

# Tecniche Algoritmiche / 3

# Programmazione Dinamica

Jocelyne Elias

<https://www.unibo.it/sitoweb/jocelyne.elias>

Moreno Marzolla

<https://www.moreno.marzolla.name/>

Dipartimento di Informatica—Scienza e Ingegneria (DISI)  
Università di Bologna

Copyright © Alberto Montresor, Università di Trento, Italy

<http://cricca.disi.unitn.it/montresor/teaching/asd/>



Copyright © 2010—2016, 2021 Moreno Marzolla, Università di Bologna, Italy

<https://www.moreno.marzolla.name/teaching/ASD/>

*This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit*

*<http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

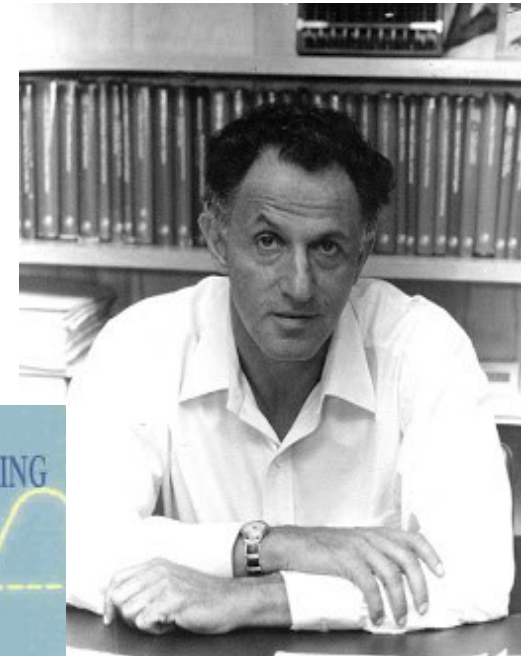
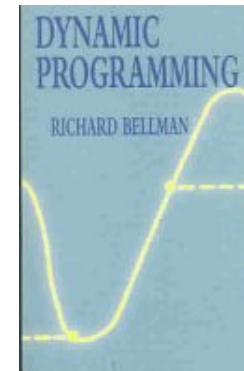
# Programmazione dinamica

- **Definizione**

- Tecnica sviluppata negli anni '50 da Richard E. Bellman
- Ambito: **problemi di ottimizzazione**
- Trovare la soluzione ottima secondo un “indice di qualità” assegnato ad ognuna delle soluzioni possibili

- **Idea**

- Risolvere un problema combinando le soluzioni di sotto-problemi
- Ma ci sono importanti differenze con divide-et-impera

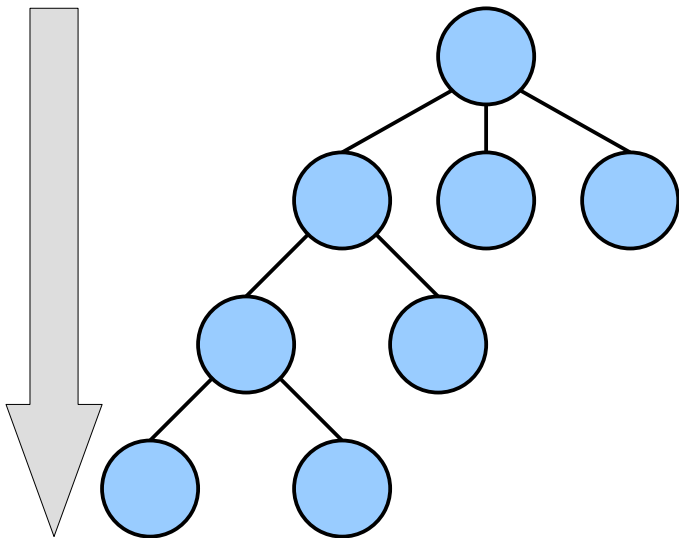


Richard Ernest Bellman (1920—1984),  
By Source, Fair use,  
<https://en.wikipedia.org/w/index.php?curid=43193672>

# Programmazione dinamica vs divide-et-impera

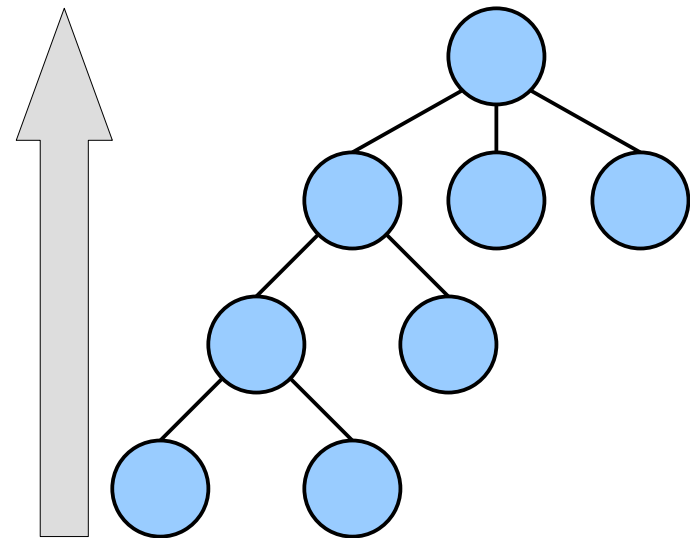
- Divide-et-impera

- Tecnica ricorsiva
- Approccio **top-down**
- Vantaggiosa quando i sottoproblemi sono **indipendenti**



- Programmazione dinamica

- Tecnica iterativa
- Approccio **bottom-up**
- Vantaggiosa quando ci sono sottoproblemi **ripetuti**



# Quando applicare la programmazione dinamica?

- Sottostruttura ottima
  - Deve essere possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più “grande”
- Sottoproblemi ripetuti
  - Un sottoproblema compare più volte

# Programmazione dinamica

- 1) Definire i (sotto-)problemi da risolvere
- 2) Definire la soluzione dei sottoproblemi
  - Cosa rappresenta la soluzione di un sotto-problema? Un valore numerico? Una stringa? Che significato ha?
- 3) Definire la soluzione del problema di partenza
  - Come posso ottenere la soluzione del problema di partenza, conoscendo le soluzioni di tutti i sotto-problemi?
- 4) Definire l'algoritmo per calcolare le soluzioni dei sotto-problemi in modo efficiente
  - Tipicamente, usando un approccio bottom-up
  - Memorizzando le soluzioni in qualche struttura dati (di solito, array o matrici) per non dover risolvere più volte lo stesso sotto-problema

# Numeri di Fibonacci





# Esempio: successione di Fibonacci

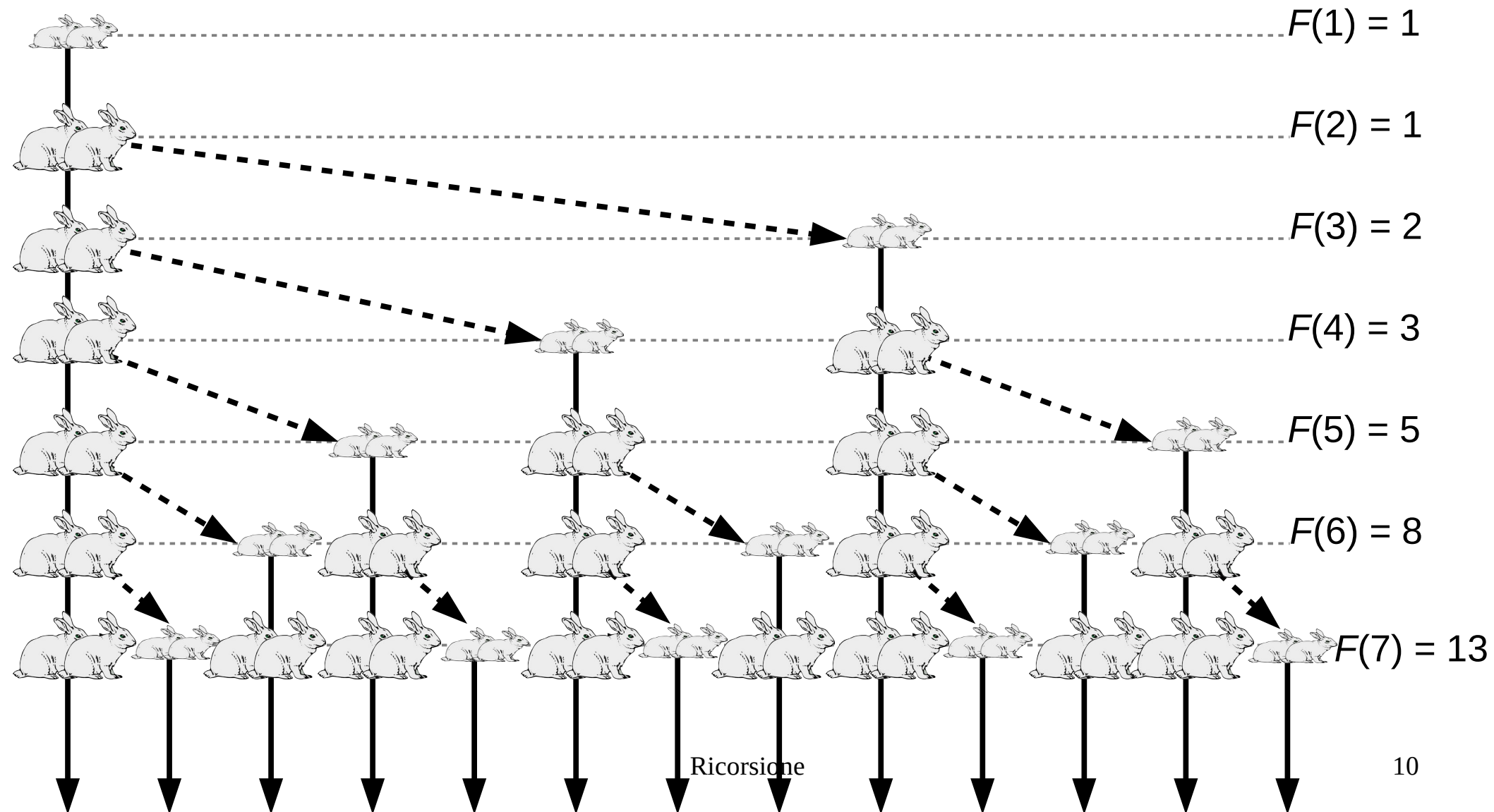
- Modello (non realistico) di crescita di una popolazione di conigli
  - Una coppia all'inizio del primo mese
  - A partire dalla **fine del secondo mese di vita**, ogni coppia produce un'altra coppia alla fine di ogni mese
  - I conigli non muoiono mai (!)
- $F(n)$  = coppie all'inizio del mese  $n$ 
  - $F(1) = 1$
  - $F(2) = 1$
  - $F(n) = F(n - 1) + F(n - 2)$  per ogni  $n > 2$



Leonardo Pisano detto Fibonacci  
(Pisa, 1170—Pisa, ca. 1250)

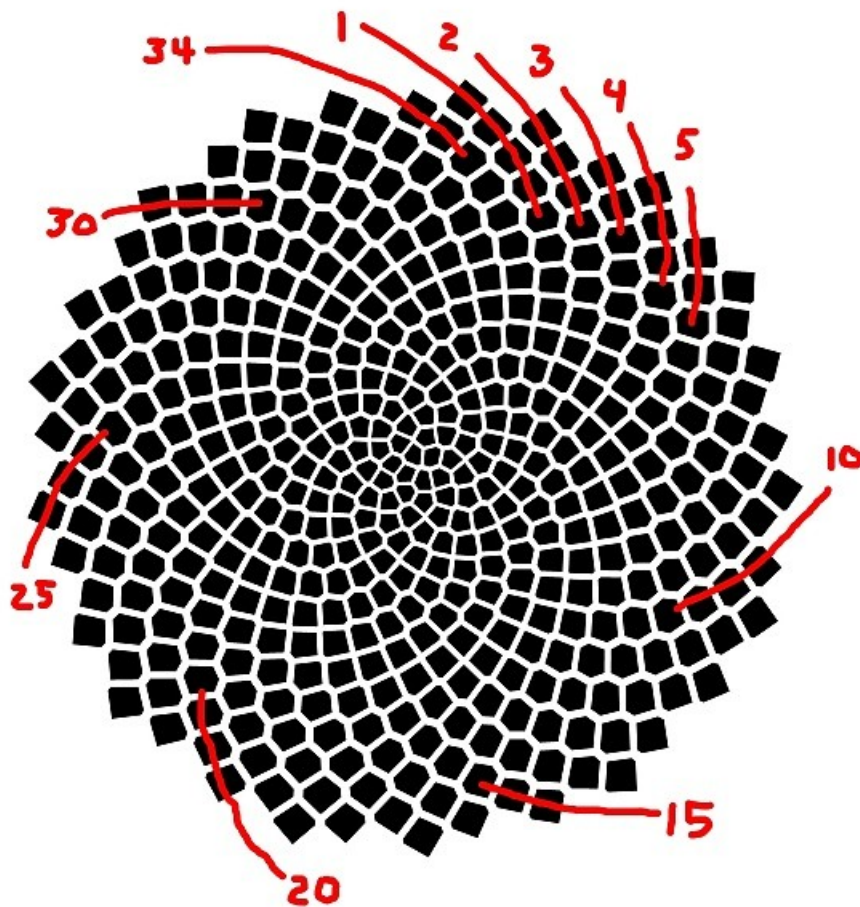
[http://it.wikipedia.org/wiki/Leonardo\\_Fibonacci](http://it.wikipedia.org/wiki/Leonardo_Fibonacci)

# I conigli di Fibonacci

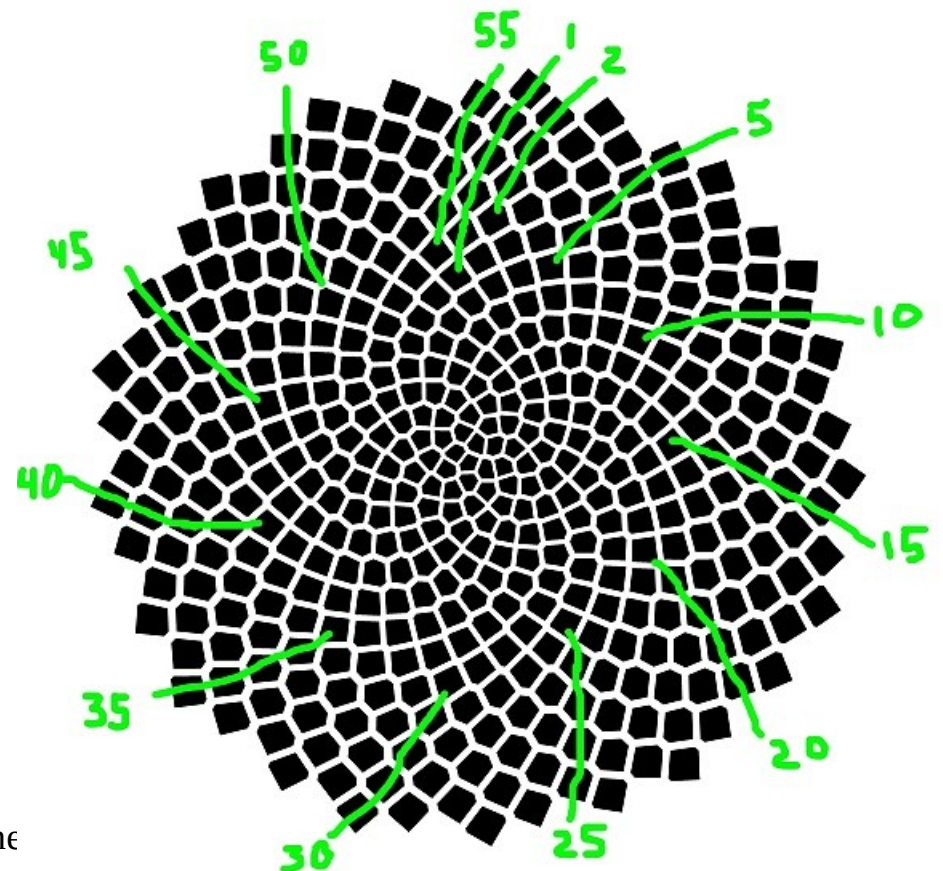


# Numeri di Fibonacci in natura

- Il numero di spirali orarie e antiorarie nei fiori di girasoli, nelle pigne e in altri vegetali (es., ananas) tendono ad assumere valori adiacenti nella successione di Fibonacci



one



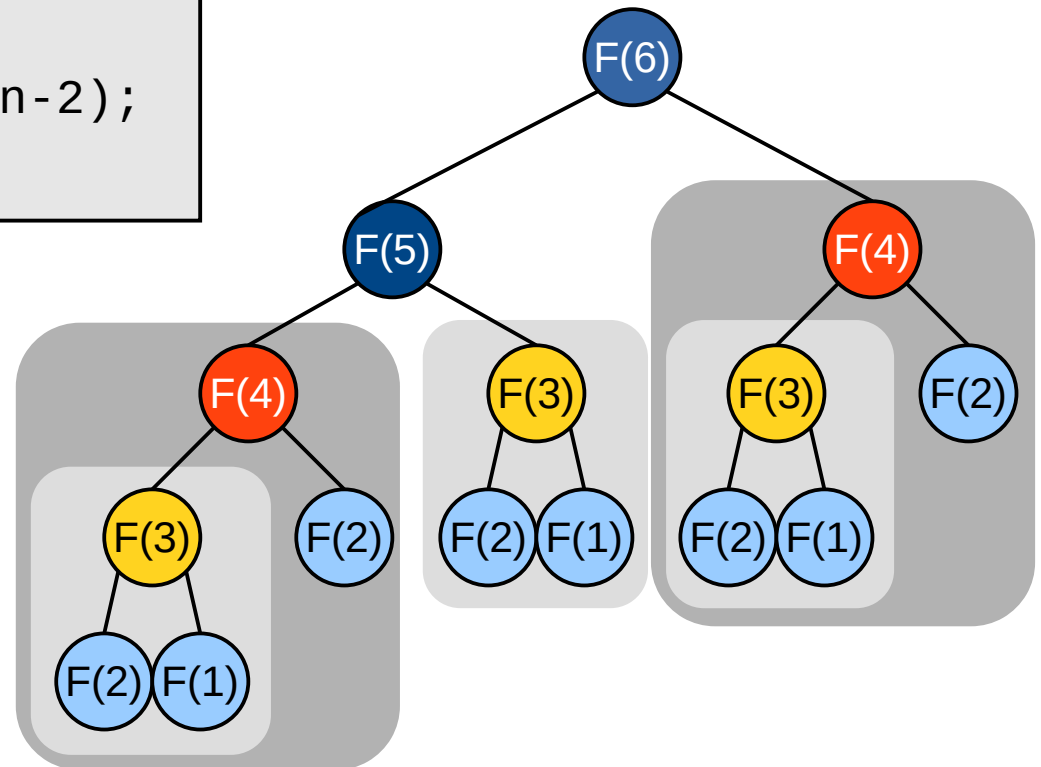
# Implementazione ricorsiva

```
integer FibRic(integer n)
  if ((n = 1) or (n = 2)) then
    return 1;
  else
    return FibRic(n-1)+FibRic(n-2);
  endif
```

- Costo asintotico

$$T(n) = \begin{cases} c_1 & n \leq 2 \\ T(n-1) + T(n-2) + c_2 & n > 2 \end{cases}$$

- Soluzione
  - $T(n) = \Omega(2^{n/2})$



# Implementazione iterativa - 1

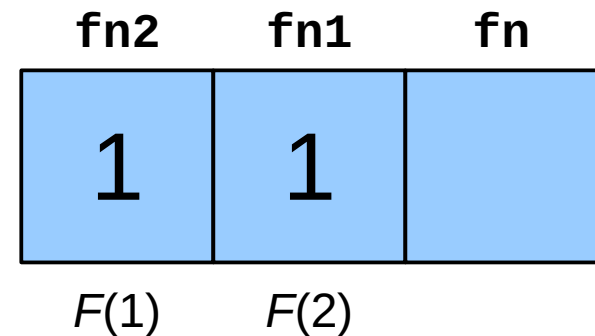
```
integer FibIter(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer f[1..n];
    f[1] ← 1;
    f[2] ← 1;
    for integer i ← 3 to n do
      f[i] ← f[i-1] + f[i-2];
    endfor
    return f[n];
  endif
```

- Complessità
  - Tempo:  $\Theta(n)$
  - Spazio:  $\Theta(n)$

n	1	2	3	4	5	6	7	8
f[]	1	1	2	3	5	8	13	21

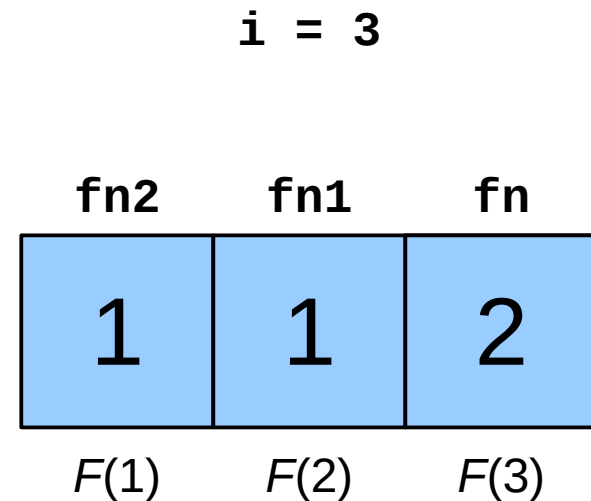
# Implementazione iterativa - 2

```
integer FibIter2(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer fn1, fn2, fn;
    fn1 ← 1;
    fn2 ← 1;
    for integer i ← 3 to n do
      fn ← fn1 + fn2;
      fn2 ← fn1;
      fn1 ← fn;
    endfor
    return fn;
  endif
```



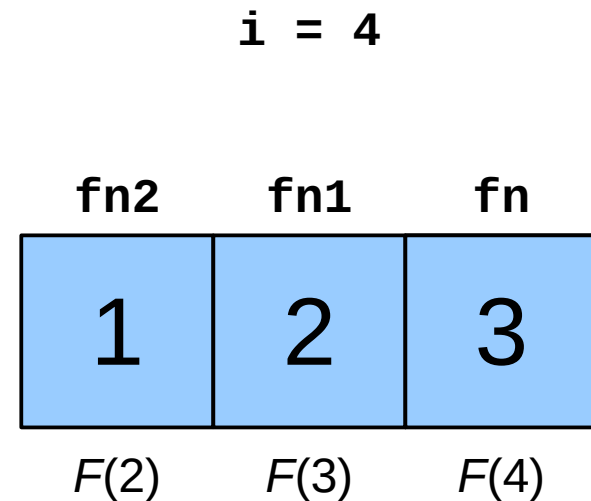
# Implementazione iterativa - 2

```
integer FibIter2(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer fn1, fn2, fn;
    fn1 ← 1;
    fn2 ← 1;
    for integer i ← 3 to n do
      fn ← fn1 + fn2;
      fn2 ← fn1;
      fn1 ← fn;
    endfor
    return fn;
  endif
```



# Implementazione iterativa - 2

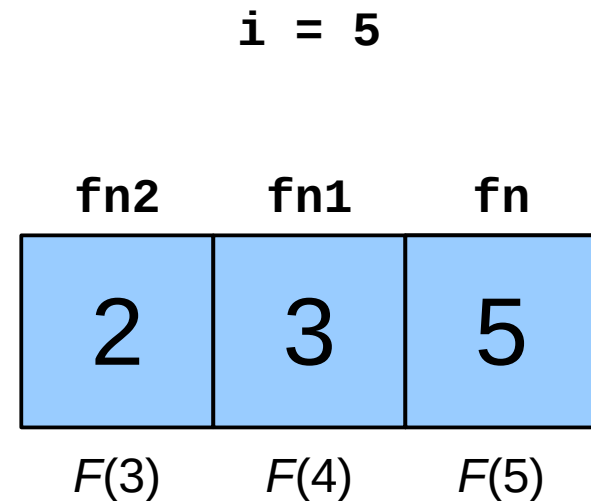
```
integer FibIter2(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer fn1, fn2, fn;
    fn1 ← 1;
    fn2 ← 1;
    for integer i ← 3 to n do
      fn ← fn1 + fn2;
      fn2 ← fn1;
      fn1 ← fn;
    endfor
    return fn;
  endif
```





# Implementazione iterativa - 2

```
integer FibIter2(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer fn1, fn2, fn;
    fn1 ← 1;
    fn2 ← 1;
    for integer i ← 3 to n do
      fn ← fn1 + fn2;
      fn2 ← fn1;
      fn1 ← fn;
    endfor
    return fn;
  endif
```



# Implementazione iterativa - 2

```
integer FibIter2(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer fn1, fn2, fn;
    fn1 ← 1;
    fn2 ← 1;
    for integer i ← 3 to n do
      fn ← fn1 + fn2;
      fn2 ← fn1;
      fn1 ← fn;
    endfor
    return fn;
  endif
```

- Complessità
  - Tempo:  $\Theta(n)$
  - Spazio:  $O(1)$ 
    - Array di 3 elementi

# Sottovettore di somma massima

# Esempio (già visto)

## sottovettore di somma massima

- Consideriamo un vettore  $V[1..n]$  di  $n$  valori reali arbitrari
- Vogliamo individuare un sottovettore non vuoto di  $V$  la somma dei cui elementi sia massima

3	-5	10	2	-3	1	4	-8	7	-6	-1
---	----	----	---	----	---	---	----	---	----	----

- Conosciamo già soluzioni di costo:
  - $O(n^3)$
  - $O(n^2)$
  - $O(n \log n)$

# Soluzione di “forza bruta”

## Costo $O(n^3)$

```
real VecSum1( real V[1..n] )
  real smax ← V[1];
  for integer i ← 1 to n do
    for integer j ← i to n do
      real s ← 0;
      for integer k ← i to j do
        s ← s + V[k];
      endfor
      if (s > smax) then
        smax ← s;
      endif
    endfor
  endfor
  return smax;
```

# Implementazione più efficiente

## Costo $O(n^2)$

```
real VecSum2( real V[1..n] )  
  real smax ← V[1];  
  for integer i ← 1 to n do  
    real s ← 0;  
    for integer j ← i to n do  
      s ← s + V[j];  
      if (s > smax) then  
        smax ← s;  
      endif  
    endfor  
  endfor  
  return smax;
```

# Implementazione divide-et-impera

## Costo $O(n \log n)$

```
real VecSumDI( real V[1..n], integer i, integer j)
  if (i = j) then
    return V[i];
  else
    integer k, m ← (i+j)/2;
    real sleft ← VecSumDI(V, i, m-1);
    real sright ← VecSumDI(V, m+1, j);
    real sa, sb, s;
    sa ← 0; s ← 0;
    for k ← m-1 downto i do
      s ← s + V[k]
      if (s > sa) sa ← s;
    endfor
    sb ← 0; s ← 0;
    for k ← m+1 to j do
      s ← s + V[k];
      if (s > sb) sb ← s;
    endfor
    return max(sleft, sright, V[m] + sa + sb);
  endif
```

# Soluzione basata su programmazione dinamica

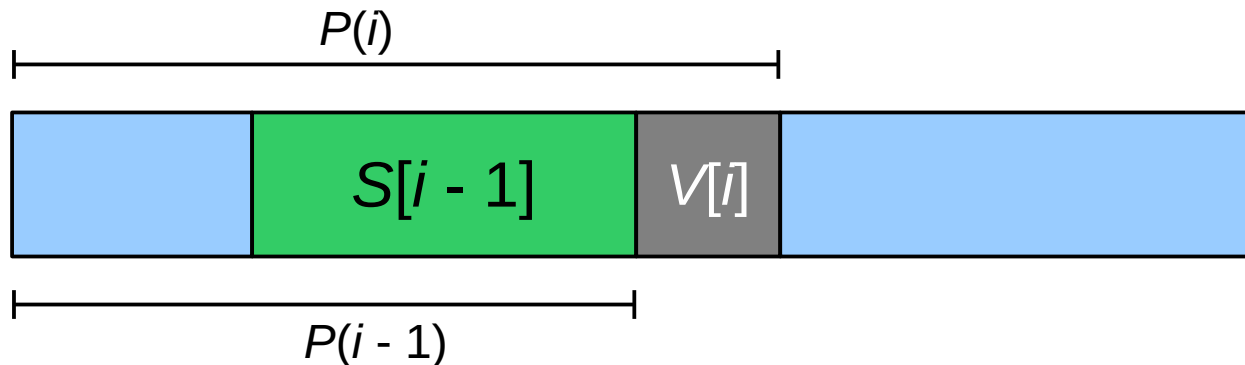
- Sia  $P(i)$  il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori non vuoti del vettore  $V[1..i]$  che hanno  $V[i]$  come ultimo elemento
- Sia  $S[i]$  il valore della soluzione di  $P(i)$ 
  - $S[i]$  è la massima somma degli elementi del sottovettori di  $V[1..i]$  che hanno  $V[i]$  come ultimo elemento
- La soluzione  $S^*$  al problema di partenza può essere espressa come

$$S^* = \max_{1 \leq i \leq n} S[i]$$



# Soluzione basata su programmazione dinamica

- $P(1)$  ammette una unica soluzione
  - $S[1] = V[1]$
- Consideriamo il generico problema  $P(i)$ ,  $i > 1$ 
  - Supponiamo di avere già risolto il problema  $P(i - 1)$ , e quindi di conoscere  $S[i - 1]$
  - Se  $S[i - 1] + V[i] \geq V[i]$  allora  $S[i] = S[i - 1] + V[i]$
  - Se  $S[i - 1] + V[i] < V[i]$  allora  $S[i] = V[i]$



# Esempio

V[]	3	-5	10	2	-3	1	4	-8	7	-6	-1
S[]	3	-2	10	12	9	10	14	6	13	7	6

$$S[i] = \max \{ V[i], V[i] + S[i - 1] \}$$

# L'algoritmo

```
real VecSumDP(real V[1..n])  
  real S[1..n];  
  S[1]  $\leftarrow$  V[1];  
  integer imax  $\leftarrow$  1;    // indice val. max in S  
  for integer i  $\leftarrow$  2 to n do  
    if ( S[i-1] + V[i]  $\geq$  V[i] ) then  
      S[i]  $\leftarrow$  S[i-1] + V[i];  
    else  
      S[i]  $\leftarrow$  V[i];  
    endif  
    if ( S[i] > S[imax] ) then  
      imax  $\leftarrow$  i;  
    endif  
  endfor  
  return S[imax];
```

# Come individuare il sottovettore?

- Fino ad ora siamo in grado di calcolare il **valore** della massima somma tra tutti i sottovettori di  $V[1..n]$
- Come facciamo a determinare **quale** sottovettore produce tale somma?
  - L'indice dell'elemento finale del sottovettore l'abbiamo
  - L'indice iniziale lo possiamo ricavare procedendo “a ritroso”
    - Se  $S[i] = V[i]$ , il sottovettore massimo inizia nella posizione  $i$

# Esempio

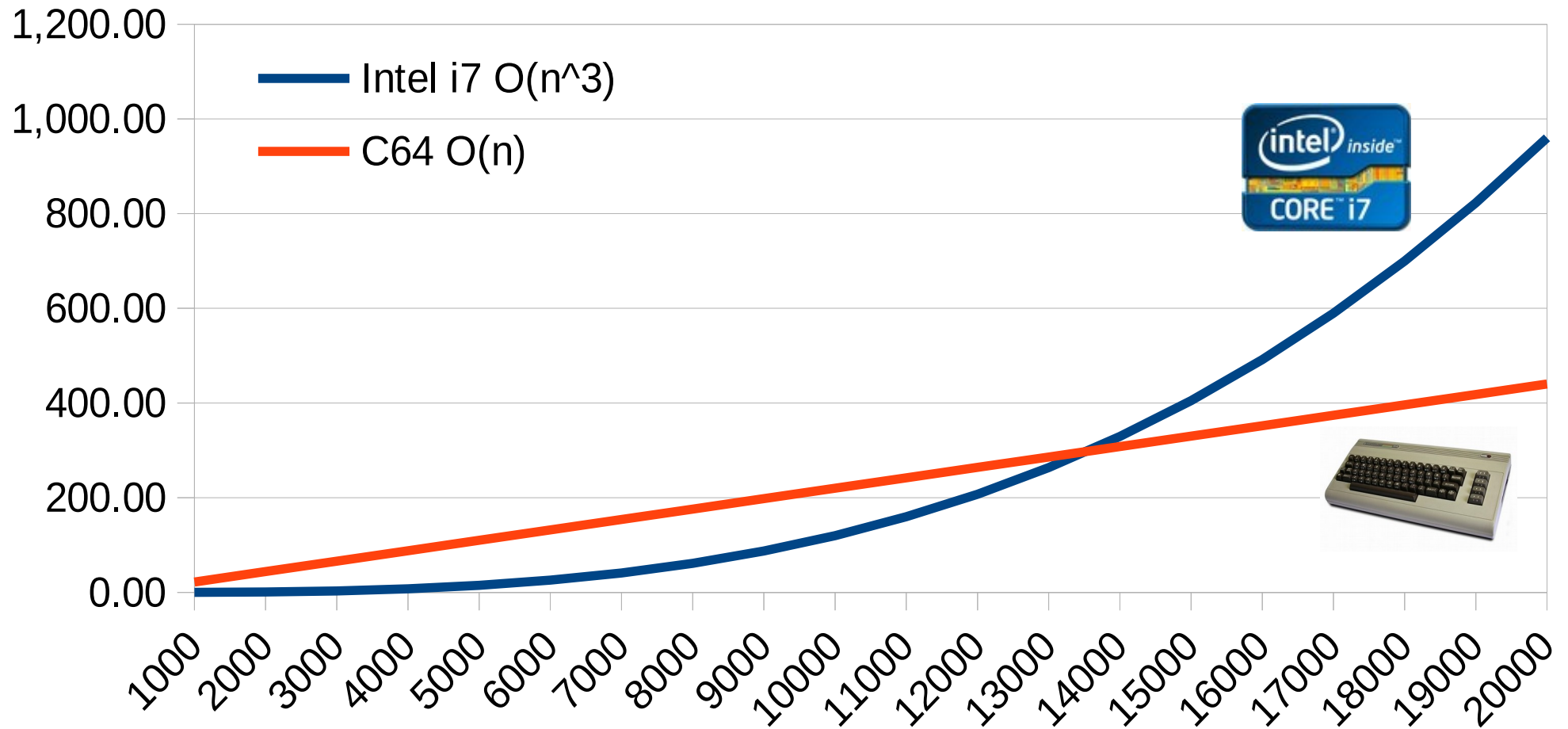
V[]	3	-5	10	2	-3	1	4	-8	7	-6	-1
S[]	3	-2	10	12	9	10	14	6	13	7	6

↑                      ↑

```
integer indiceInizio(real V[1..n], real S[1..n], integer imax)
  integer i ← imax;
  while ( S[i] ≠ V[i] ) do
    i ← i - 1;
  endwhile
  return i;
```

# Sottovettore di somma massima

Tempi di esecuzione in secondi



# Problema dello Zaino (*Knapsack problem*)

# Problema dello zaino

- Abbiamo un insieme  $X = \{1, 2, \dots, n\}$  di  $n$  oggetti
- L'oggetto  $i$ -esimo ha peso  $p[i]$  e valore  $v[i]$
- Disponiamo un contenitore in grado di trasportare al massimo un peso  $P$
- Vogliamo determinare un sottoinsieme  $Y$  di  $X$  tale che
  - Il peso complessivo degli oggetti in  $Y$  sia  $\leq P$
  - Il valore complessivo degli oggetti in  $Y$  sia il massimo possibile, tra tutti gli insiemi di oggetti che possiamo inserire nel contenitore



# Definizione formale

- Vogliamo determinare un sottoinsieme  $Y$  di  $X$  di oggetti tale che:

$$\sum_{x \in Y} p[x] \leq P$$

e tale da massimizzare il valore complessivo:

$$\sum_{x \in Y} v[x]$$

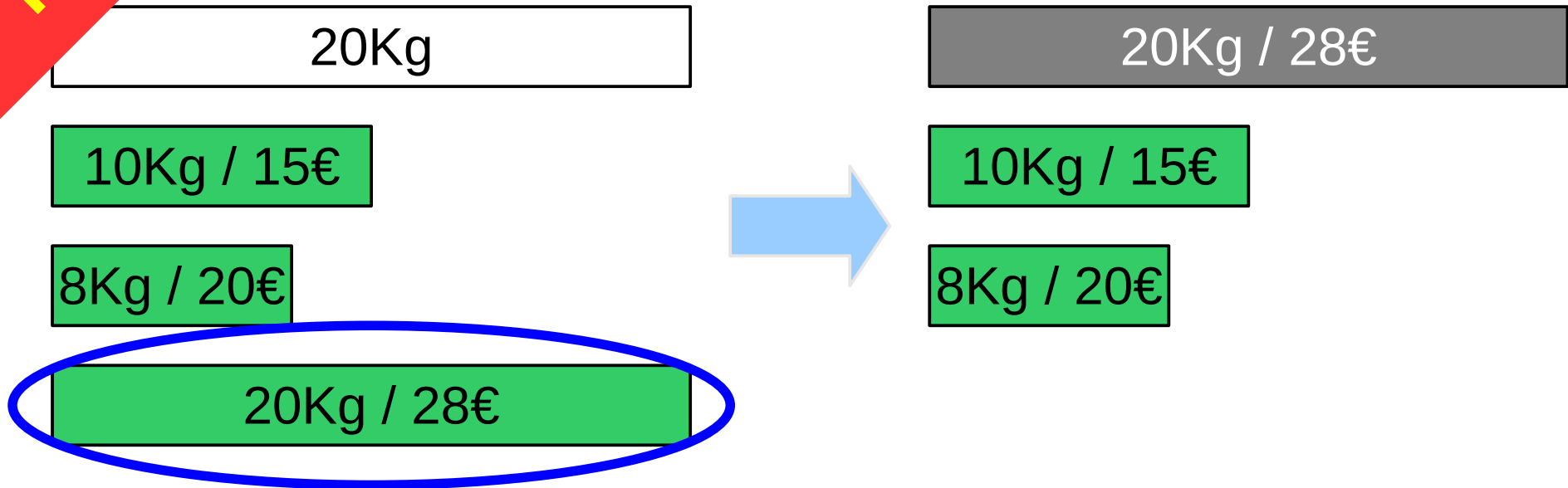
Non  
funziona

# Approccio greedy #1

- Ad ogni passo, scelgo tra gli oggetti non ancora nello zaino quello di **valore** massimo, tra tutti quelli che hanno un peso minore o uguale alla capacità residua dello zaino
  - Questo algoritmo **non fornisce sempre la soluzione ottima**

Non  
funziona

## Esempio



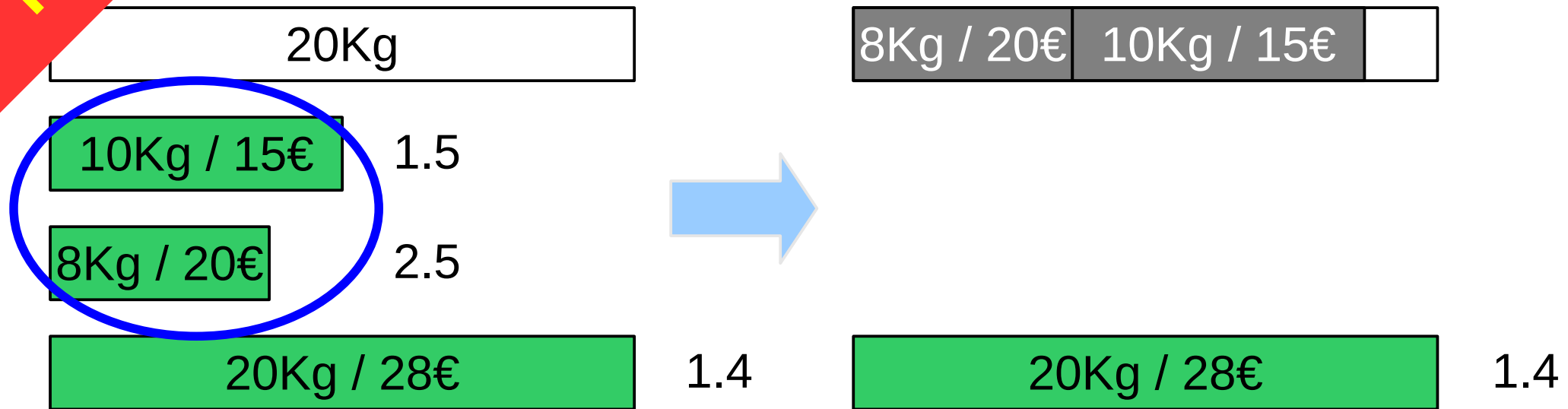
- La soluzione calcolata in questo esempio non è la soluzione ottima!

## Approccio greedy #2

- Ad ogni passo, scelgo tra gli oggetti non ancora nello zaino quello di **valore specifico** massimo, tra tutti quelli che hanno un peso minore o uguale alla capacità residua dello zaino
  - Il **valore specifico** è definito come il valore di un oggetto diviso il suo peso
  - Anche questo approccio non fornisce sempre la soluzione ottima

Non funziona

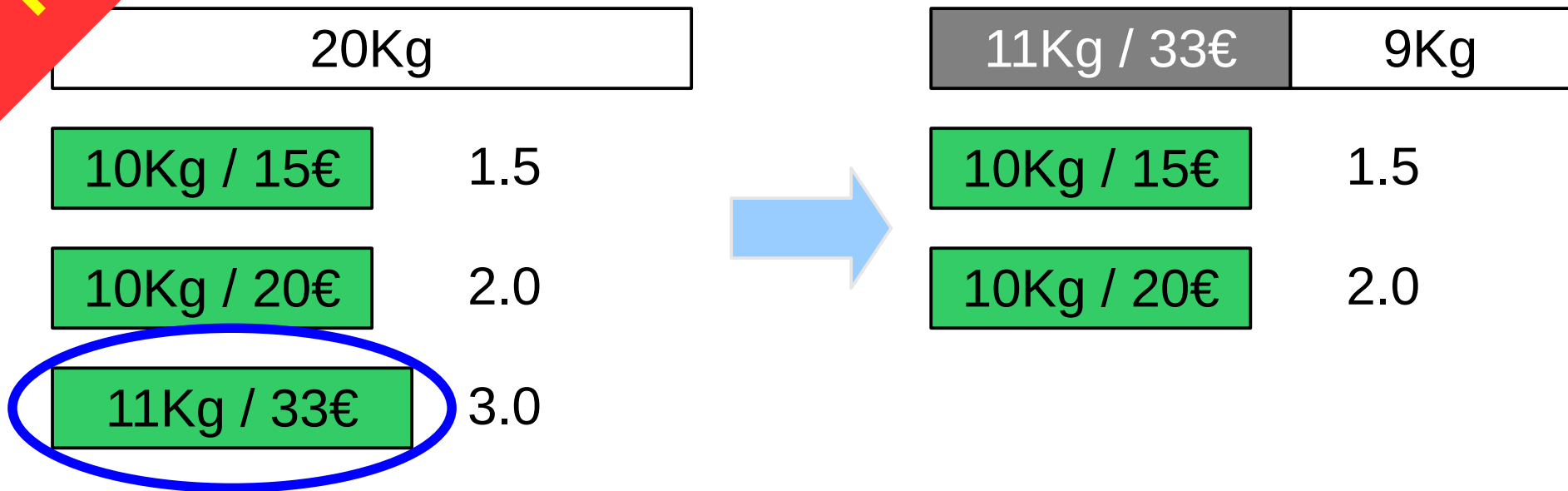
## Esempio



- In questo esempio, l'algoritmo greedy #2 calcola la soluzione ottima, ma...

Non funziona

## Esempio

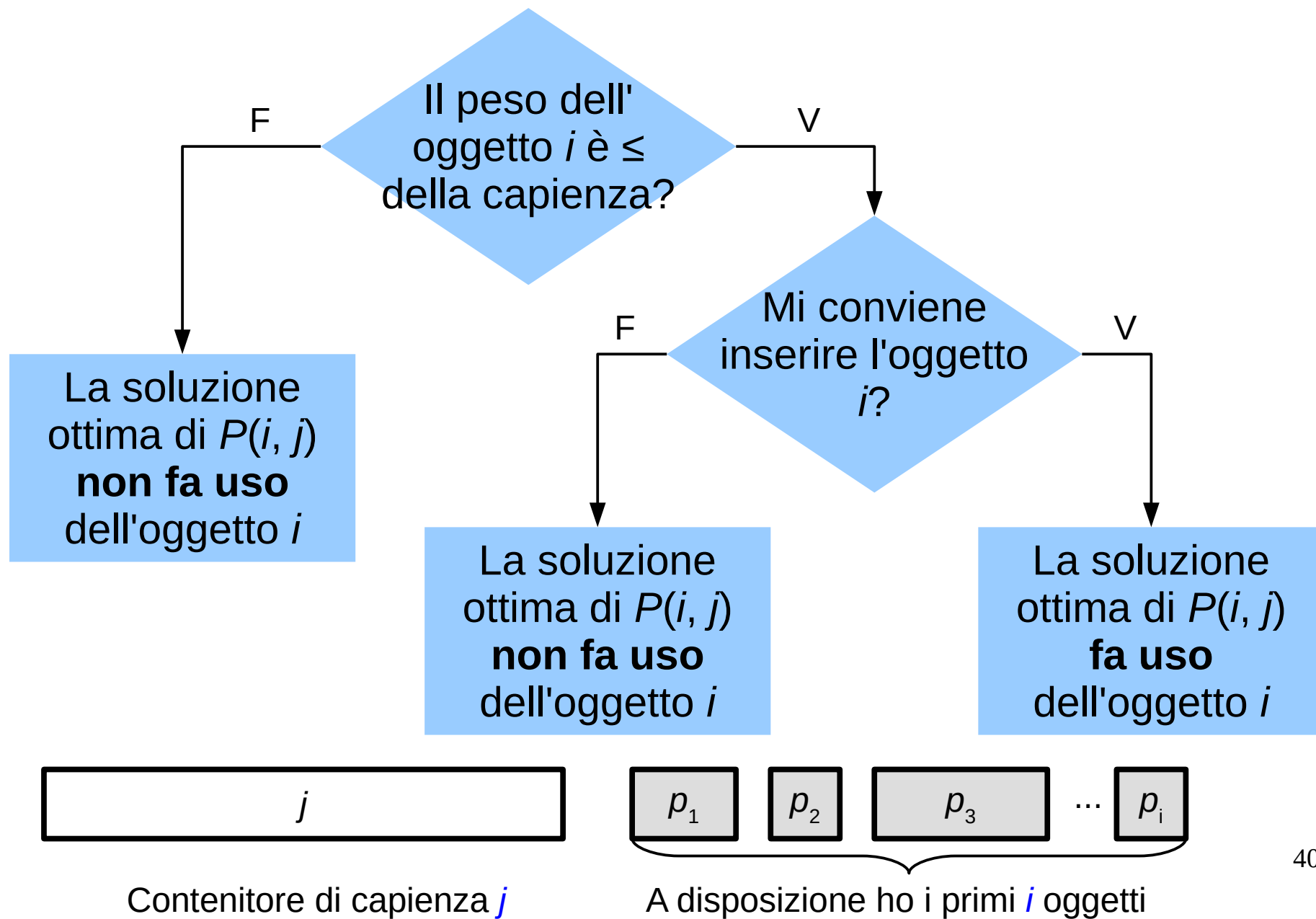


- ...in questo altro esempio anche l'algoritmo greedy #2 non produce la soluzione ottima

# Soluzione ottima basata sulla Programmazione Dinamica

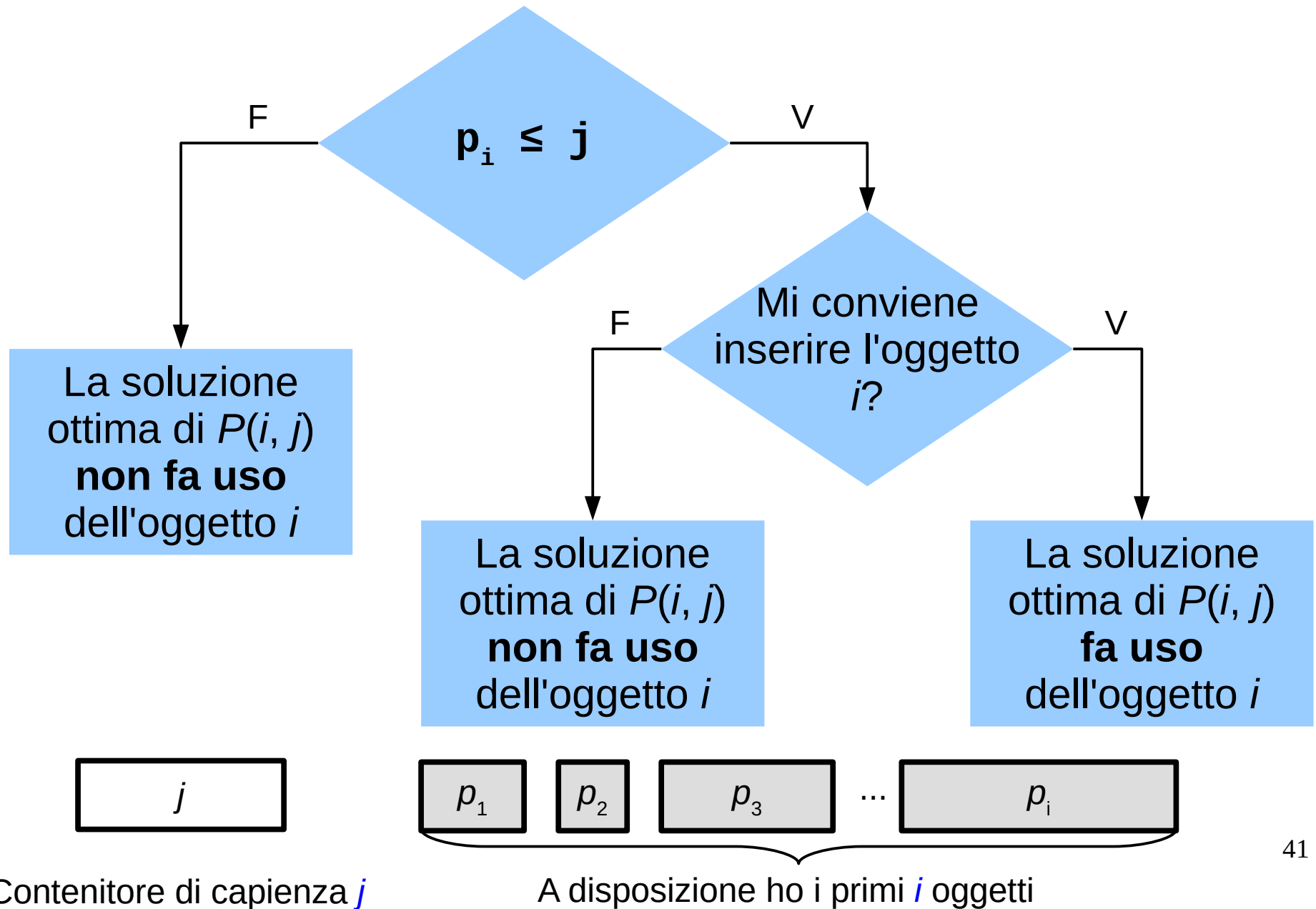
- NB: l'algoritmo di programmazione dinamica richiede che i pesi siano **numeri interi**
- Definizione dei sottoproblemi  $P(i, j)$ 
  - “Riempire uno zaino di capienza  $j$ , utilizzando un opportuno sottoinsieme dei primi  $i$  oggetti, massimizzando il valore degli oggetti usati”
- Definizione delle soluzioni  $V[i, j]$ 
  - $V[i, j]$  è il **massimo valore** ottenibile da un sottoinsieme degli oggetti  $\{1, 2, \dots, i\}$  in uno zaino che ha capacità  $j$
  - $i = 1, 2, \dots, n$
  - $j = 0, 1, \dots, P$

# Schema di soluzione

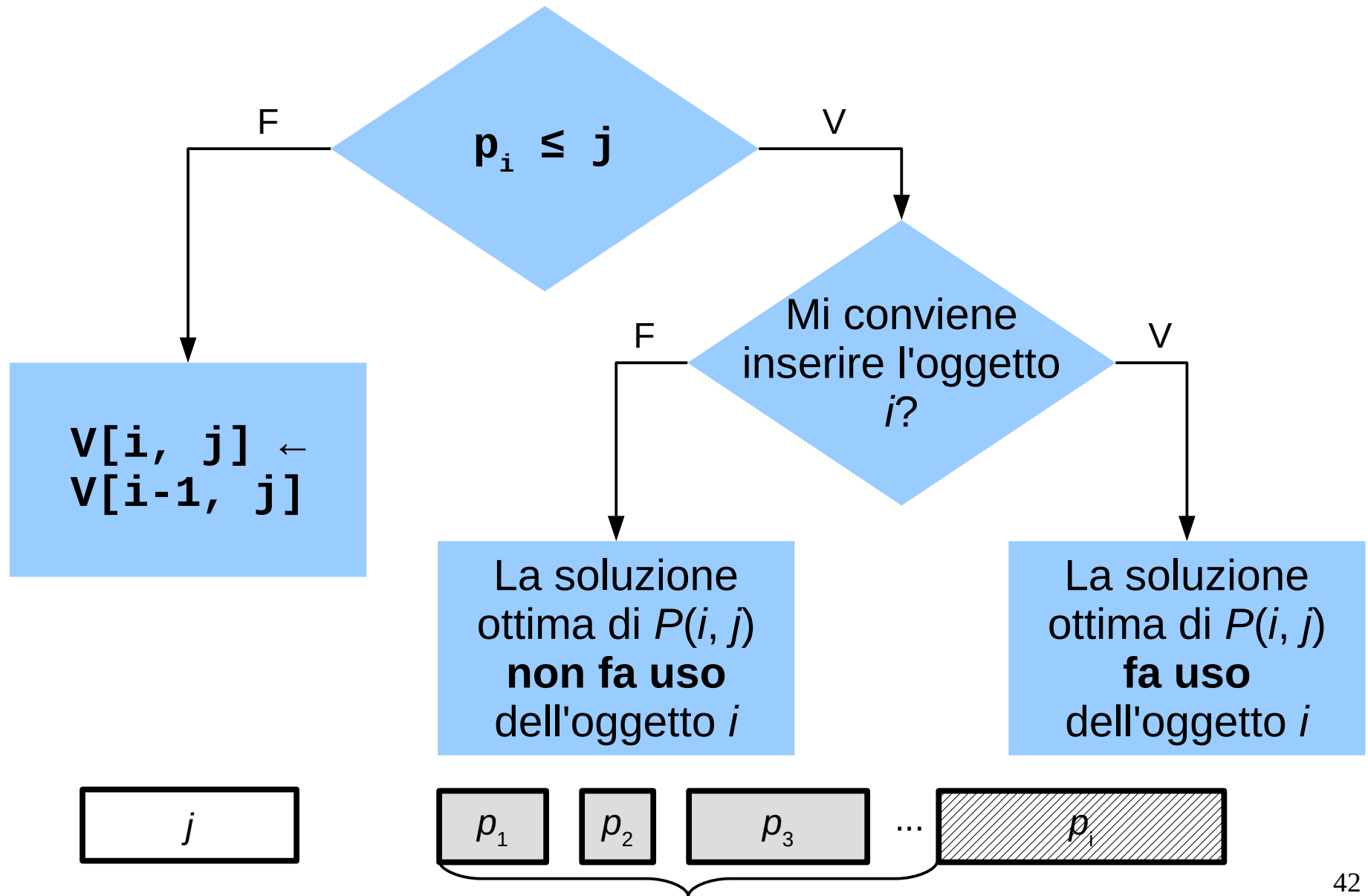




# Schema di soluzione

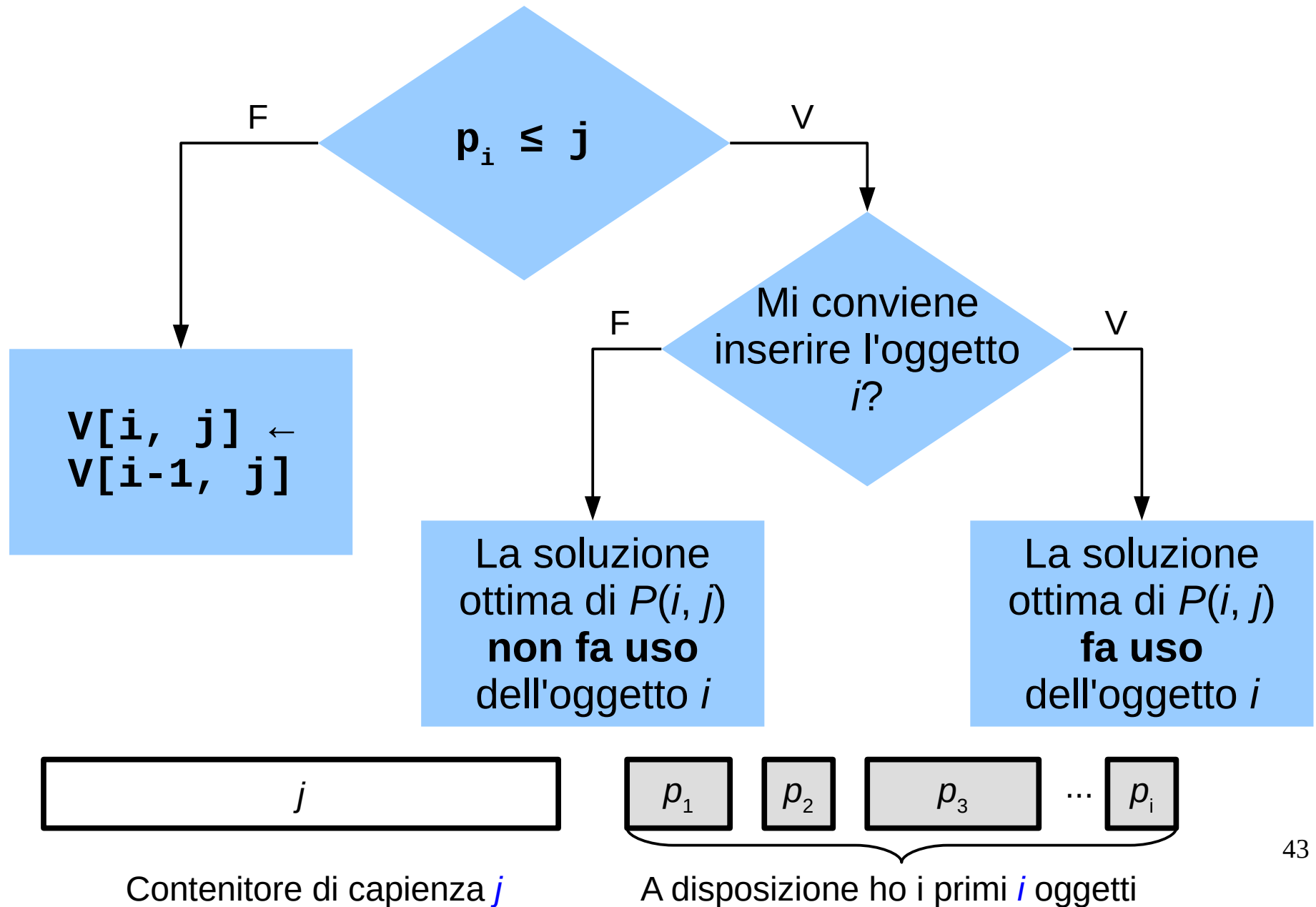


# Schema di soluzione

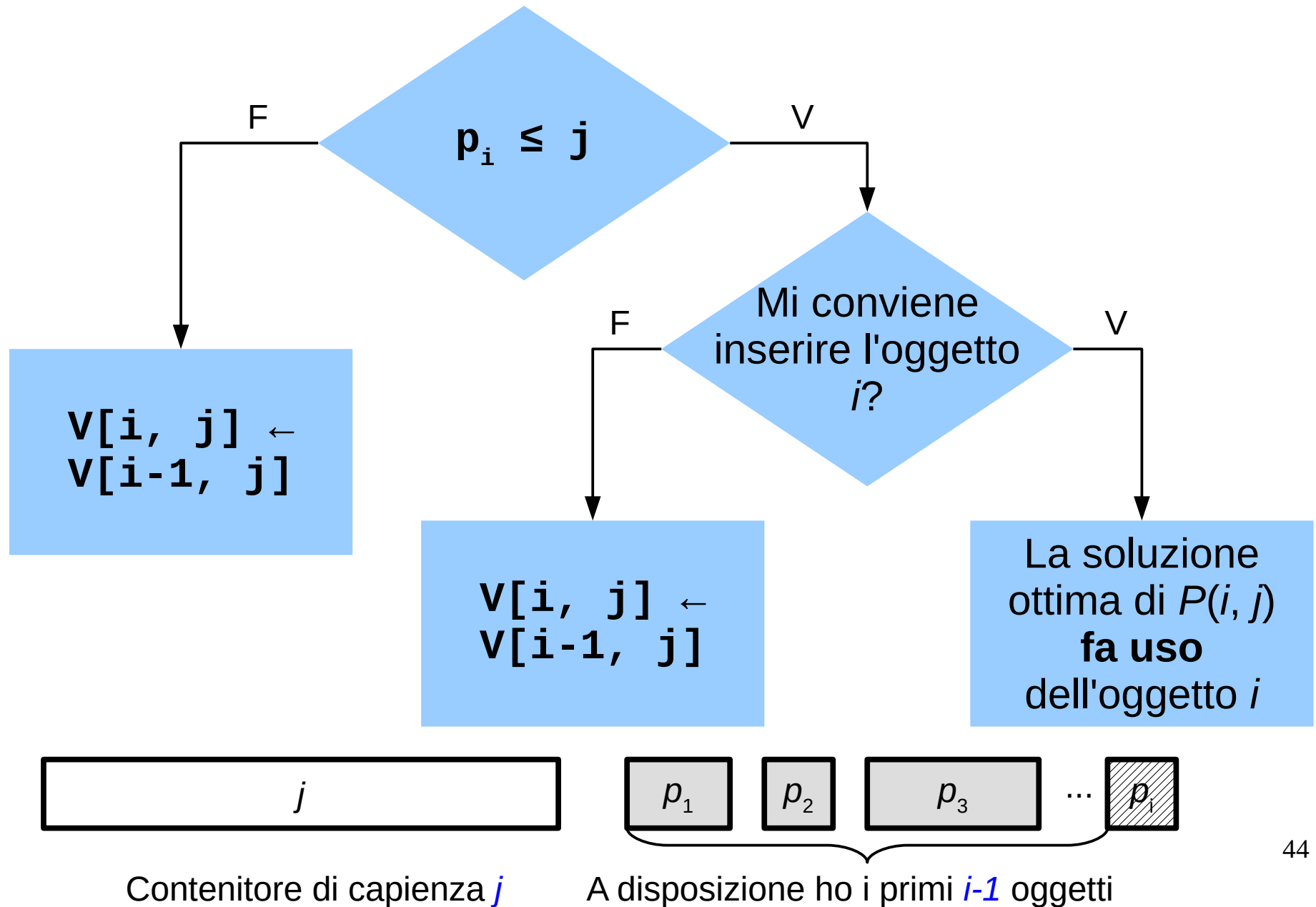


Contenitore di capienza  $j$  A disposizione ho i primi  $i - 1$  oggetti

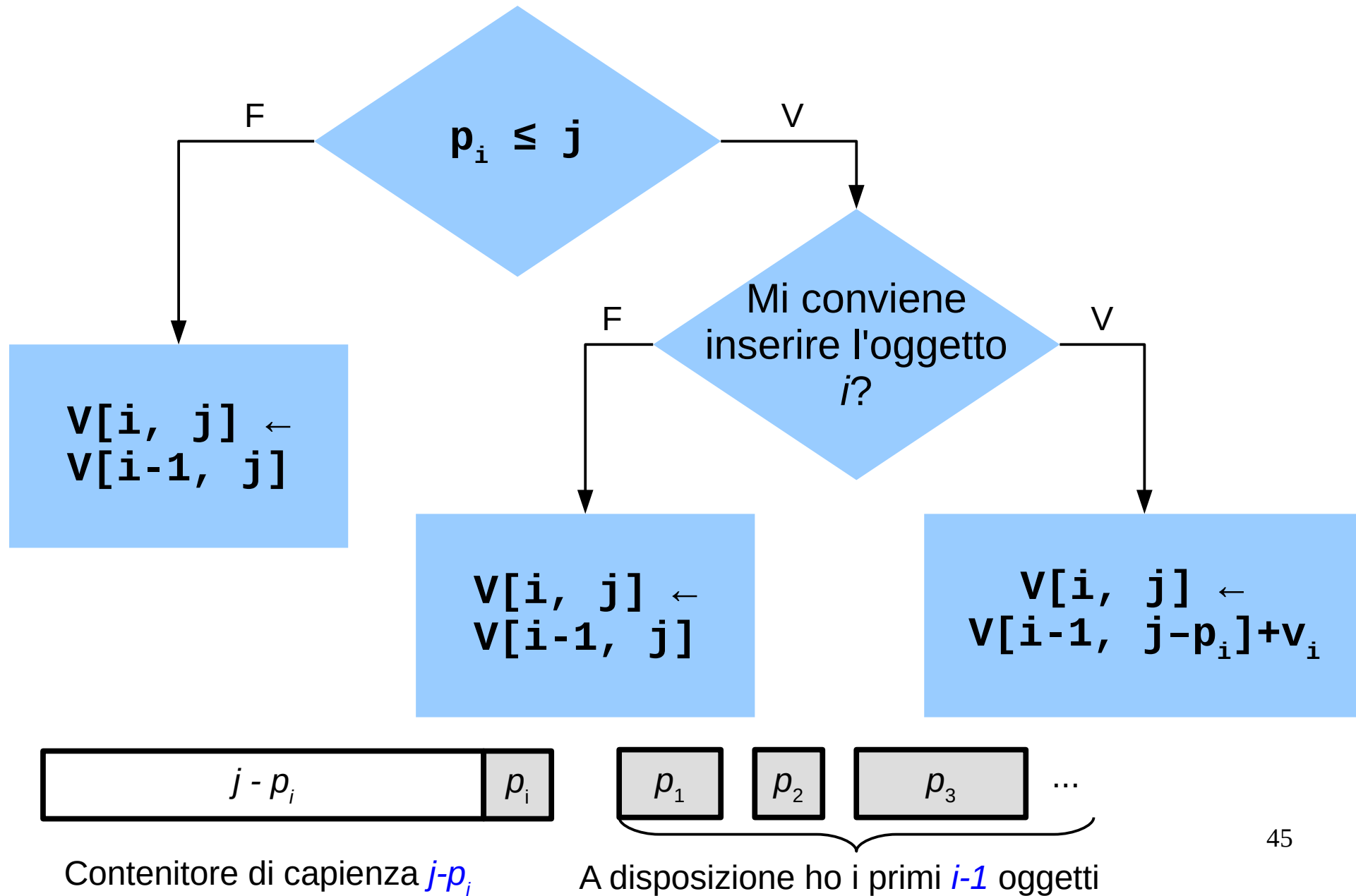
# Schema di soluzione



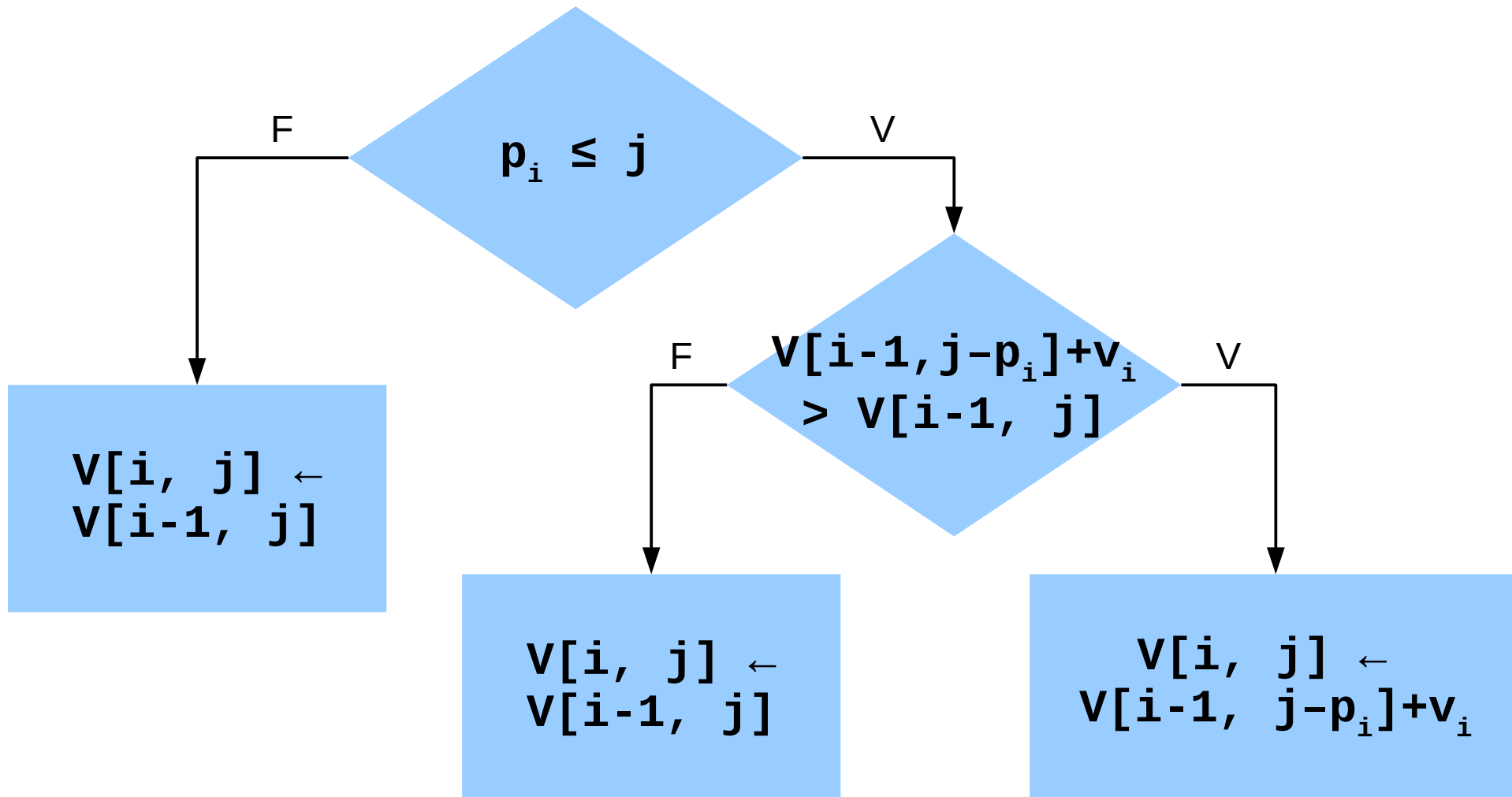
# Schema di soluzione



# Schema di soluzione



# Schema di soluzione



# Calcolo delle soluzioni: caso generale

- Se  $j < p[i]$  significa che l' $i$ -esimo oggetto è troppo pesante per essere contenuto nello zaino. Quindi  $V[i, j] = V[i - 1, j]$
- Se  $j \geq p[i]$  possiamo scegliere se
  - Inserire l'oggetto  $i$ -esimo nello zaino. Il valore massimo ottenibile in questo caso è  $V[i - 1, j - p[i]] + v[i]$
  - Non inserire l'oggetto  $i$ -esimo nello zaino. Il massimo valore ottenibile in questo caso è  $V[i - 1, j]$
  - Scegliamo l'alternativa che massimizza il valore:  
 $V[i, j] = \max\{ V[i - 1, j], V[i - 1, j - p[i]] + v[i] \}$

$V[i, j]$  è il massimo valore ottenibile da un sottoinsieme degli oggetti  $\{1, 2, \dots, i\}$  in uno zaino che ha capacità massima  $j$

# Calcolo delle soluzioni: casi base

- Primo caso base: zaino di capienza zero
  - $V[i, 0] = 0$  per ogni  $i = 1..n$ 
    - Se la capacità dello zaino è zero, il massimo valore ottenibile è zero (nessun oggetto)
- Secondo caso base: ho a disposizione solo l'oggetto 1
  - $V[1, j] = v[1]$  se  $j \geq p[1]$ 
    - C'è abbastanza spazio per l'oggetto numero 1
  - $V[1, j] = 0$  se  $j < p[1]$ 
    - Non c'è abbastanza spazio per l'oggetto numero 1

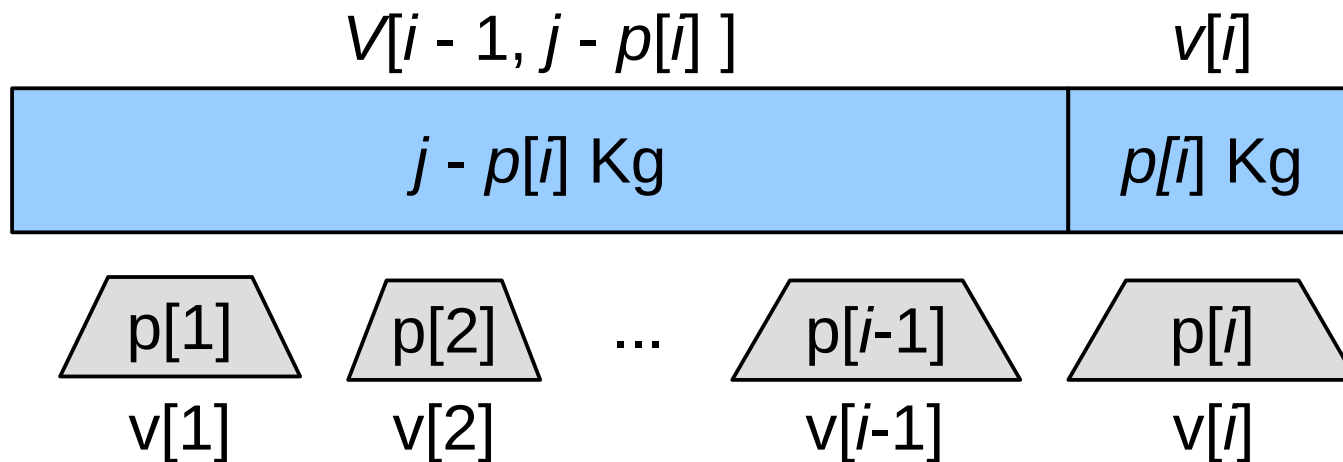
$V[i, j]$  è il massimo valore ottenibile da un sottoinsieme degli oggetti  $\{1, 2, \dots, i\}$  in uno zaino che ha capacità massima  $j$



# Soluzione ottima basata sulla Programmazione Dinamica

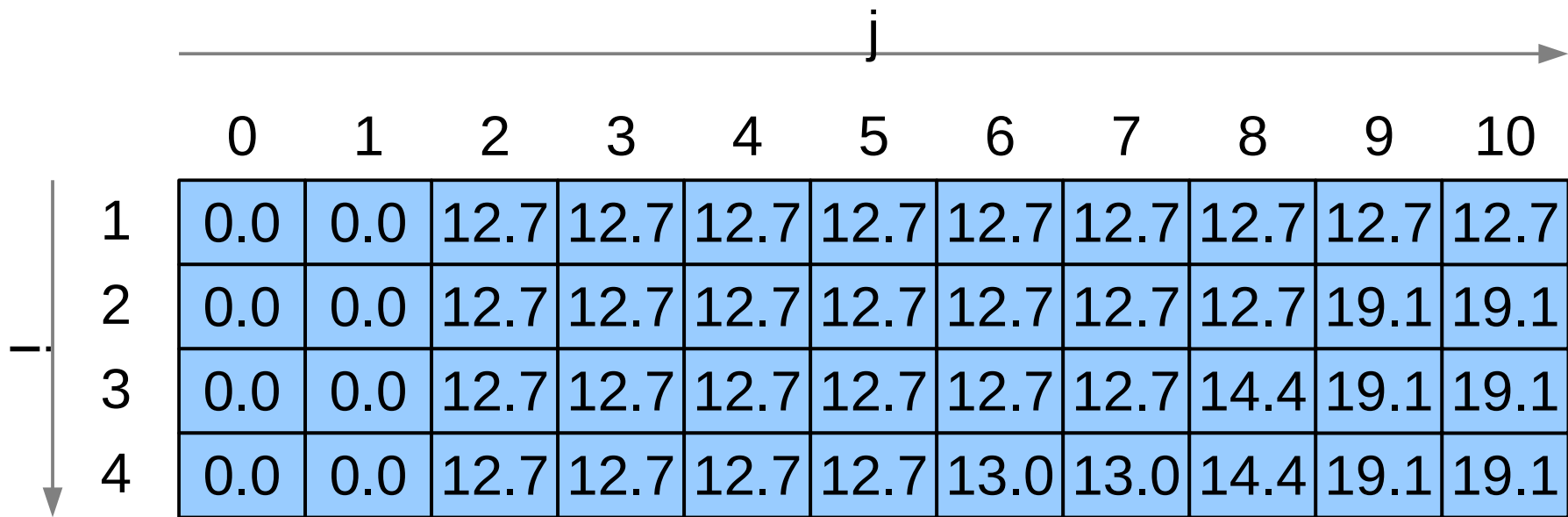
- Riassumendo

$$V[i, j] = \begin{cases} V[i-1, j] & \text{se } j < p[i] \\ \max \{ V[i-1, j], V[i-1, j - p[i]] + v[i] \} & \text{se } j \geq p[i] \end{cases}$$



# Tabella di programmazione dinamica / matrice $V$

$$p = [ 2, 7, 6, 4 ]$$
$$v = [ 12.7, 6.4, 1.7, 0.3 ]$$



	0	1	2	3	4	5	6	7	8	9	10
1	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	12.7	12.7	12.7
2	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	12.7	19.1	19.1
3	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	14.4	19.1	19.1
4	0.0	0.0	12.7	12.7	12.7	12.7	13.0	13.0	14.4	19.1	19.1

# Tabella di programmazione dinamica / matrice V

$$p = [2, 7, 6, 4]$$

$$v = [12.7, 6.4, 1.7, 0.3]$$

	0	1	2	3	4	5	6	7	8	9	10
1	0.0	0.0	12.7	12.7	12.7		12.7	12.7	12.7	12.7	12.7
2	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	12.7	19.1	19.1
3	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	14.4	19.1	19.1
4	0.0	0.0	12.7	12.7	12.7	12.7	13.0	13.0	14.4	19.1	19.1

$$V[3,8] = \max \{ V[2,8], V[2,8-6] + 1.7 \}$$

$$= \max \{ 12.7, 14.4 \}$$

# Stampare la soluzione

- Il valore della soluzione ottima del problema di partenza è  $V[n, P]$
- Come facciamo a sapere **quali oggetti** fanno parte della soluzione ottima?
  - Usiamo una tabella ausiliaria booleana  **$use[i, j]$**  che ha le stesse dimensioni di  $V[i, j]$
  - $use[i, j] = true$  se e solo se l'oggetto  $i$ -esimo fa parte della soluzione ottima del problema  $P(i, j)$  che ha valore  $V[i, j]$

# Tabella di programmazione dinamica / stampa soluzione ottima

```
integer j ← P;  
integer i ← n;  
while ( i > 0 ) do  
    if ( use[i,j] = true ) then  
        stampa "Seleziono oggetto ", i  
        j ← j - p[i];  
    endif  
    i ← i - 1;  
endwhile
```

$p = [ 2, 7, 6, 4 ]$   
 $v = [ 12.7, 6.4, 1.7, 0.3 ]$

	0	1	2	3	4	5	6	7	8	9	10
1	F	F	T	T	T	T	T	T	T	T	T
2	F	F	F	F	F	F	F	F	F	T	T
3	F	F	F	F	F	F	F	F	T	F	F
4	F	F	F	F	F	F	T	T	F	F	F

# Conclusioni

- La programmazione dinamica è una tecnica potente
- Va però applicata (dove applicabile) con **disciplina**
  - Identificare i sotto-problemi
  - Definire le soluzioni dei sotto-problemi
  - Calcolare le soluzioni nei casi semplici
  - Calcolare le soluzioni nel caso generale

# **Materiale extra**

# *Seam Carving*



# Seam Carving

- Algoritmo per ridimensionare immagini in modo “intelligente”
  - Shai Avidan and Ariel Shamir. 2007. *Seam carving for content-aware image resizing*. In ACM SIGGRAPH 2007 (SIGGRAPH '07). ACM, New York, NY, USA, <http://doi.acm.org/10.1145/1275808.1276390>



[https://en.wikipedia.org/wiki/Seam\\_carving](https://en.wikipedia.org/wiki/Seam_carving)





Scaling



Cropping

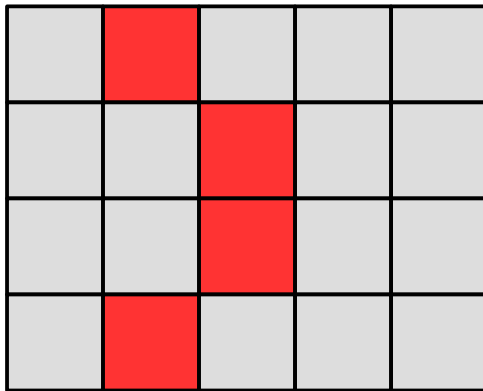


Seam Carving

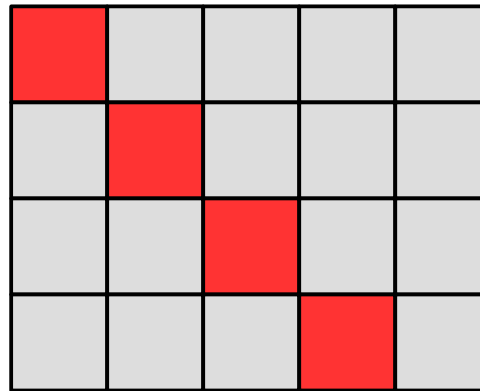


# Seam Carving

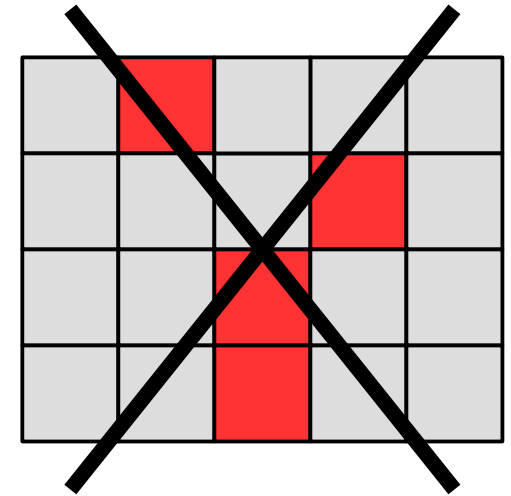
- L'immagine viene ridimensionata togliendo *cuciture*
- Una cucitura (*seam*) è un cammino composto da pixel adiacenti di “minima importanza”
  - Se l'immagine ha  $M$  righe e  $N$  colonne, una cucitura (verticale) è una sequenza di  $M$  pixel, uno per ogni riga
  - Due pixel sono adiacenti se hanno un lato o uno spigolo in comune



OK



OK



No: pixel non adiacenti

# Seam Carving

- Assegna ad ogni pixel  $(i, j)$  un peso  $E[i, j] \in [0, 1]$  che indica quanto il pixel è “importante”
  - Es.: quanto un pixel è diverso da quelli adiacenti
  - 0 = non importante,  
1 = molto importante
- Determina una cucitura verticale di peso minimo
- Rimuovi i pixel della cucitura, ottenendo una immagine  $M \times (N - 1)$
- Ripeti il procedimento fino ad ottenere la larghezza desiderata.

0.1	0.0	0.2	0.9	0.8
0.9	0.2	0.8	0.4	0.7
0.8	0.8	0.1	0.7	0.8
0.1	0.0	0.6	0.5	0.7

0.1	0.0	0.2	0.9	0.8
0.9	0.2	0.8	0.4	0.7
0.8	0.8	0.1	0.7	0.8
0.1	0.0	0.6	0.5	0.7

0.1	0.2	0.9	0.8
0.9	0.8	0.4	0.7
0.8	0.8	0.7	0.8
0.1	0.6	0.5	0.7

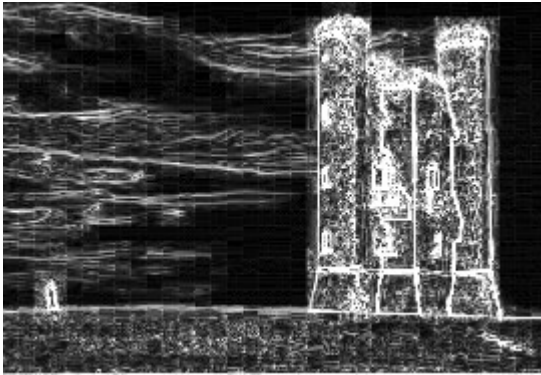


# Esempio

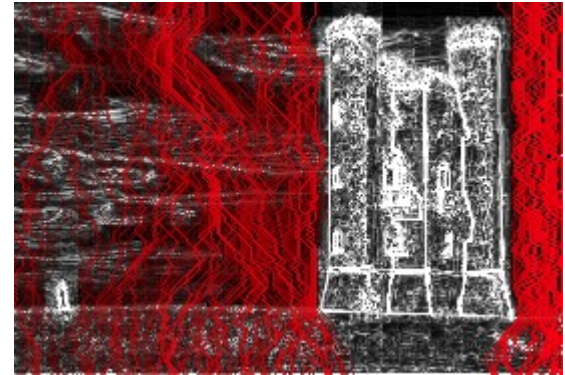
Immagine originale



Pesi (nero=0, bianco=1)



Alcune cuciture di peso minimo



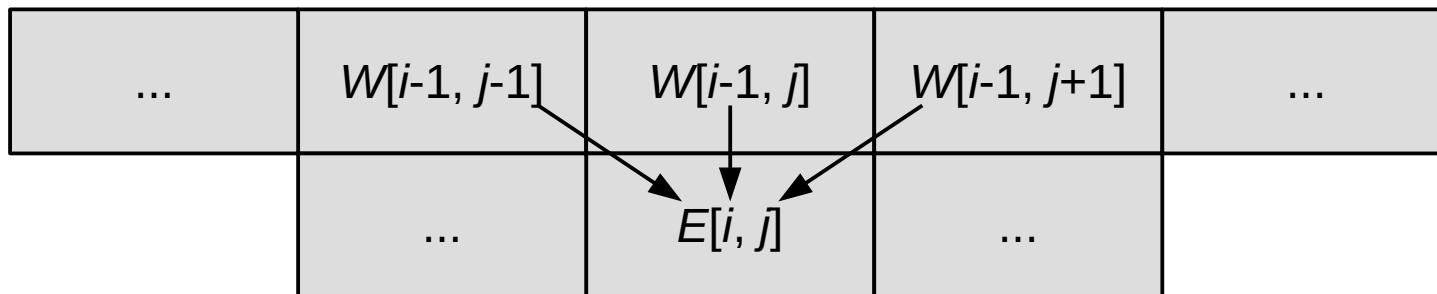
[https://en.wikipedia.org/wiki/Seam\\_carving](https://en.wikipedia.org/wiki/Seam_carving)

# Determinare le cuciture di peso minimo con la programmazione dinamica

- Definizione dei sottoproblemi  $P(i, j)$ 
  - Determinare una cucitura di peso minimo che termina nel pixel di coordinate  $(i, j)$
- Definizione delle soluzioni  $W[i, j]$ 
  - Minimo peso tra tutte le possibili cuciture che terminano nel pixel di coordinate  $(i, j)$
- Calcolo delle soluzioni  $W[i, j]$ 
  - Vedi slide successiva
- Definizione della soluzione del problema originario
  - La cucitura di peso minimo avrà peso pari al minimo tra  $\{ W[M, 1], \dots W[M, N] \}$

# Calcolo di $W[i, j]$

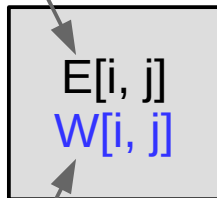
- Casi base ( $i = 1$ ):
  - $W[1, j] = E[1, j]$  per ogni  $j = 1, \dots, M$
- Caso generale ( $i > 1$ ):
  - Se  $j = 1$   
 $W[i, j] = E[i, j] + \min \{ W[i - 1, j], W[i - 1, j + 1] \}$
  - Se  $1 < j < N$   
 $W[i, j] = E[i, j] + \min \{ W[i - 1, j - 1], W[i - 1, j], W[i - 1, j + 1] \}$
  - Se  $j = N$   
 $W[i, j] = E[i, j] + \min \{ W[i - 1, j - 1], W[i - 1, j] \}$





# Esempio

Energia del pixel  $(i, j)$

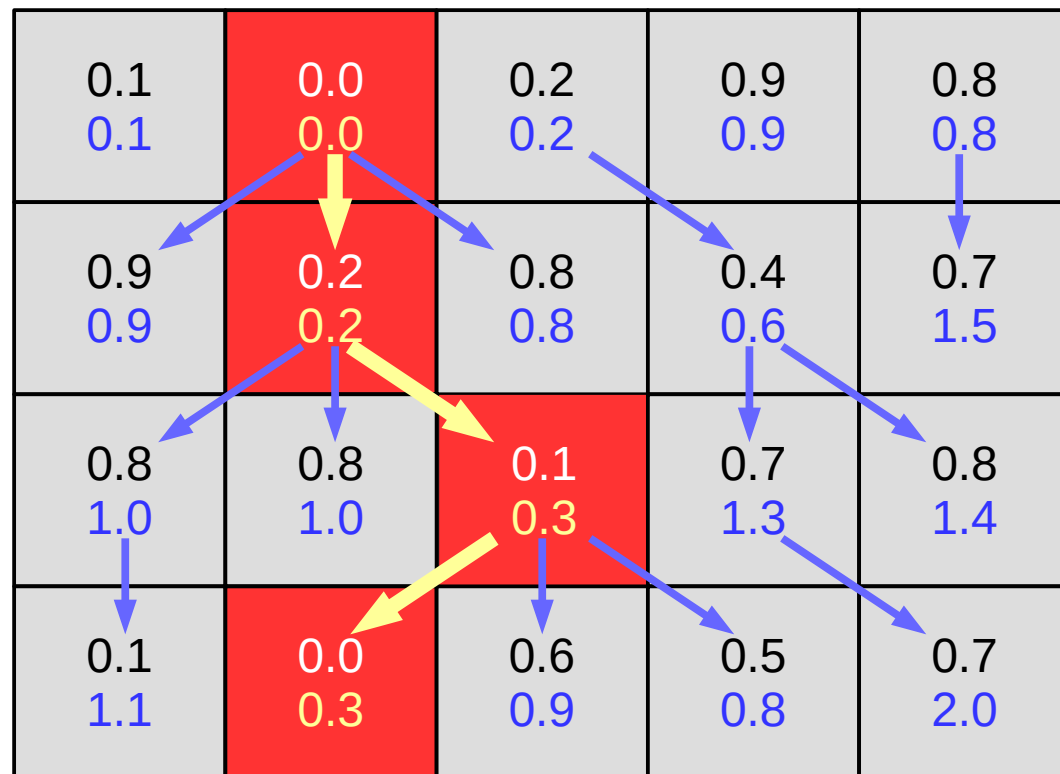


Minimo peso tra tutte le cuciture che iniziano sulla prima riga e terminano nel pixel  $(i, j)$

0.1 0.1	0.0 0.0	0.2 0.2	0.9 0.9	0.8 0.8
0.9 0.9	0.2 0.2	0.8 0.8	0.4 0.6	0.7 1.5
0.8 1.0	0.8 1.0	0.1 0.3	0.7 1.3	0.8 1.4
0.1 1.1	0.0 0.3	0.6 0.9	0.5 0.8	0.7 2.0

# Esempio

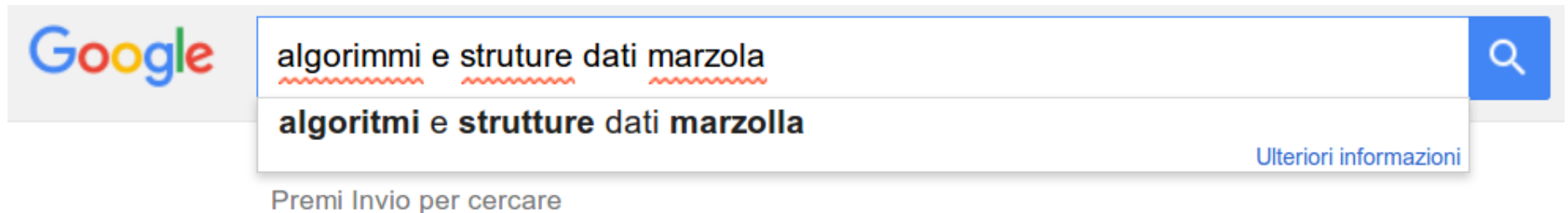
$E[i, j]$ $W[i, j]$
------------------------



# Distanza di Levenshtein

# Introduzione

- I correttori ortografici sono in grado di suggerire le parole corrette più simili a quello che abbiamo digitato
- Come si fa a decidere quanto “simili” sono due stringhe (=sequenze di caratteri) ?



# Distanza di Levenshtein

- Basata sul concetto di “*edit distance*”
  - Numero di operazioni di “editing” che sono necessarie per trasformare una stringa  $S$  in una nuova stringa  $T$
- Trasformazioni ammesse
  - Lasciare immutato il carattere corrente (costo 0)
  - Cancellare un carattere (costo 1)
  - Inserire un carattere (costo 1)
  - Sostituire il carattere corrente con uno diverso (costo 1)
- Dopo ciascuna operazione ci si sposta sul carattere successivo
  - Si inizia dal primo carattere di  $S$

# Esempio

- Trasformiamo “LIBRO” in “ALBERO”
  - Una possibilità è cancellare tutti i caratteri di “LIBRO” e inserire tutti quelli di “ALBERO”
  - Costo totale: 5 cancellazioni + 6 inserimenti = 11

<u>L</u> IBRO	→	<u>I</u> BRO	cancellazione L
<u>I</u> BRO	→	<u>B</u> RO	cancellazione I
<u>B</u> RO	→	<u>R</u> O	cancellazione B
<u>R</u> O	→	<u>O</u>	cancellazione R
<u>O</u>	→	<u>  </u>	cancellazione O
<u>  </u>	→	A <u>  </u>	inserimento A
A <u>  </u>	→	AL <u>  </u>	inserimento L
AL <u>  </u>	→	ALB <u>  </u>	inserimento B
ALB <u>  </u>	→	ALBE <u>  </u>	inserimento E
ALBE <u>  </u>	→	ALBER <u>  </u>	inserimento R
ALBER <u>  </u>	→	ALBERO <u>  </u>	inserimento O

# Esempio

- Possiamo ottenere lo stesso risultato con un costo inferiore
  - 2 inserimento + 1 cancellazione = 3

<u>L</u> IBRO	→	A <u>L</u> IBRO	inserisco A
AL <u>I</u> BRO	→	AL <u>I</u> BRO	lascio immutato
AL <u>I</u> BRO	→	AL <u>B</u> RO	cancello I
AL <u>B</u> RO	→	AL <u>B</u> RO	lascio immutato
AL <u>B</u> RO	→	AL <u>B</u> ER <u>O</u>	inserisco E
AL <u>B</u> ER <u>O</u>	→	AL <u>B</u> ER <u>O</u>	lascio immutato
AL <u>B</u> ER <u>O</u>	→	AL <u>B</u> ER <u>O</u> <u>_</u>	lascio immutato

# Definizione

- Due stringhe  $S[1..n]$  e  $T[1..m]$  di  $n$  ed  $m$  caratteri, rispettivamente
  - Una o entrambe potrebbero anche essere vuote.
- La distanza di Levenshtein tra  $S[1..n]$  e  $T[1..m]$  è il costo minimo tra tutte le sequenze di operazioni di editing che trasformano  $S$  in  $T$
- Alcune definizioni aggiuntive
  - $S[1..i]$  la stringa composta dai primi  $i$  caratteri di  $S$ 
    - se  $i = 0$  è la sottostringa vuota
  - $T[1..j]$  la stringa composta dai primi  $j$  caratteri di  $T$ 
    - se  $j = 0$  è la sottostringa vuota

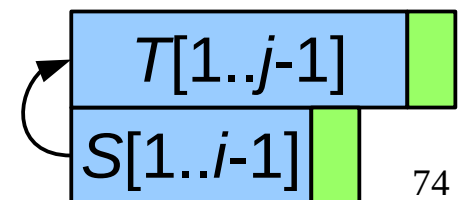
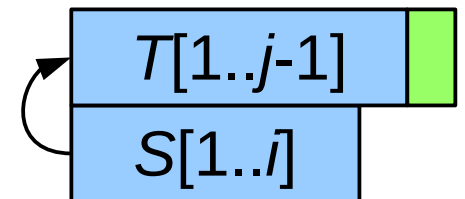
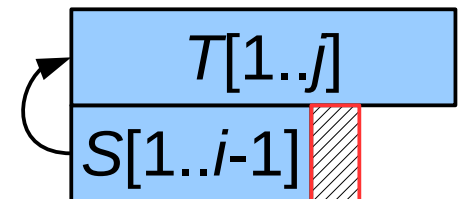
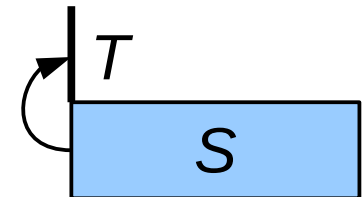
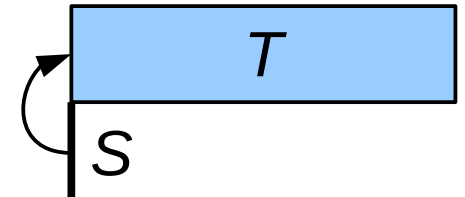


# Determinare la distanza di Levenshtein con la programmazione dinamica

- Definizione dei sottoproblemi  $P(i, j)$ 
  - Determinare il minimo numero di operazioni di editing necessarie per trasformare il prefisso  $S[1..i]$  di  $S$  nel prefisso  $T[1..j]$  di  $T$
- Definizione delle soluzioni  $L[i, j]$ 
  - minimo numero di operazioni di editing necessarie per trasformare il prefisso  $S[1..i]$  di  $S$  nel prefisso  $T[1..j]$  di  $T$
- Calcolo della soluzione del problema originario
  - La distanza di Levenshtein tra  $S[1..n]$  e  $T[1..m]$  è il valore  $L[n, m]$

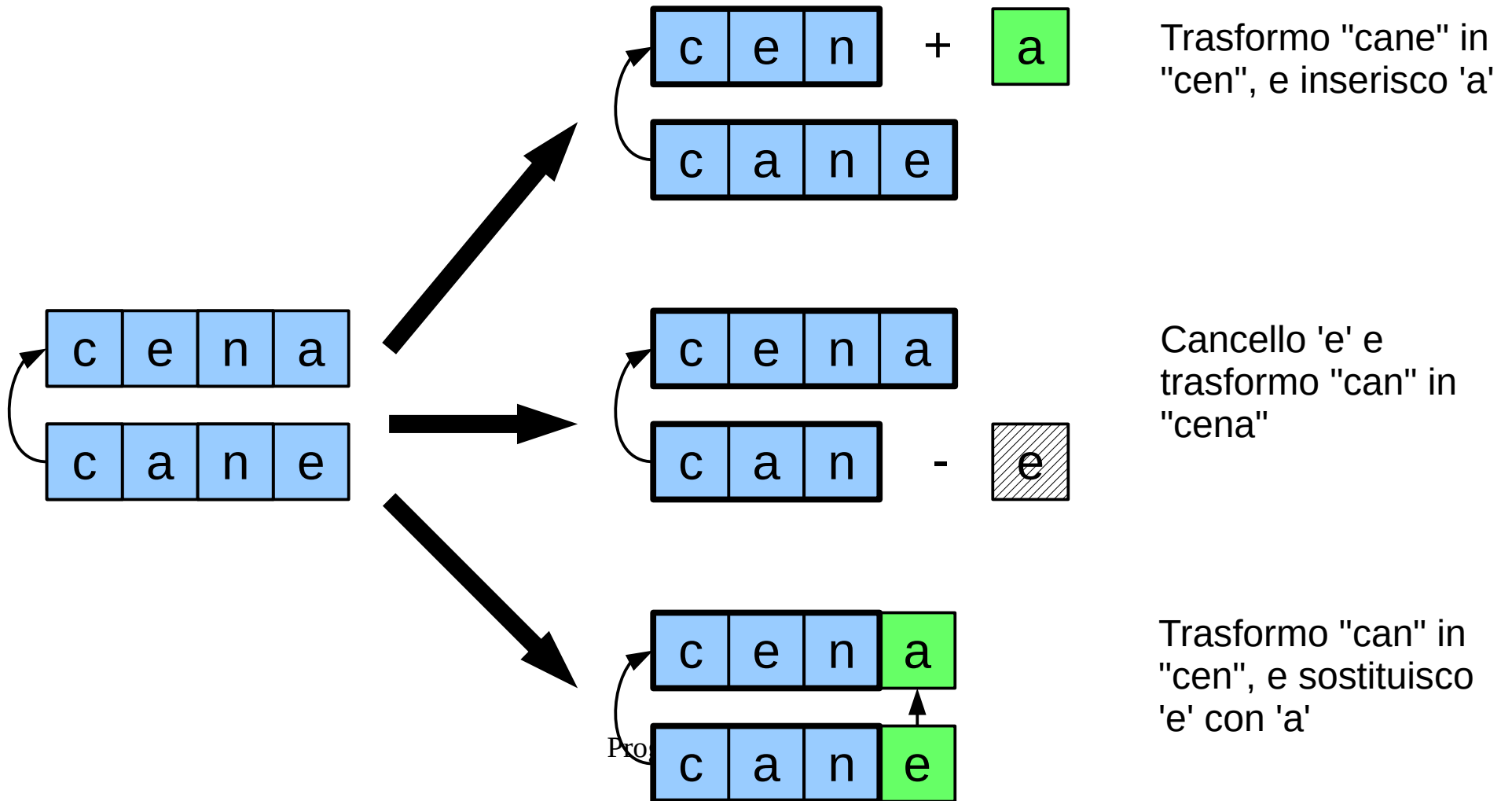
# Calcolo di $L[i, j]$

- Se  $i = 0$  oppure  $j = 0$ 
  - Il costo per trasformare una stringa vuota in una non vuota è dato dalla lunghezza della stringa non vuota (occorre inserire o cancellare tutti i caratteri)
- Se  $i > 0$  e  $j > 0$ , il minimo costo tra:
  - Trasformare  $S[1..i - 1]$  in  $T[1..j]$ , e **cancellare** l'ultimo carattere  $S[i]$  di  $S$
  - Trasformare  $S[1..i]$  in  $T[1..j - 1]$  e **inserire** l'ultimo carattere  $T[j]$  di  $T$
  - Trasformare  $S[1..i - 1]$  in  $T[1..j - 1]$  e **cambiare**  $S[i]$  in  $T[j]$  se diversi



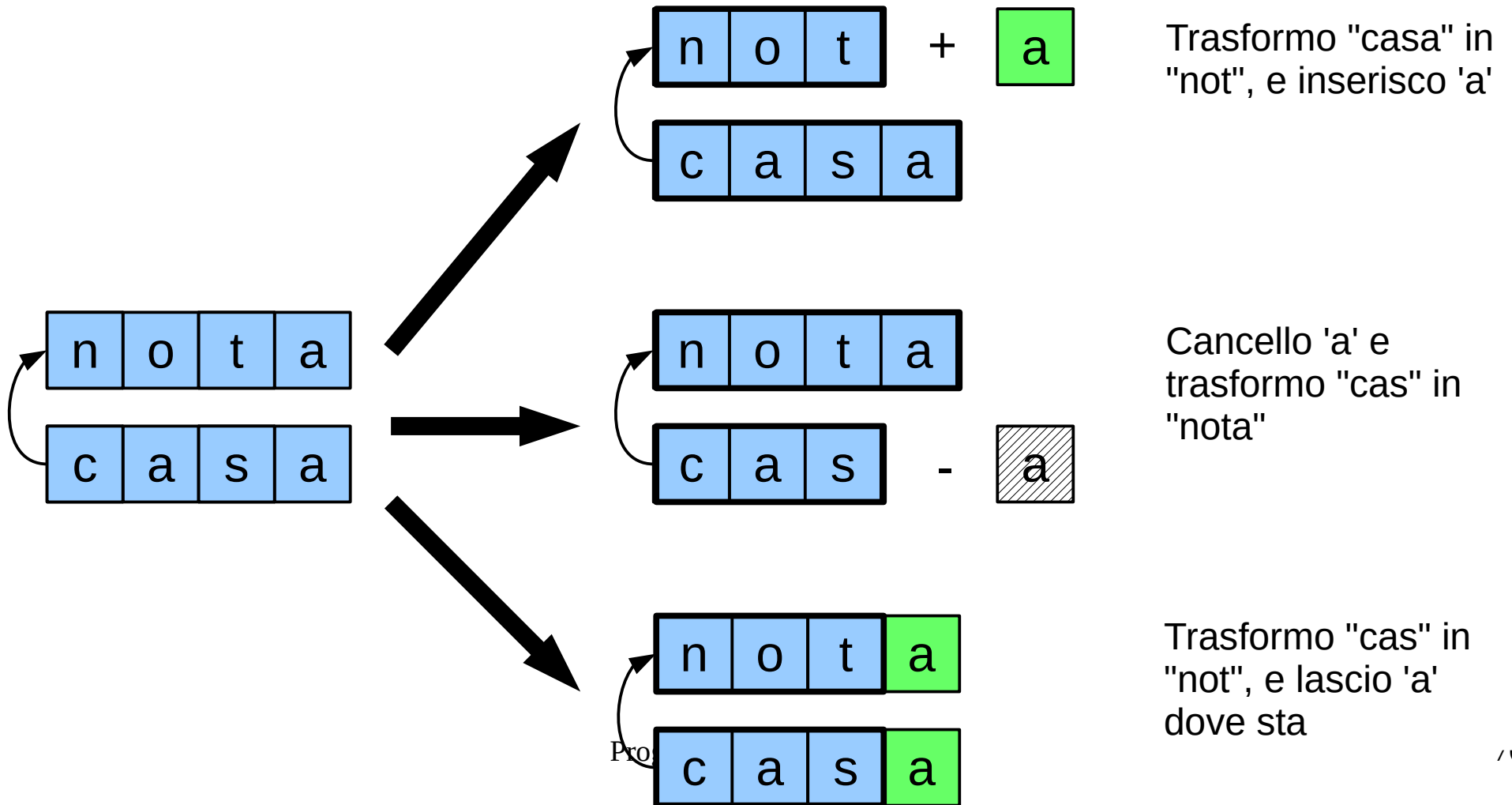
# Esempio

- Trasformare S = "cane" in T = "cena"



# Esempio

- Trasformare  $S = \text{"casa"}$  in  $T = \text{"nota"}$



# Calcolo di $L[i, j]$

- Se  $i = 0$  oppure  $j = 0$

- $L[i, j] = \max \{ i, j \}$

- Altrimenti

- Se  $S[i] = T[j]$

- $L[i, j] = \min \{ L[i-1, j] + 1, L[i, j-1] + 1, L[i-1, j-1] \}$

- Se  $S[i] \neq T[j]$

- $L[i, j] = \min \{ L[i-1, j] + 1, L[i, j-1] + 1, L[i-1, j-1] + 1 \}$

Costo per trasformare  $S[1..i-1]$  in  $T[1..j]$ , e cancellare il carattere  $S[i]$

Costo per trasformare  $S[1..i]$  in  $T[1..j-1]$ , e aggiungere il carattere  $T[j]$

Costo per trasformare  $S[1..i-1]$  in  $T[1..j-1]$ , e lasciare invariato l'ultimo carattere

# Esempio

	“ ”	A	L	B	E	R	O
“ ”	0	1	2	3	4	5	6
L	1	1	1	2	3	4	5
I	2	2	2	2	3	4	5
B	3	3	3	2	3	4	5
R	4	4	4	3	3	3	4
O	5	5	5	4	4	4	3

*Minimo numero di operazioni  
di editing necessarie per  
trasformare LIB in ALBE*

Programmazione Dinamica

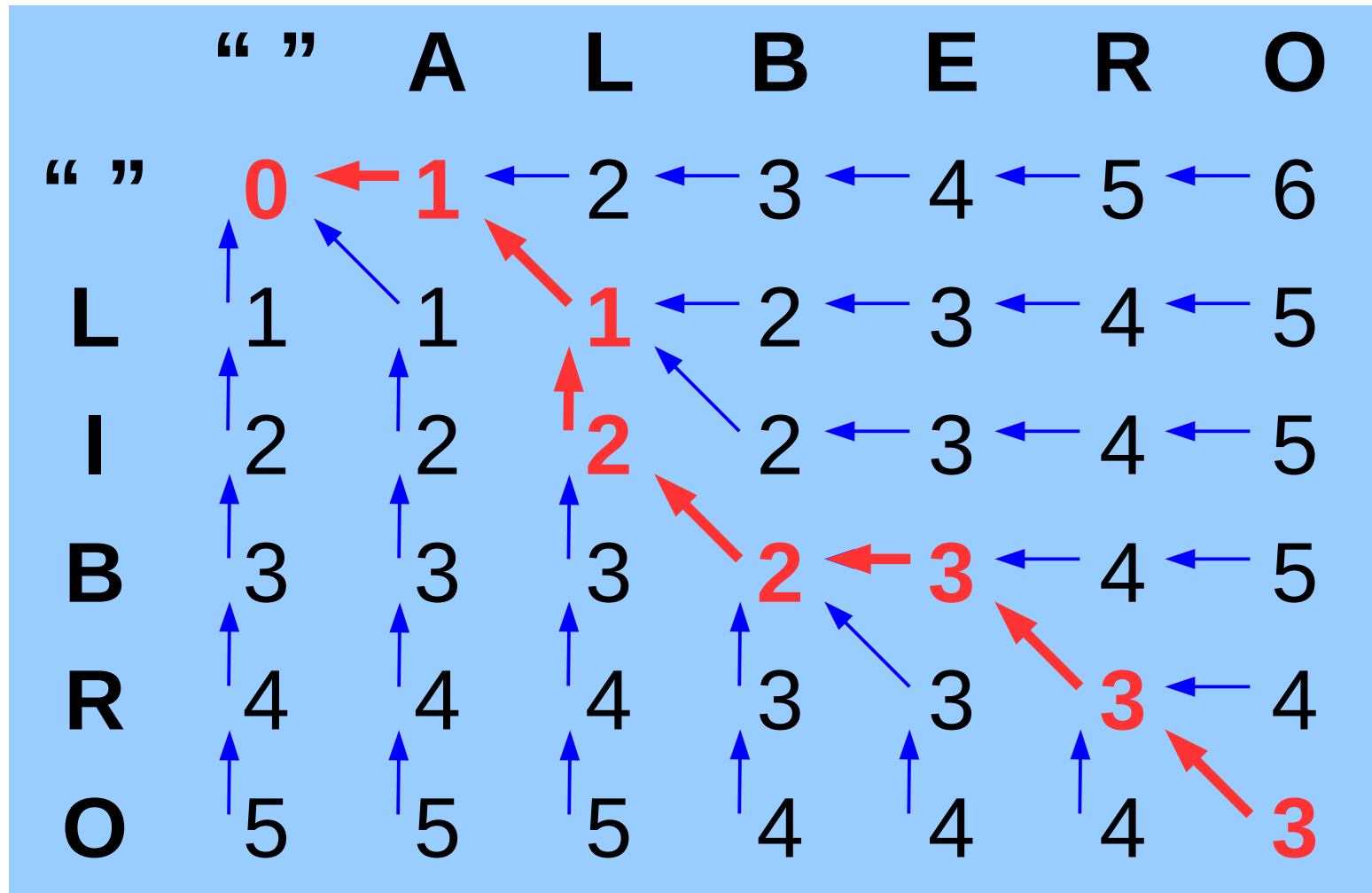
*Distanza di Levenshtein  
tra LIBRO e ALBERO*

# Esempio

	“ ”	A	L	B	E	R	O
“ ”	0	1	2	3	4	5	6
L	1	1	1	2	3	4	5
I	2	2	2	2	3	4	5
B	3	3	3	2	3	4	5
R	4	4	4	3	3	3	4
O	5	5	5	4	4	4	3

 = DELETE  
  = INSERT  
  = CHANGE

# Esempio



 = DELETE  
  = INSERT  
  = CHANGE



# Subset-Sum Problem / 1

- È dato un insieme  $X = \{1, 2, \dots, n\}$  di  $n$  oggetti
  - L'oggetto  $i$ -esimo ha peso  $p[i]$
  - I pesi sono numeri interi positivi
- Disponiamo un contenitore in grado di trasportare al massimo un peso  $C$  (capacità dello zaino)
- Vogliamo trovare un sottoinsieme  $Y$  di  $X$  (se esiste) tale che il peso complessivo degli oggetti in  $Y$  sia esattamente uguale a  $C$

# Subset-Sum Problem / 2

- Definizione dei problemi  $P(i, j)$ 
  - “Determinare se esiste un sottoinsieme (anche vuoto) dei primi  $i$  oggetti aventi peso complessivo esattamente uguale a  $j$ ”
- Definizione delle soluzioni  $B[i, j]$ 
  - $B[i, j] = \text{true}$  se e solo se esiste un sottoinsieme (anche vuoto) dei primi  $i$  oggetti  $\{1, 2, \dots, i\}$  avente peso complessivo uguale a  $j$

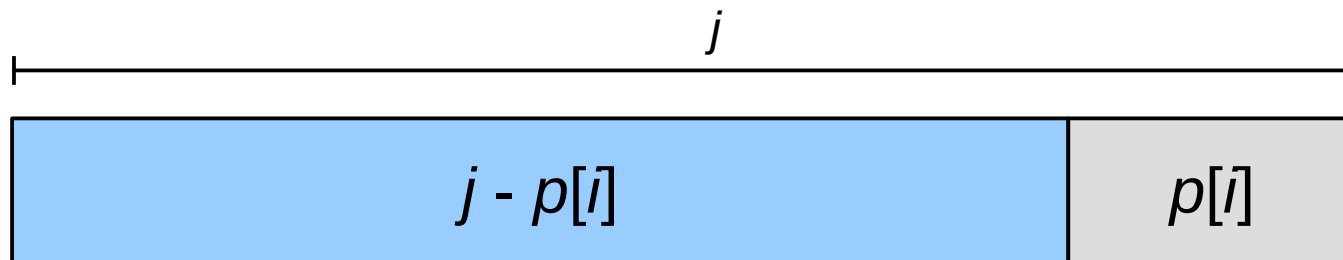
# Calcolo delle soluzioni: casi base

- $B[i, 0] = \text{true}$  per ogni  $i = 1, \dots, n$ 
  - L'insieme vuoto è sottoinsieme di qualunque insieme di oggetti, e ha peso uguale a 0
- $B[1, j] = \text{true}$  se  $j = p[1]$ 
  - La capacità dello zaino è pari al peso dell'unico oggetto a disposizione (il primo)
- $B[1, j] = \text{false}$  se  $j > 0, j \neq p[1]$ 
  - La capacità  $j$  è diversa dal peso dell'oggetto 1

$B[i, j] = \text{true}$  se e solo se esiste un sottoinsieme (anche vuoto) dei primi  $i$  oggetti  $\{1, 2, \dots, i\}$  avente peso complessivo esattamente uguale a  $j$

# Calcolo delle soluzioni: caso generale

- Se  $j < p[i]$  significa che l' $i$ -esimo oggetto è troppo pesante per essere contenuto nello zaino. In tal caso  $B[i, j] \leftarrow B[i - 1, j]$
- Se  $j \geq p[i]$  ho due possibilità
  - Inserire l'oggetto  $i$ -esimo nello zaino. In questo caso il problema  $P(i, j)$  ammette soluzione se  $B[i - 1, j - p[i]]$  è *true*
  - Non inserire l'oggetto  $i$ -esimo nello zaino. In questo caso il problema  $P(i, j)$  ammette soluzione se  $B[i - 1, j]$  è *true*



$B[i, j] = \text{true}$  se e solo se esiste un sottoinsieme (anche vuoto) dei primi  $i$  oggetti  $\{1, 2, \dots, i\}$  avente peso complessivo esattamente uguale a  $j$

# Quali oggetti fanno parte della soluzione (se esiste)?

- Sfruttiamo un array booleano  $U[i, j]$ 
  - $i = 1, \dots, n$
  - $j = 0, \dots, C$
- $U[i, j] = \text{true}$  se e solo se l'oggetto  $i$ -esimo fa parte di un sottoinsieme dei primi  $i$  oggetti  $\{1, \dots, i\}$  di peso complessivo  $j$

```

bool SubsetSum(integer p[1..n], integer C)
  bool B[1..n, 0..C], U[1..n, 0..C]; integer i, j;
  B[1,0]  $\leftarrow$  true; U[1,0]  $\leftarrow$  false;
  for j  $\leftarrow$  1 to C do
    if (j = p[1]) then
      B[1,j]  $\leftarrow$  true; U[1,j]  $\leftarrow$  true;
    else
      B[1,j]  $\leftarrow$  false; U[1,j]  $\leftarrow$  false;
    endif
  endfor
  for i  $\leftarrow$  2 to n do
    for j  $\leftarrow$  0 to C do
      if ( j  $\geq$  p[i] ) then
        B[i,j]  $\leftarrow$  B[i-1,j] or B[i-1,j-p[i]];
        U[i,j]  $\leftarrow$  B[i-1,j-p[i]];
      else
        B[i,j]  $\leftarrow$  B[i-1,j]; U[i,j]  $\leftarrow$  false;
      endif
    endfor
  endfor
  if ( not B[n,C] ) then
    print "nessuna soluzione";
  else
    i  $\leftarrow$  n; j  $\leftarrow$  C;
    while ( i > 0 and j > 0 ) do
      if U[i,j] then
        print "ho usato l'oggetto " i
        j  $\leftarrow$  j - p[i];
      endif
      i  $\leftarrow$  i - 1;
    endwhile;
  endif
  return B[n, C];

```