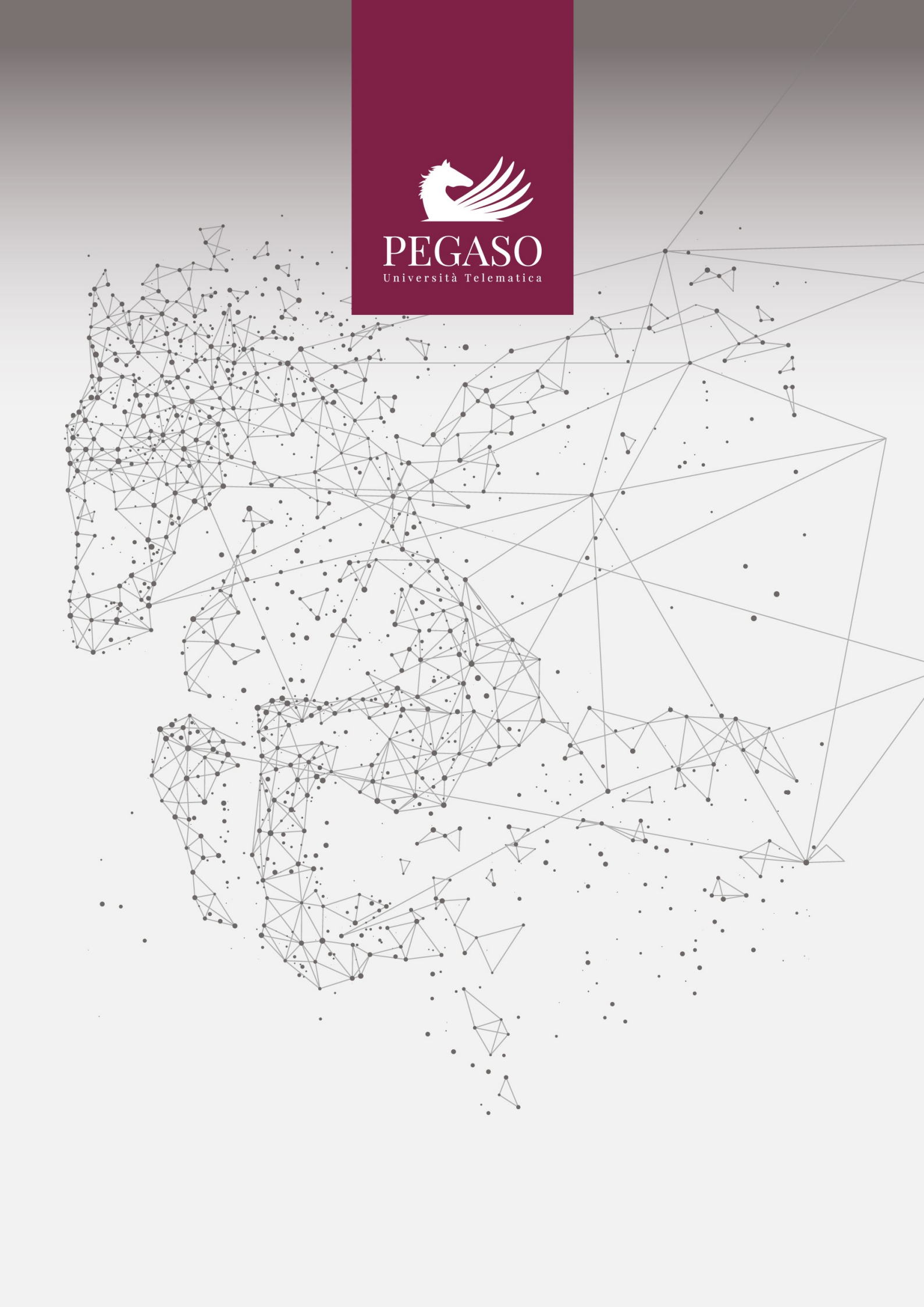
PEGASO
Università Telematica

Indice

1. Programmazione dinamica

Quando dobbiamo affrontare un problema possiamo approcciare diverse tecniche di risoluzione:

- Divide-et-impera
- Programmazione dinamica / memoization
- Tecnica greedy
- Ricerca locale
- Backtrack
- Algoritmi probabilistici
- Tecniche di soluzione per problemi intrattabili

Focalizziamo la nostra attenzione sulla **programmazione dinamica**.

La tecnica di programmazione dinamica, simile al metodo divide et impera, permette di risolvere problemi combinando le soluzioni dei sottoproblemi. Tuttavia, mentre il metodo divide et impera suddivide il problema in sottoproblemi indipendenti, la programmazione dinamica si applica quando i sottoproblemi hanno in comune dei sottosottoproblemi. In questo modo, un algoritmo divide et impera può svolgere più lavoro del necessario, risolvendo ripetutamente i sottosottoproblemi comuni.

Per evitare questo problema, la programmazione dinamica risolve ogni sottoproblema una sola volta e salva la sua soluzione in una tabella. Ciò permette di evitare il lavoro di ricalcolare la soluzione ogni volta che si presenta il sottoproblema. La programmazione dinamica si applica in particolare ai problemi di ottimizzazione, dove esistono molte soluzioni possibili, ciascuna con un valore associato. L'obiettivo è trovare la soluzione ottima, ovvero quella con il valore massimo o minimo.

Il processo di sviluppo di un algoritmo di programmazione dinamica può essere suddiviso in quattro fasi:

1. nella prima fase si cerca di caratterizzare la struttura di una soluzione ottima
2. nella seconda fase si definisce in modo ricorsivo il valore di una soluzione ottima
3. nella terza fase si calcola il valore di una soluzione ottima secondo uno schema bottom-up, ovvero partendo dal basso verso l'alto
4. nella quarta fase si costruisce la soluzione ottima dalle informazioni calcolate

Le prime tre fasi formano la base per risolvere un problema applicando la programmazione dinamica, mentre la quarta fase può essere omessa se si cerca soltanto il valore di una soluzione ottima.

Tuttavia, a volte durante la fase 4 si inseriscono informazioni aggiuntive per ottimizzare ulteriormente la soluzione.

Sintetizzando:

- Si divide un problema ricorsivamente in sottoproblemi
- Ogni sottoproblema viene risolto una volta sola
- La sua soluzione viene memorizzata in una tabella
- Nel caso un sottoproblema debba essere risolto **nuovamente**, si ricerca la sua soluzione dalla tabella (essendo già stato risolto, la soluzione è stata scritta in tabella).

ATTENZIONE: la tabella è facilmente indirizzabile – lookup in $O(1)$

L'approccio generale è quello per cui abbiamo a che fare con problemi di ottimizzazione per i quali cerchiamo la definizione della soluzione in maniera ricorsiva, in particolare il valore della soluzione in maniera ricorsiva (essendo un problema di ottimizzazione, il valore sarà ad es. un minimo, un massimo ecc.); a questo punto, se vi sono dei problemi "ripetuti" posso utilizzare la **programmazione dinamica** o una sua specializzazione che è la **memoization**. Ovviamente, se non vi sono problemi ripetuti posso utilizzare un classico approccio **divide et impera**.

Sia la programmazione dinamica che la memoization producono una tabella delle soluzioni.

Ricordiamo che il termine programmazione dinamica (Dynamic Programming) è stato coniato da Richard Bellman agli inizi degli anni '50, nell'ambito dell'ottimizzazione matematica: inizialmente, si riferiva al processo di risolvere un problema compiendo le migliori decisioni una dopo l'altra:

- "Dynamic" doveva dare un senso "temporale"
- "Programming" si riferiva all'idea di creare "programmazioni ottime", per esempio nel campo della logistica

2. Hateville

Hateville è un villaggio particolare, composto da n case, numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati: il vicino x odia i vicini $x-1$ e $x+1$ (se esistenti).

Hateville vuole organizzare una sagra: ogni abitante x ha intenzione di donare una quantità $D[x]$, ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.

Qual è la quantità massima di fondi che può essere raccolta?

Consideriamo un vicino i. Gli scenari possibili sono:

- Il vicino i fa la donazione
- Il vicino i non fa la donazione

Se il vicino i non fa la donazione, abbiamo che $DP[i] = DP[i-1]$ cioè il numero di donazioni sarà determinato dagli altri vicini; se il vicino i fa la sua donazione, il numero di donazioni sarà dato dalla sua donazione, unita a quelle di $i-2$ vicini (perché il vicino $i+1$ ed $i-1$ non doneranno) e dunque: $DP[i] = D[i] + DP[i-2]$

Come faccio a scegliere tra 2 opzioni possibili?

$DP[i] = max(DP[i-1], D[i] + DP[i-2])$

Questo scenario ovviamente vale per $i \geq 2$; ovviamente occorre includere anche i casi base che sono:

$$i = 0 \rightarrow DP[i] = 0$$
$$i = 1 \rightarrow DP[i] = D[1]$$

```
int hateville(int[ ] D, int n)
int[] DP = new int[0…n]
DP[0] = 0
DP[1] = D[1]
for i = 2 to n
     DP[i] = max(DP[i-1], DP[i-2] + D[i])
return DP[n]
```

Consideriamo il seguente vettore delle quantità:

$$D = [10, 5, 5, 8, 4, 7, 12]$$

Costruiamo la tabella inserendo per il momento solo i valori relativi alle disponibilità di ogni vicino i, ed i settaggi iniziali dell'algoritmo per $i < 2$ cioè $DP[0] = 0$ e $DP[1] = D[1]$:

n	0	1	2	3	4	5	6	7
D	-	10	5	5	8	4	7	12
DP	0	10	-	-	-	-	-	-

Figura 1: Tabella DP

Occorre applicare la seguente formula:

$$DP[i] = max(DP[i-1], DP[i-2] + D[i])$$

Pertanto:

- $DP[2] = max(DP[1], DP[0] + D[2]) = max(10,5) = 10$

n	0	1	2	3	4	5	6	7
D	-	10	5	5	8	4	7	12
DP	0	10	10	-	-	-	-	-

Figura 2: DP[2]

- $DP[3] = max(DP[2], DP[1] + D[3]) = max(10,10+5) = 15$

n	0	1	2	3	4	5	6	7
D	-	10	5	5	8	4	7	12
DP	0	10	10	15	-	-	-	-

Figura 3: DP[3]

- $DP[4] = max(DP[3], DP[2] + D[4]) = max(15,10+8) = 18$

n	0	1	2	3	4	5	6	7
D	-	10	5	5	8	4	7	12
DP	0	10	10	15	18	-	-	-

Figura 4: DP[4]

- $DP[5] = max(DP[4], DP[3] + D[5]) = max(18,15 + 4) = 19$
- $DP[6] = max(DP[5], DP[4] + D[6]) = max(19,18 + 7) = 25$
- $DP[7] = max(DP[6], DP[5] + D[7]) = max(19,19 + 12) = 31$

Risultato finale:

n	0	1	2	3	4	5	6	7
D	-	10	5	5	8	4	7	12
DP	0	10	10	15	18	19	25	31

Figura 5: Tabella Hateville

ATTENZIONE: in DP abbiamo il valore totale ma non sappiamo quali vicini hanno contribuito; per questo possiamo operare nel seguente modo:

- si analizza l'elemento $DP[i]$
- se $DP[i] = DP[i - 1]$, la casa i non è stata selezionata
- se $DP[i] = DP[i - 2] + D[i]$, la casa i è stata selezionata
- Se entrambe le equazioni sono vere, una vale l'altra

Per ricostruire la soluzione fino ad i, lavoriamo in modo ricorsivo:

Se $DP[i] = DP[i - 1]$, si prende la soluzione fino a $i - 1$ senza aggiungere nulla, altrimenti, si prende la soluzione fino a $i - 2$ e si aggiunge i

```
Set solution(int[] DP, int[] D, int i)
if i == 0
     return Ø
else if i == 1
     return {1}
else if DP[i] == DP[i - 1]
     return solution(DP, D, i - 1)
else
     Set sol = solution(DP, D, i - 2)
     sol.insert(i)
     return sol
```

Parliamo di complessità:

- La complessità computazionale di solution() è $T(n) = \Theta(n)$
- La complessità computazionale e spaziale di hateville() è:
  - $T(n) = \Theta(n)$
  - $S(n) = \Theta(n)$

3. Il problema dello zaino

Il problema dello zaino, (**Knapsack** problem), è un problema di ottimizzazione combinatoria posto nel modo seguente:

Sia dato uno zaino che possa sopportare un determinato peso (capacità) e siano dati N oggetti, ognuno dei quali caratterizzato da un peso (weight) e un valore (profit). Il problema si propone di scegliere quali di questi oggetti mettere nello zaino per ottenere il maggiore valore senza eccedere il peso sostenibile dallo zaino stesso.

Lo zaino possiede dunque un limite di capacità: fissato questo limite, si vuole individuare un sottoinsieme di oggetti il cui peso sia inferiore alla capacità dello zaino, ma il valore totale degli oggetti deve essere massimale, cioè più alto o uguale al valore di qualunque altro sottoinsieme di oggetti.

Il nome Knapsack è stato introdotto da George B. Dantzig nel 1957.

L'input del problema è il seguente:

- Vettore w: $w[i]$ è il peso (weight) dell'oggetto i-esimo
- Vettore p: $p[i]$ è il profitto (profit) dell'oggetto i-esimo
- La capacità C dello zaino

L'output del problema è invece un insieme $S \subseteq \{1,\ldots,n\}$ tale che:

- Il peso totale deve essere minore o uguale alla capacità:

$$w(S) = \sum_{i \in S} \mathrm{w[i]} \leq \mathrm{C}$$

- Il profitto totale deve essere massimizzato:

$$\max\left(p(S) = \sum_{i \in S} \mathrm{p[i]}\right)$$

Facciamo direttamente un esempio:

- *Peso* = [4, 2 ,3, 4]
- *Profitto* = [10, 7, 8, 6]
- *capacità massima* =9

Costruiamo la tabella della programmazione dinamica:

oggetto\profitto	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0									
2	0									
3	0									
4	0									

Tale tabella avrà:

- 9 colonne (il valore della capacità dello zaino) +1 (a rappresentare il valore della capacità 0, cioè lo zaino vuoto)
- 4 righe (i possibili oggetti) +1 (a rappresentare cioè nessun oggetto)

La singola $cella(i,j)$ rappresenta il massimo valore (il massimo profitto) ottenibile con una capacità j utilizzando solo i primi i oggetti.

Ad es., se consideriamo la cella $(2,3)$ stiamo considerando il massimo valore ottenibile con una capacità 3, utilizzando solo i primi 2 oggetti.

Ovviamente la prima riga e la prima colonna sono tutti zeri.

Tale massimo profitto viene anche indicato con $DP[i][j]$ (dynamic programming)

Proviamo a ragionare nella seguente maniera: posso decidere di prendere o meno un oggetto da inserire nello zaino. Supponiamo di trovarci a considerare una capacità 5: dovrei settare i vari valori nelle celle $(0,5),(1,5),(2,5),(3,5),(4,5)$, cioè i vari $DP[0][5]$,$DP[1][5]$, $DP[2][5]$, $DP[3][5]$, $DP[4][5]$.

Ovviamente il valore in $(0,5)$ sarà pari a 0.

Ragioniamo ora su $(1,5)$: posso decidere si prendere o meno questo oggetto.

Se **non lo prendo**, di fatto il valore che dovrò mettere in (1,5) è lo stesso del precedente (0,5) perché non c'è stato alcun cambiamento a livello di profitto, e pertanto $DP[1][5] = DP[0][5]$, cioè, generalizzando: $DP[i][j] = DP[i-1][j]$.

Se **lo prendo** invece, dovrò mettere in (1,5) il valore presente in $DP[0][?]$: stavolta devo ragionare sulla colonna. In questo caso specifico, qualsiasi sia il valore della colonna che dovrò individuare, tale valore sarà sempre pari a 0, tuttavia, generalizziamo e domandiamoci cosa dovrei scrivere se mi trovassi in $DP[i-1][?]$ con un valore di i generico. **SE** ho deciso di prendere l'oggetto i, questo significa che dovrò "sottrarre" il peso di questo oggetto dalla capacità, che non sarà più 5 (come in questo esempio che stiamo facendo), ma diventerà $5 - w(1) = 5 - 4 = 1$, cioè devo sottrarre alla capacità corrente (in questo caso

5) il peso dell'oggetto che sto prendendo (in questo caso 4) e quindi la "colonna" che dovrò considerare sarà $j - w(i)$, e dunque: $DP[i][j] = DP[i-1][j-w(i)]$ ma in realtà non abbiamo finito perché, avendo scelto l'oggetto, devo aggiungere il profitto associato all'oggetto e quindi il risultato finale sarà:

$$DP[i][j] = DP[i-1][j-w(i)] + P[i]$$

Quindi cosa devo fare? L'oggetto lo devo prendere o non prendere? A tutti gli effetti quello che devo fare è:

$$DP[i][j] = \max(DP[i-1][j-w(i)] + P[i], DP[i-1][j])$$

Per essere rigorosi dovremmo anche definire cosa accade nel caso in cui mi trovassi con una capacità "negativa", che potrebbe succedere qualora scegliessi un oggetto il cui peso eccede la capacità residua (cosa che ovviamente non deve essere possibile fare); in questo caso vado a settare un profitto pari a $-\infty$.

L'algoritmo finale è il seguente:

```
int knapsack(int[] w, int[] p, int n, int C)
DP = new int[0…n][0…C]
for i = 0 to n
     DP[i][0] = 0
for j = 0 to C do
     DP[0][j] = 0
for i = 1 to n
     for j = 1 to C
          if w[i] ≤ j
               DP[i][j]= max(DP[i-1][j-w(i)]+P[i],DP[i-1][j])
          else
               DP[i][j]=DP[i – 1][j]
return DP[n][C]
```

Tornando quindi al nostro esempio e provando a riempire la colonna 5:

- DP [0][5] = 0
- DP [1][5] = max (DP[0][5 - 4] + 10), DP[0][5]) = 10
- DP [2][5] = max (DP[1][5 - 2] + 7), DP[1][5]) = max(DP[1][3] + 7,10)

Quindi capisco che devo aver calcolato precedentemente il valore di $DP[1][3]$.

È bene dunque calcolare i valori della tabella a partire dalle colonne più a sinistra per arrivare a quelle di destra.

oggetto\profitto	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

Riportiamo per comodità i parametri di interesse:

- Peso = [4, 2, 3, 4]
- Profitto = [10, 7, 8, 6]
- $DP[i][j] = \max(DP[i-1][j-w(i)] + P[i], DP[i-1][j])$

Facciamo alcuni calcoli

- $DP[2][2] = \max(DP[1][2-2] + 7, DP[1][2]) = \max(DP[1][0] + 7,0) = \max(7,0)$
- $DP[3][2] = \max(DP[2][2-3] + 8, DP[2][2]) = \max(-\infty, 7)$
- $DP[4][2] = \max(DP[3][2-4] + 6, DP[3][2]) = \max(-\infty, 7)$
- $DP[2][3] = \max(DP[1][3-2] + 7, DP[1][3]) = \max(DP[1][1] + 7,0) = \max(7,0)$
- $DP[3][3] = \max(DP[2][3-3] + 8, DP[2][3]) = \max(DP[2][0] + 8,7) = \max(8,7)$
- $DP[1][4] = \max(DP[0][4-4] + 10, DP[0][4]) = \max(10,0)$
- […]

La complessità dell'algoritmo è pari a $O(nC)$, tuttavia l'algoritmo è "pseudo-polinomiale", infatti sono necessari $k = \log_2 C$ di bit per rappresentare C e quindi la complessità è $O(n2^k)$

Di seguito l'implementazione in Python:

```
def knapsack(W, wt, val):
    n = len(val)
    dp = [[0] * (W+1) for _ in range(n+1)]

    # Fill in the DP matrix using bottom-up approach
    for i in range(1, n+1):
        for w in range(1, W+1):
            if wt[i-1] <= w:
                dp[i][w]=max(dp[i-1][w],val[i-1]+dp[i-1][w-wt[i-1]])
            else:
                dp[i][w] = dp[i-1][w]
```

```
    # Return the maximum value that can be obtained
    return dp[n][W]
```

La funzione knapsack prende tre argomenti: W, wt e val. W è la capacità massima dello zaino, wt è un array che contiene i pesi degli oggetti, e val è un array che contiene i valori degli oggetti.

Per esempio, supponiamo che si abbia uno zaino con capacità massima $W = 10$ e quattro oggetti con pesi $wt = [5, 4, 6, 3]$ e valori $val = [10, 40, 30, 50]$. Per trovare il massimo valore che si può ottenere selezionando un sottoinsieme di oggetti, si può chiamare la funzione in questo modo:

knapsack(10, [5, 4, 6, 3], [10, 40, 30, 50]) # Output: 90

Il risultato sarà 90, perché il massimo valore che si può ottenere selezionando un sottoinsieme di oggetti con peso totale non superiore a 10 è 90, che si ottiene selezionando il secondo e il quarto oggetto.

La funzione **knapsack** utilizza la programmazione dinamica per riempire una matrice dp di dimensioni $(n + 1)\ x\ (W + 1)$, dove n è il numero di oggetti. L'elemento $dp[i][w]$ contiene il massimo valore che si può ottenere selezionando un sottoinsieme degli oggetti 1 a i con peso totale non superiore a w.

La soluzione del problema viene quindi restituita dall'elemento $dp[n][W]$, che rappresenta il massimo valore che si può ottenere selezionando tutti gli n oggetti con peso totale non superiore a W.

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione .