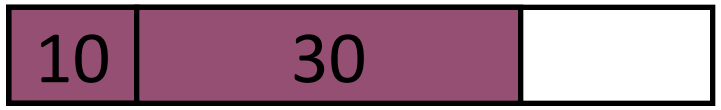




Scegli per primo l'oggetto che ha il più alto valore per chilo

£60 + £120 + £80 = £260

£60 + £125 = £185



£60 + £120 = £180

£120 + £125 = £245

Scelta greedy

Il problema dello zaino 0-1 *non* soddisfa la proprietà di *scelta greedy*.

£60 + £125 = £185

£120 + £125 = £245

Sottostruttura ottima

Teorema: Se S è soluzione ottima con insieme di oggetti I capacità W allora se rimuovo un oggetto h da S ottengo la soluzione ottima del problema con oggetti I' = I – {h} e peso W- w_h

Prova: Sia $S \subseteq I$ la soluzione ottima al problema con oggetti I e capacità W (*I* è l'insieme complessivo degli *n* oggetti, $i \in I$ ha peso $w_{i)}$

Il *profitto* di *S* sarà quindi $\Sigma_{i \in S} v_i$ e il peso di S $W = \Sigma_{i \in S} w_i$

Se rimuoviamo l'oggetto *h* da *S* otteniamo una sol. *S'* per il sottoproblema $W - p_h$ con *n*-1 oggetti (*h* escluso).

S' deve essere ottima! Perché?

Sottostruttura ottima

Teorema: Il problema dello zaino 0-1 soddisfa la *sottostruttura ottima*.

Prova:

Prova: Sia $S \subseteq I$ la soluzione ottima al problema con oggetti I e capacità W (*I* è l'insieme complessivo degli *n* oggetti, $i \in I$ ha peso $w_{i)}$

Il *profitto* di *S* sarà quindi $\Sigma_{i \in S} v_i$ e il peso di S $W = \Sigma_{i \in S} w_i$

Se rimuoviamo l'oggetto *h* da *S* otteniamo una sol. *S'* per il sottoproblema $W - p_h$ con n-1 oggetti (*h* escluso).

S' deve essere ottima! Perché?

Per contraddizione

Supponi che S' non è ottima e S'' è l'ottimo, S'≠ S'' allora

se inserisco oggetto h in S'' ottengo una soluzione migliore di S contraddizione

Zaino 0-1: programmazione dinamica

Problema dello zaino con n oggetti e Massimo peso W

Sottoproblemi

$M(i,w)$ = **massimo** valore dello zaino con peso al pari o minore di w e con solo gli oggetti 1,…,i disponibili.

- Ovviamente M(n,W) è la soluzione ottima del problema dato
- Inoltre osserviamo

$M(0,w) = 0$ per ogni w *(Non considero alcun oggetto)*

$M(i,0) = 0$ per ogni i *(Non permetto alcun oggetto con peso 0)*

$M(i,w) = M(i\text{-}1,w)$ per ogni i e per $w < w_i$

$M(i,w) = \max\{ v(i\text{-}1,w), v(i\text{-}1,w\text{-}w_i)+v_i \}$ altrimenti

Algoritmo per lo Zaino 0-1

Def. M(i, w) = valore della soluzione ottima per gli oggetti 1, …, i con limite di peso totale w.

- Caso 1: La soluzione ottima per i primi i oggetti, con limite di utilizzo w non include l'oggetto i.

- La soluzione ottima è in questo caso la soluzione ottima per {1, 2, …, i-1} con limite di utilizzo w in questo caso M(i, w) = M(i-1, w)

- Caso 2: La soluzione ottima per i primi i oggetti, con limite di utilizzo w include l'oggetto i.

- La soluzione ottima include la soluzione ottima per { 1, 2, …, i–1 } con limite di utilizzo w– w_i è in questo caso M(i, w) = v_i +M(i-1, w-w_i)

Algoritmo per lo Zaino 0-1

- Caso 1: La soluzione ottima OPT(i, w) = OPT(i-1, w)
- Caso 2: La soluzione ottima è OPT(i, w) = v_i +OPT(i-1, w-w_i)

La soluzione ottima per i primi i oggetti con limite di utilizzo w va ricercata tra le soluzioni ottime per i due casi. Questo pero` se i>0 e wi <w.

- Se i=0, banalmente si ha OPT(i,w)=0.
- Se w_i >w, e` possibile solo il caso 1 perche' i non puo` far parte della soluzione in quanto ha peso maggiore del peso trasportabile.

In conclusione

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ M(i-1,w),\ v_i + M'(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Problema dello zaino: algoritmo di programmazione dinamica

Knapsack $(n, w_1, \ldots, w_n, v_1, \ldots, v_n, W)$

```
for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 0 to W
        if (wi > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Complessità algoritmo di programmazione dinamica

Knapsack $(n, w_1, \ldots, w_n, v_1, \ldots, v_n, W)$

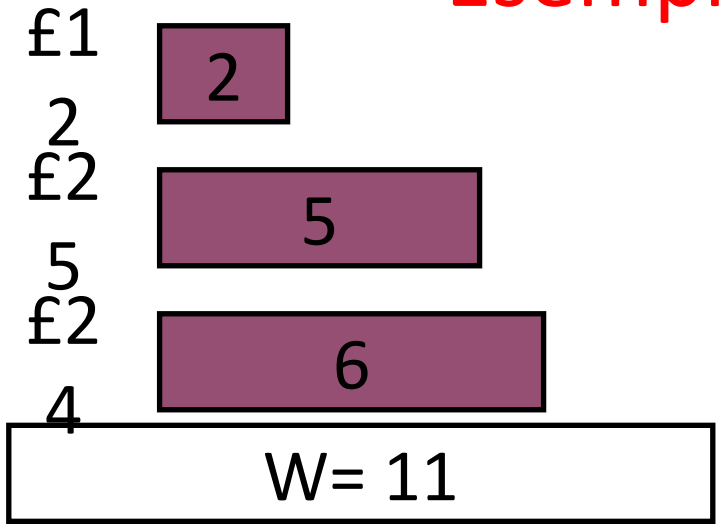
```
for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

- Chiaramente la complessità dell'algoritmo è O(nW)
- Nota O(nW) **NON è polinomiale nella lunghezza dell'input**: per rappresentare W utilizziamo $(\log_2 W)$ bit.
- Lo stesso vale per tutti i pesi e i profitti degli oggetti
- Quindi l'algoritmo di programmazione dinamica per il problema dello zaino 0-1 ha un costo esponenziale nella lunghezza dell'input.

Esempio



$M(0,w) = 0$ per ogni w

$M(i,0) = 0$ per ogni i

$M(i,w) = M(i\text{-}1,w)$ per ogni i e per $w < p_i$

$M(i,w) = \max\{ M(i\text{-}1,w), M(i\text{-}1,w\text{-}w_i)+v_i \}$

i \ *w*	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	12	12	12	12	12	12	12	12	12	12
2	0	0	12	12	12	25	25	37	37	37	37	37
3	0	0	12	12	12	25	25	37	37	37	37	49

Matrice finale con 3 oggetti e zaino di capacità pari a 11

- La prima riga rappresenta le soluzioni con 0 oggetti disponibili
- La casella in rosso rappresenta la soluzione ottima con zaino capac. 11 e oggetti scelti fra tutti i disponibili
- la casella in giallo rappresenta la soluzione ottima con zaino capac. 5 e oggetti scelti fra tutti i disponibili
- La casella in verde rappresenta la soluzione ottima con zaino capac. 11 e oggetti scelti fra i primi due (escludendo oggetto 3)

Conclusioni: problema dello zaino

- Il problema mostra *sottostruttura ottima*
- Problema dello Zaino frazionario
  - La tecnica Greedy funziona
  - Usare programmazione dinamica è inutilmente complicato (il metodo è sovradimensionato)
- Problema dello Zaino 0-1
  - La tecnica Greedy non funziona
  - È necessaria la programmazione dinamica

Esercizio

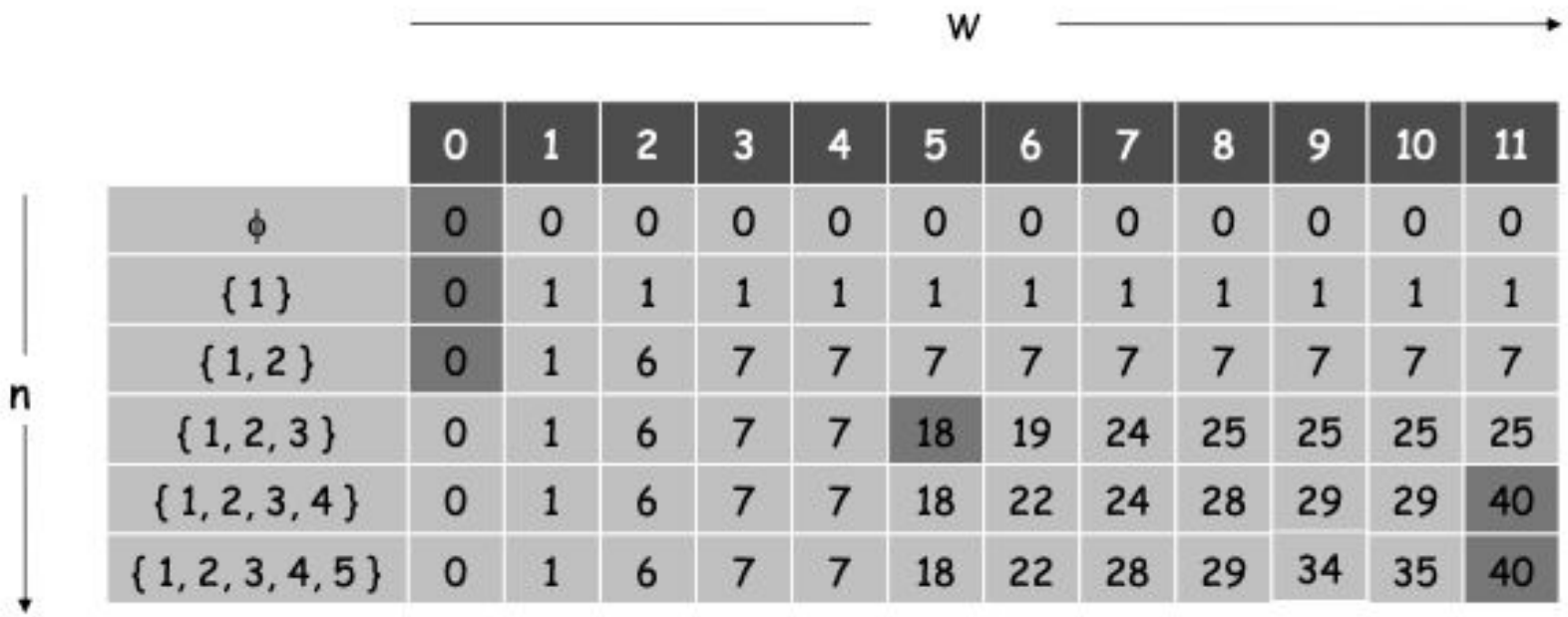
Applicare l'algoritmo di programmazione dinamica alla seguente istanza del problema dello zaino

- Capacità zaino W=11
- 5 oggetti i cui valori sono rappresentati in tabella

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Esercizio: soluzione

La tabella finale calcolata dall'algoritmo:

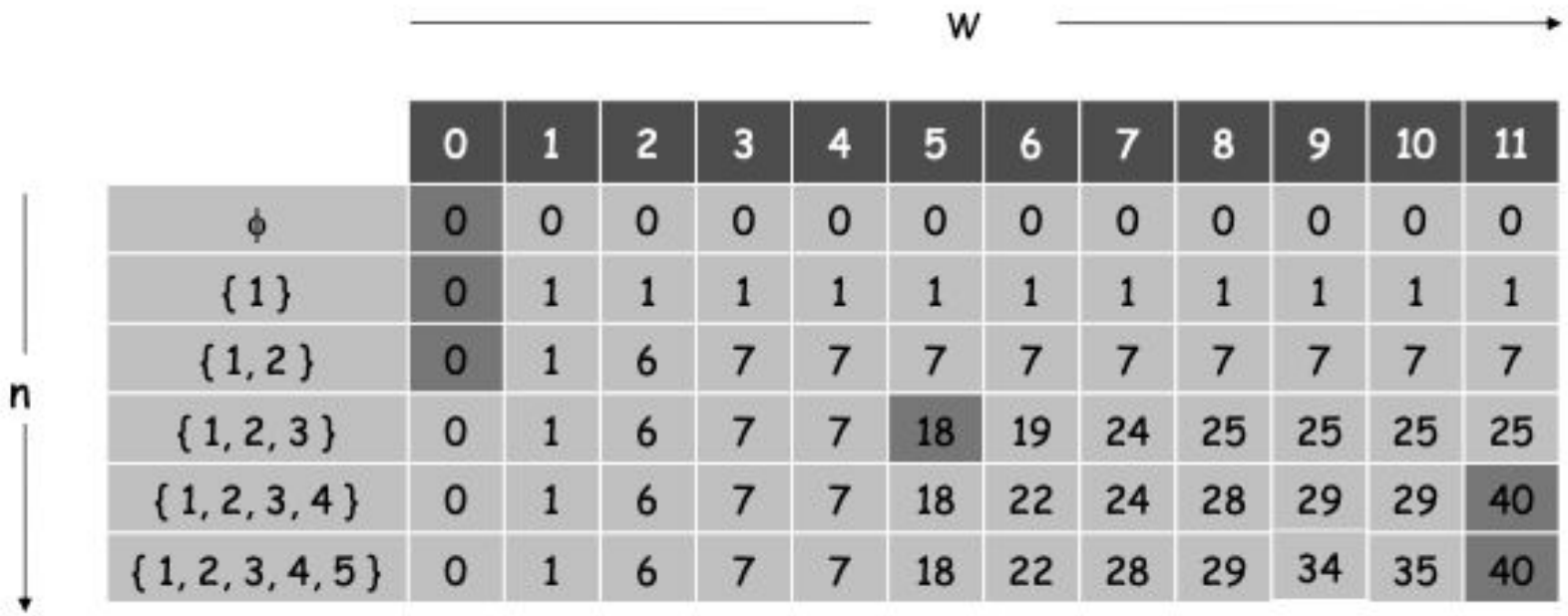


	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	35	40

Domanda che soluzioni rappresentano le tre caselle scure nella tabella?

Esercizio: soluzione

La tabella finale calcolata dall'algoritmo:



	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	35	40

Domanda che soluzioni rappresentano le tre caselle scure nella tabella?

Risposta

Dall'alto in basso sono

- La soluzione ottima per capacità zaino pari a 5 e oggetti nell'insieme {1,2,3}
- La soluzione ottima per capacità zaino pari a 11 e oggetti nell'insieme {1,2,3,4}
- La soluzione ottima per capacità zaino pari a 11 e oggetti nell'insieme {1,2,3,4,5}

Min-Zaino: una variante del problema Zaino

Il seguente problema Min-Zaino e' la versione minimizzazione del problema dello zaino

Min Zaino: input n oggetti; sia I l'insieme degli oggetti

- L'oggetto i-esimo pesa w_i chili e costa v_i euro
- Vogliamo individuare un sottoinsieme di oggetti S in modo da minimizzare il costo complessivo del carico ma che abbiano un peso complessivo di almeno W chili.

Consideriamo due varianti del problema:

- variante frazionaria: si possono anche inserire *porzioni di oggetti*
- Variante 0-1: un oggetto e'preso interamente o non e' preso affatto.

Min-Zaino: una variante del problema Zaino

Dimostrare le seguenti proprietà del problema

- Min-Zaino evidenzia una *sottostruttura ottima delle soluzioni*
- Min Zaino frazionario
  - La tecnica Greedy funziona
- Min Zaino 0-1
  - La tecnica Greedy non funziona
  - Fornire un algortimo di programmazione dinamica

Le prove/gli esempi sono analoghi a quanto visto per la versione Max-Zaino

Problema delle parole equivalenti di Lewis Carrol

Nel 1879, Lewis Carroll (l'autore di *'Alice nel paese delle meraviglie*') propose ai lettori di Vanity Fair il seguente enigma: trasformare una parola inglese in un'altra attraversando una serie di parole inglesi intermedie, dove ogni parola nella sequenza differisce dalla successivo con una sola sostituzione.

Per trasformare head in tail possiamo fare

head --> heal □ teal □ tell □ tall □ tail.

Diciamo che due parole v e w sono equivalenti se v può essere trasformata in w sostituendo singole lettere in modo tale che tutte le parole intermedie siano parole inglesi presenti in un dizionario inglese.

Trova un algoritmo per risolvere il seguente problema sulle Parole

Problema delle parole equivalenti di Lewis Carrol

- Date due parole e un dizionario, scopri se le parole sono equivalenti.
- Input: il dizionario, D (un insieme di parole) e due parole v e w dal dizionario.
- Risultato: una trasformazione di v in w mediante sostituzioni tali che tutte le parole intermedie appartengano a D.
- Se non è possibile alcuna trasformazione, l'output "v e w non sono equivalenti

Problema delle parole equivalenti di Lewis Carrol: soluzione

Dato un dizionario D e due parole v e w definiamo un grafo G=(V,E) (non diretto) tale che

- I nodi del grafo corrispondono alle parole del dizionario (ogni parola rappresenta un nodo)
- Esiste un arco fra i nodi che rappresentano due x e y se le parole associate ai due nodi differiscono per una sostituzione

Teorema: E' possibile trasformare v in w con una sequenza di sostituzioni ⍰⍰ (se e solo se) esiste un cammino in G fra i due nodi associati alle parole v e w