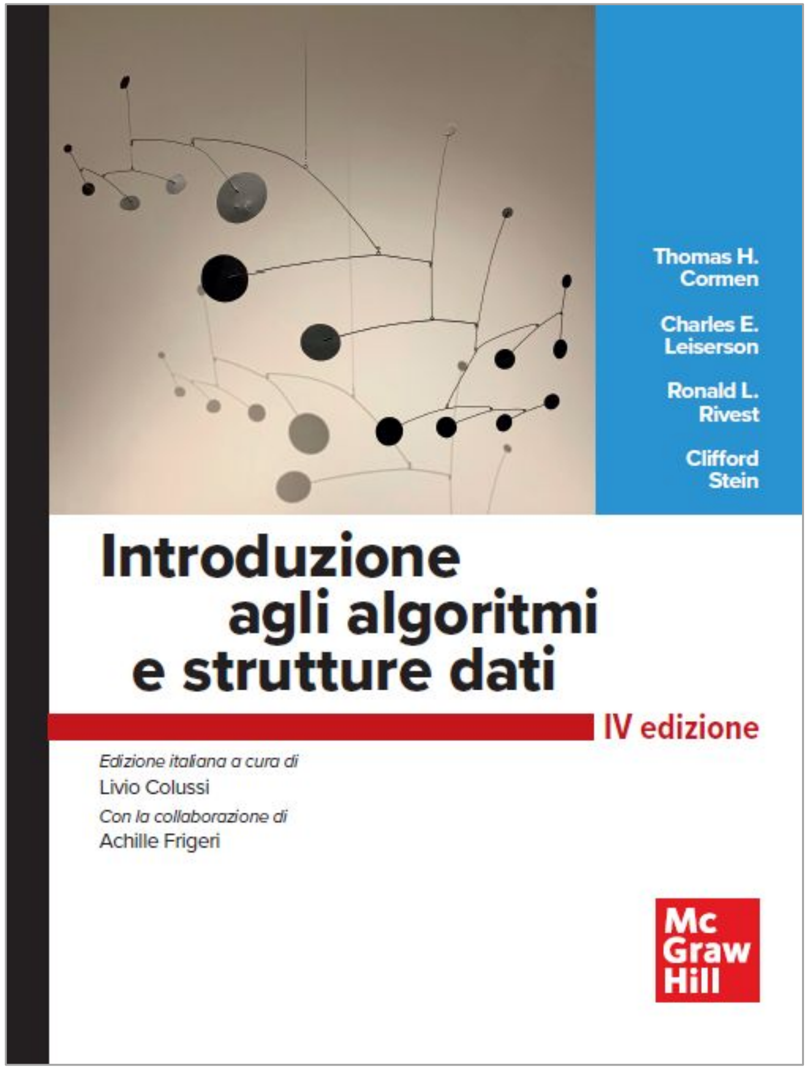


Capitolo 15

Algoritmi avidi

Algoritmi avidi

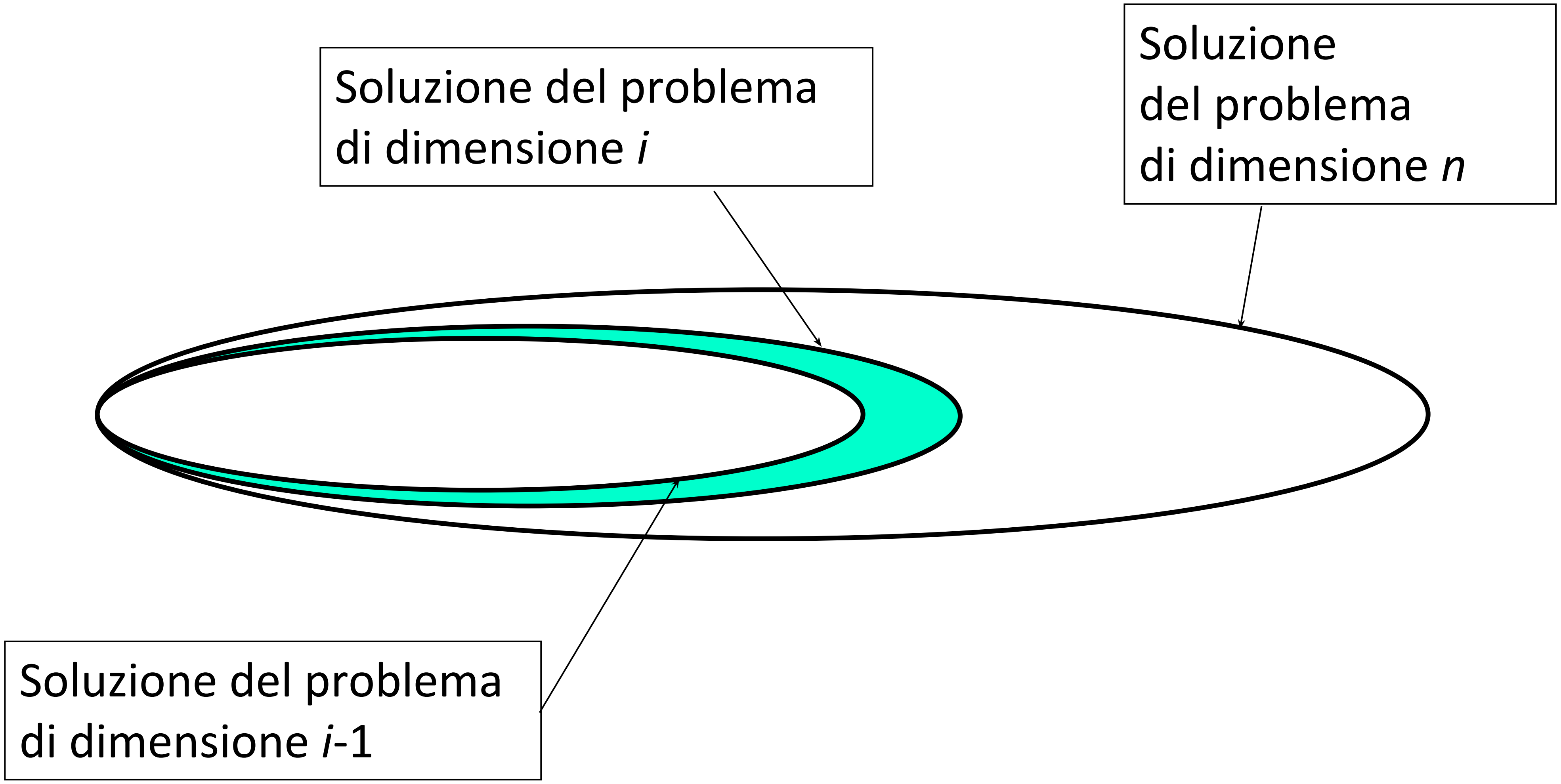
Tecniche di soluzione dei problemi viste finora:

- Metodo iterativo
- Divide et impera
- Programmazione dinamica

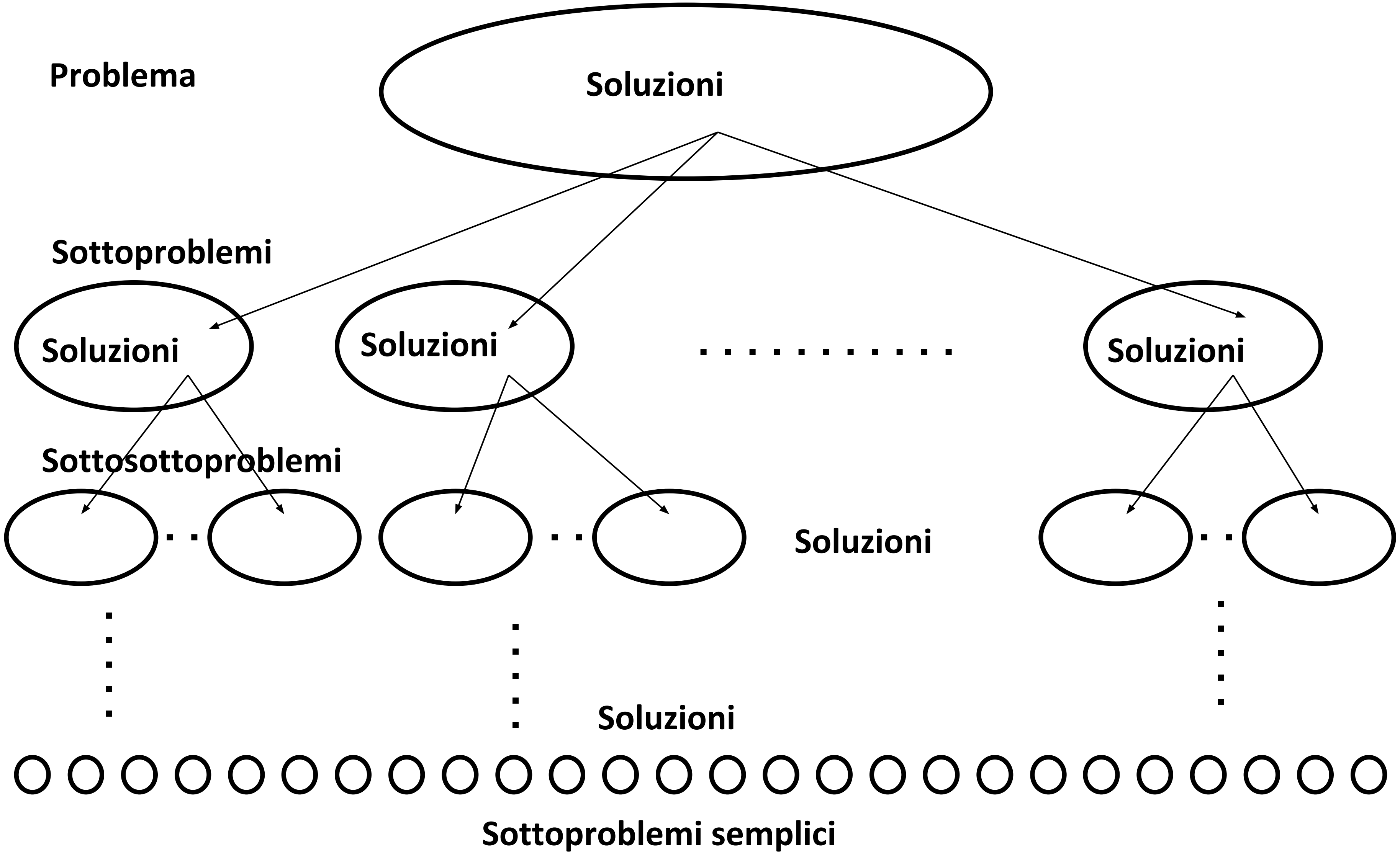
Nuova tecnica:

- Algoritmi avidi

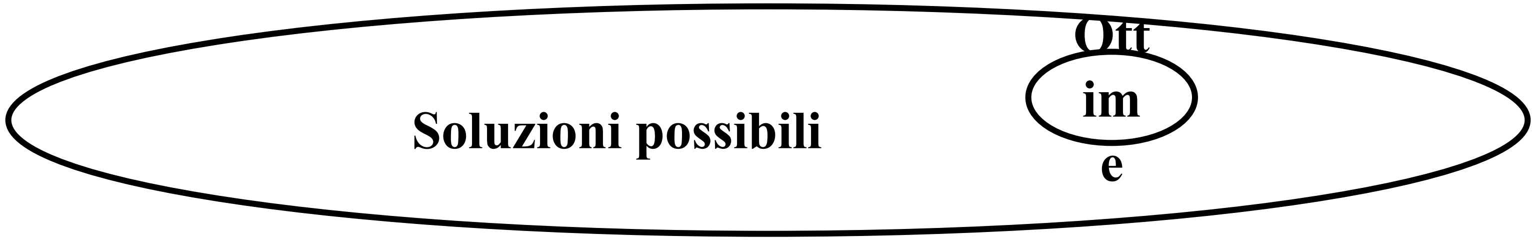
Metodo iterativo



Divide et impera



In un problema di ottimizzazione abbiamo un insieme generalmente molto grande di soluzioni e dobbiamo scegliere tra di esse una soluzione che sia ottima in qualche senso (costo minimo, valore massimo, lunghezza minima, ecc.)



Possiamo risolvere un problema di questo tipo con una ***enumerazione esaustiva***

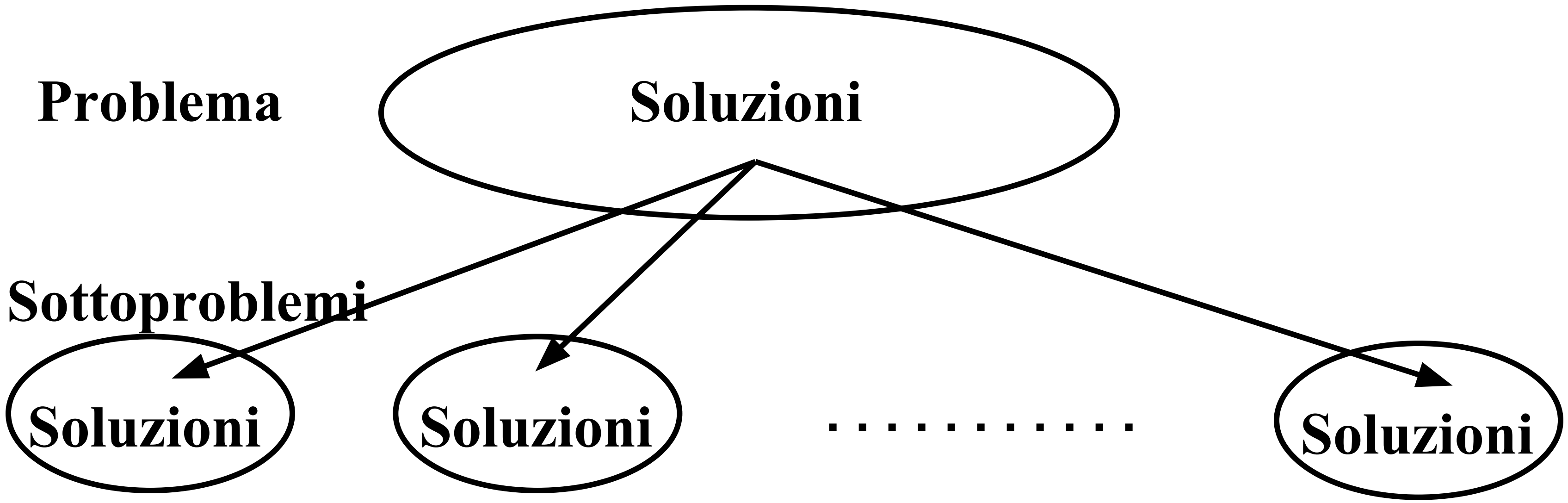
- si generano tutte le soluzioni possibili,

- si calcola il costo di ciascuna di esse

- e infine se ne seleziona una di ottima.

Purtroppo l’insieme di soluzioni è generalmente molto grande (spesso esponenziale nella dimensione dell’input) per cui una enumerazione esaustiva richiede tempo esponenziale.

Molto spesso le soluzioni di un problema di ottimizzazione si possono costruire estendendo o combinando tra loro soluzioni di sottoproblemi.



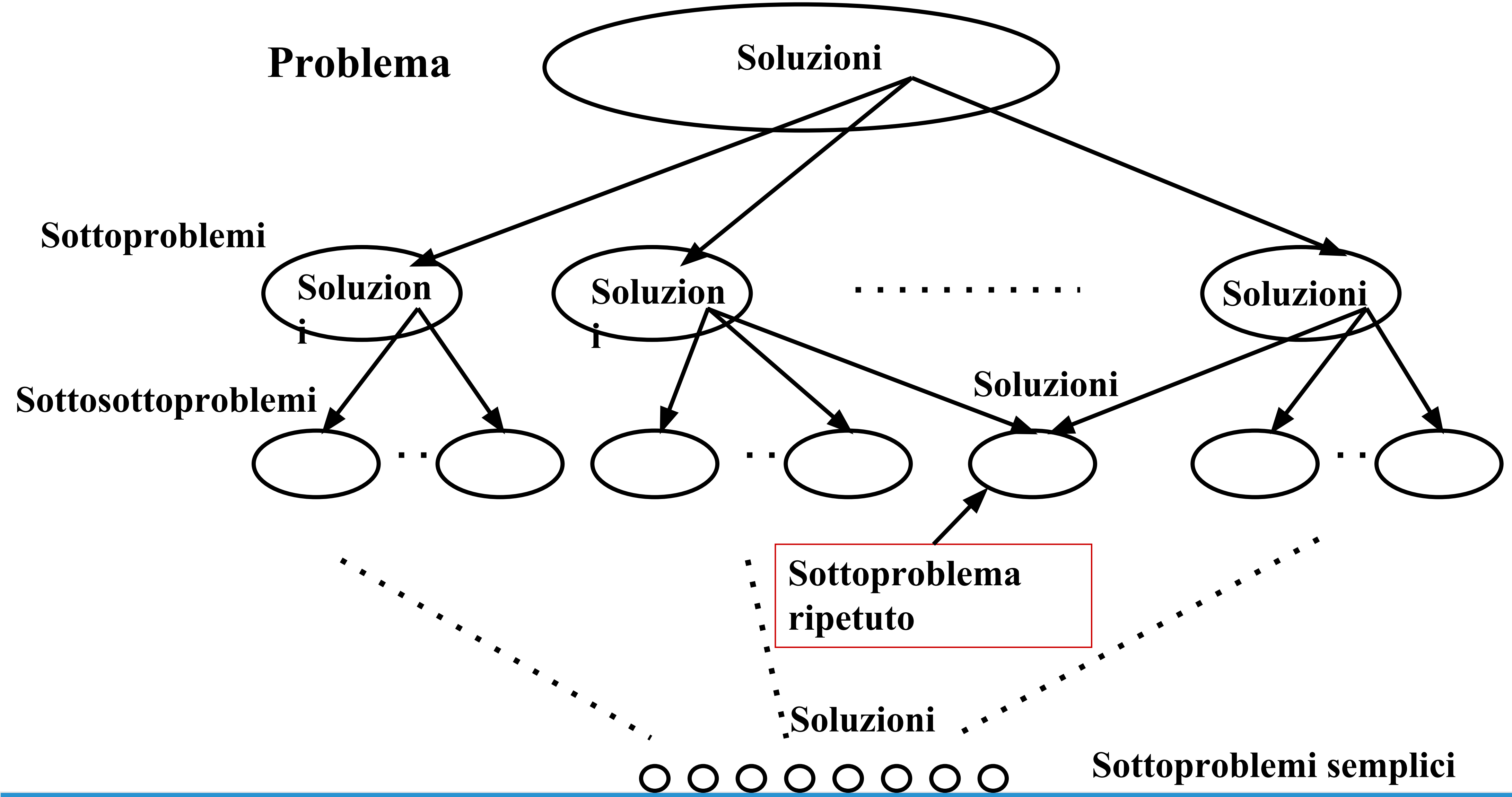
Esempio: raggiungere Trieste da Torino.

Sottoproblemi: Torino-Asti, Asti-Trieste; Torino-Novara, Novara-Trieste, ecc.

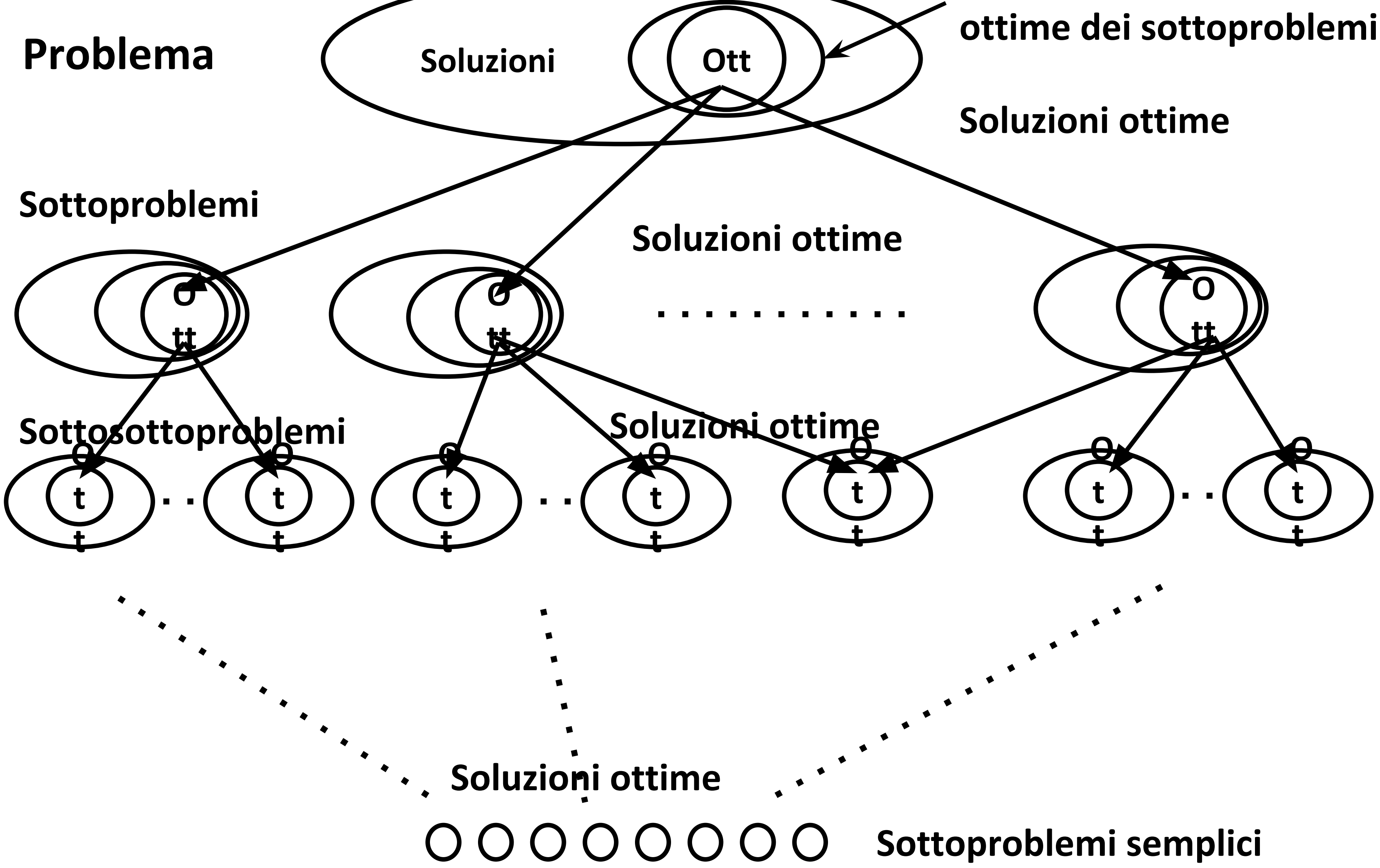
Abbiamo visto che perché la ***programmazione dinamica*** sia vantaggiosa rispetto all'enumerazione esaustiva bisogna che siano soddisfatte due condizioni:

1. esistenza di sottoproblemi ripetuti;
2. sottostruttura ottima.

Sottoproblemi ripetuti

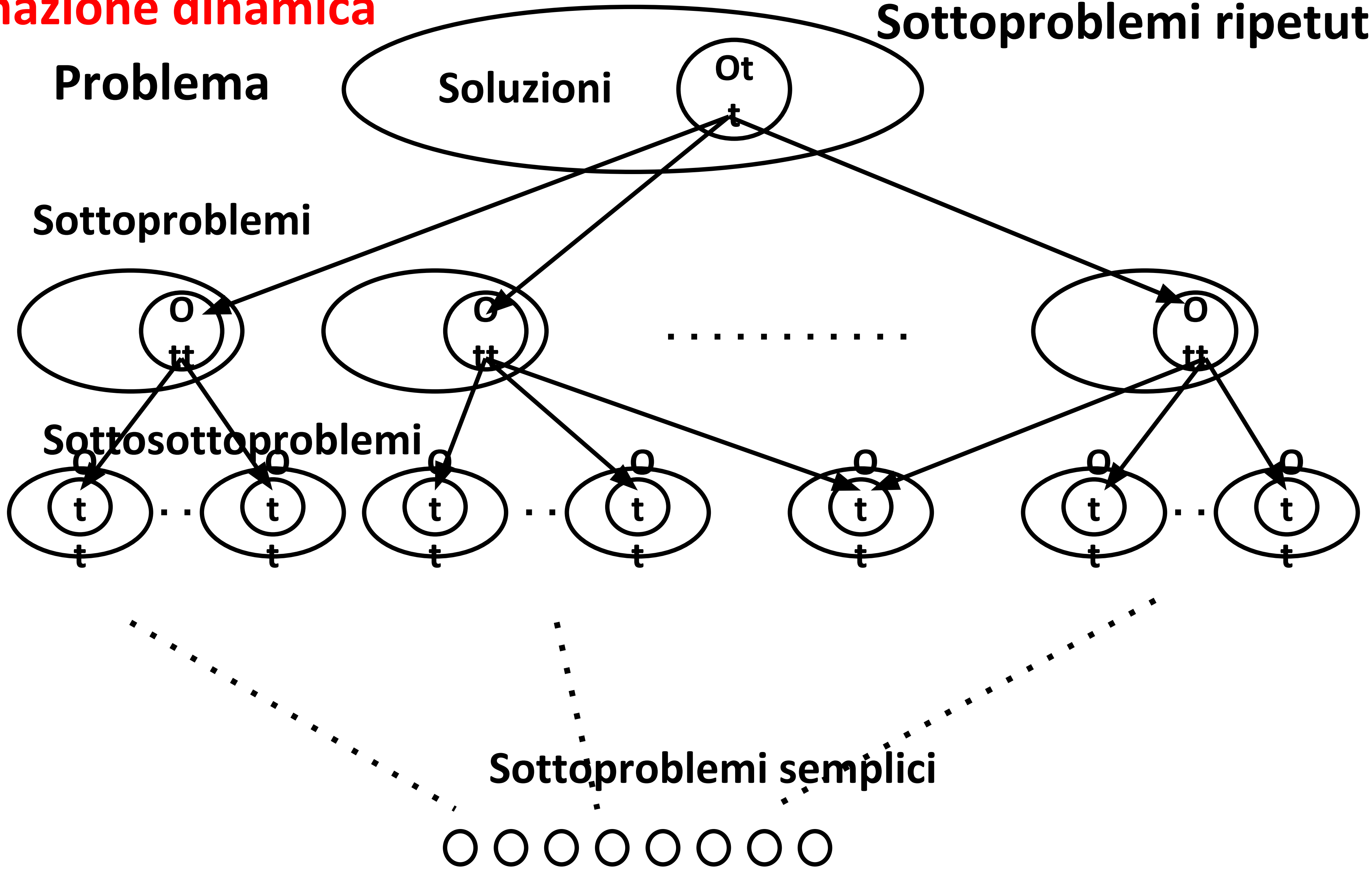


Sottostruttura ottima

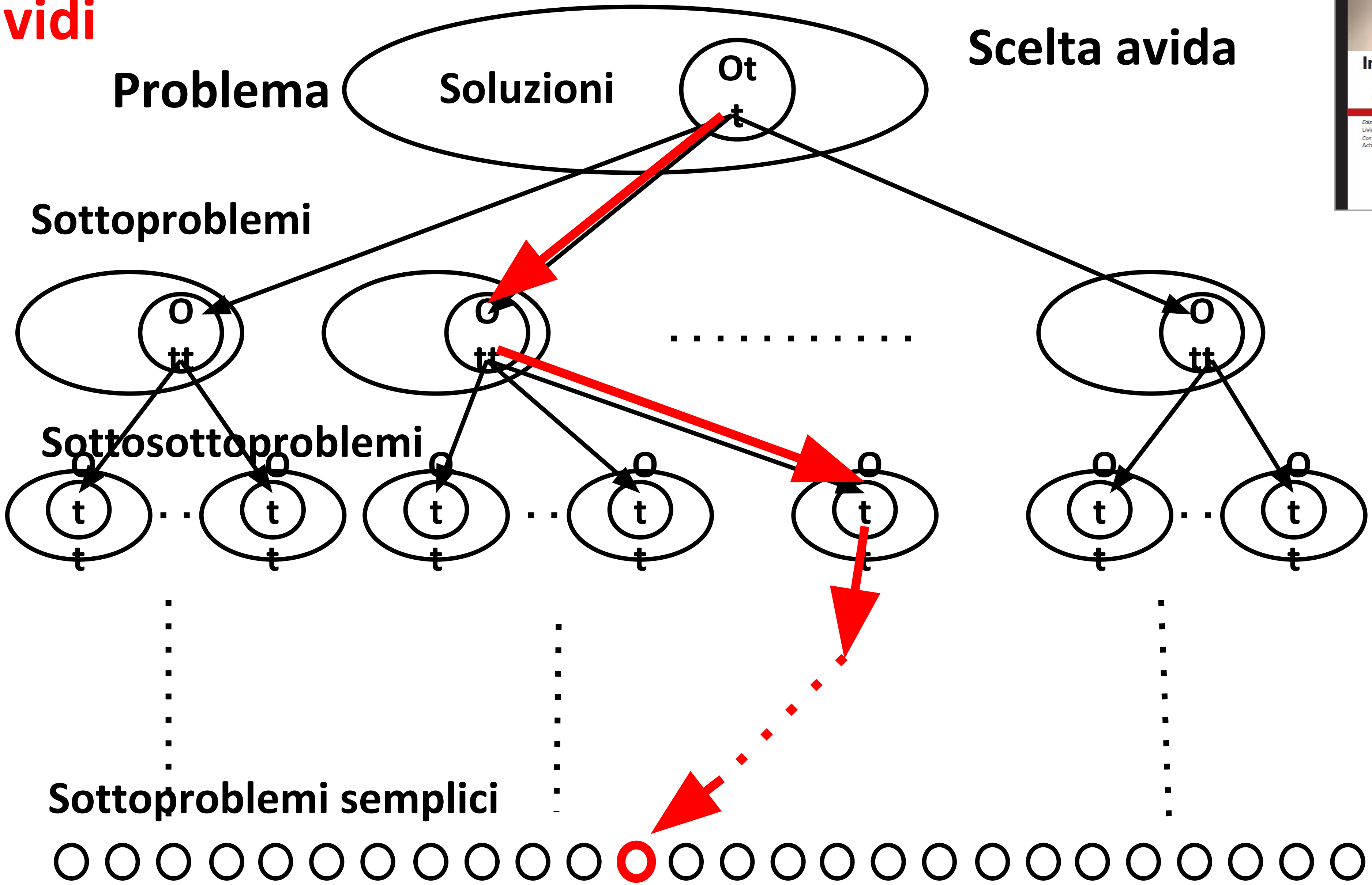


Programmazione dinamica

Sottoproblemi ripetuti



Algoritmi avidi



Algoritmi avidi

1) Ogni volta si fa la scelta che sembra migliore localmente;
2) in questo modo per alcuni problemi si ottiene una soluzione globalmente ottima.

Problema della scelta delle attività

n ***attività*** $a_1,\ldots,a_n$ usano la stessa risorsa (es: lezioni da tenere in una stessa aula).

Ogni attività a_i ha un ***tempo di inizio*** s_i ed un ***tempo di fine*** f_i con $s_i < f_i$.

a_i occupa la risorsa nell'intervallo di tempo $[s_i, f_i)$.

a_i ed a_j sono ***compatibili*** se $[s_i, f_i)$ ed $[s_j, f_j)$ sono disgiunti.

Problema: scegliere il massimo numero di attività compatibili.

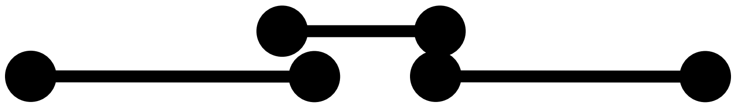
Strategie avide

Scegliere l’attività che inizia per prima

Non funziona

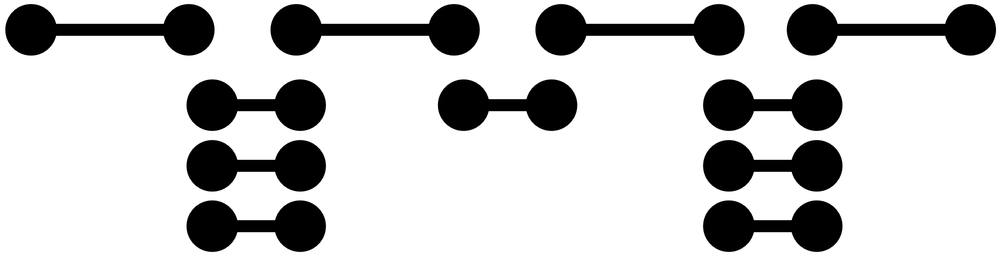
Scegliere l’attività che dura meno tempo

Non funziona



Scegliere l’attività incompatibile con il minor numero di altre attività

Non funziona



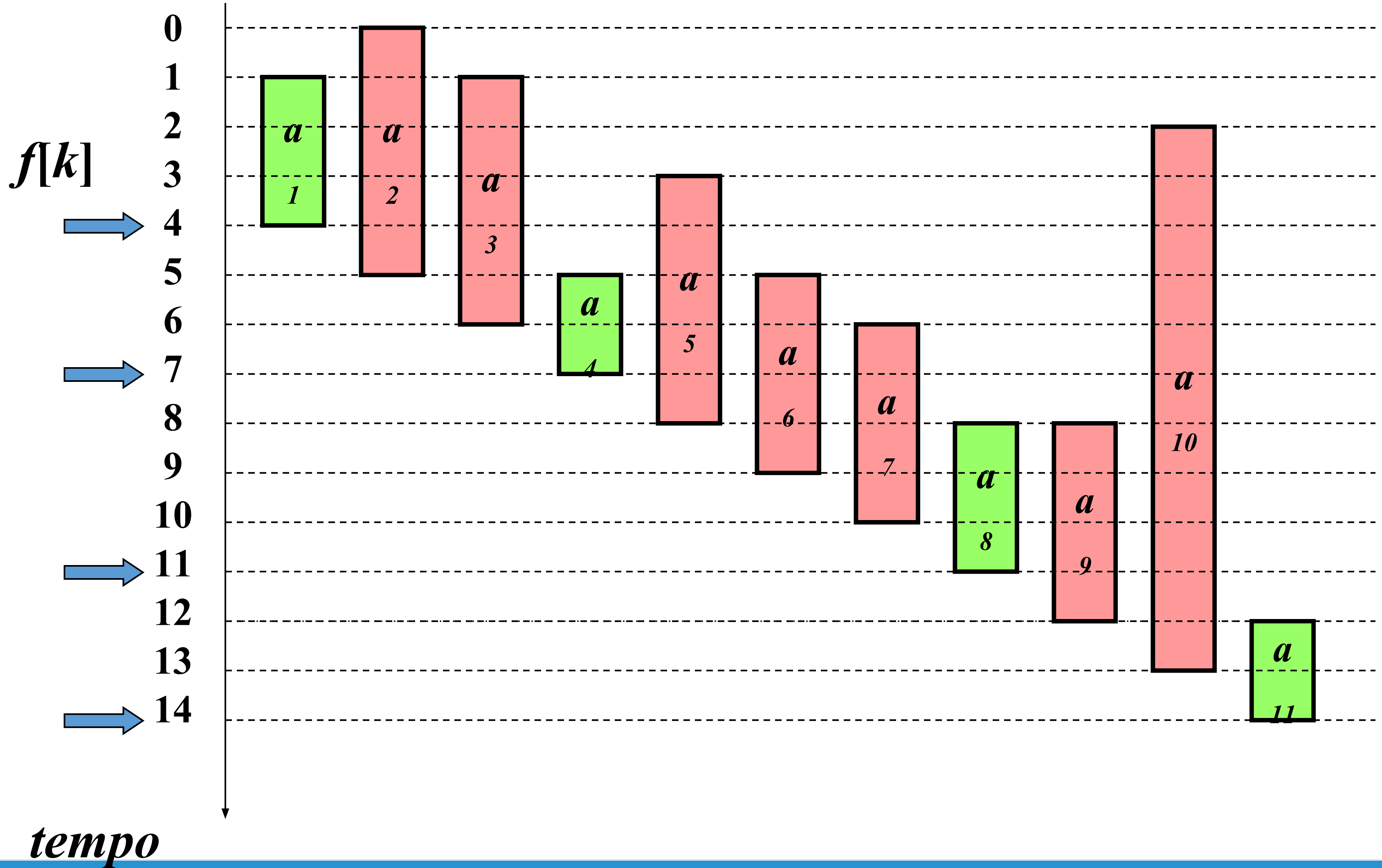
Strategia che funziona:
Scegliere l'attività che termina per prima.

Greedy-Activity-Selector(*Attività*)
  Scelte = ∅, *Compatibili* = *Attività*
  while *Compatibili* ≠ ∅
    " in *Compatibili* scegli l'attività '*a*' che
      termina per prima, aggiungi '*a*' a
      Scelte e togli da *Compatibili* tutte
      le attività incompatibili con '*a*' "
  return *Scelte*

Per implementarla supponiamo che le n attività $\boldsymbol{a_1,\dots,a_n}$ siano ordinate per tempo di fine non decrescente $\boldsymbol{f_1 \le \dots \le f_n}$, altrimenti possiamo ordinarle in tempo $\boldsymbol{O(n \log n)}$.
I tempi di inizio e fine sono dati nei due array s e f.

Greedy-Activity-Selector(s, f, n) // $f_1 \le \dots \le f_n$
  $A = \{a_1\}$, $k = 1$
  for $m = 2$ **to** n
    if $s[m] \ge f[k]$
      $A = A \cup \{a_m\}$, $k = m$
  return A

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	0	1	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Greedy-Activity-Selector(s, f, n)

$A = \{a_1\}$, $k = 1$

for $m = 2$ **to** n

  if $s[m] \geq f[k]$

    $A = A \cup \{a_m\}$, $k = m$

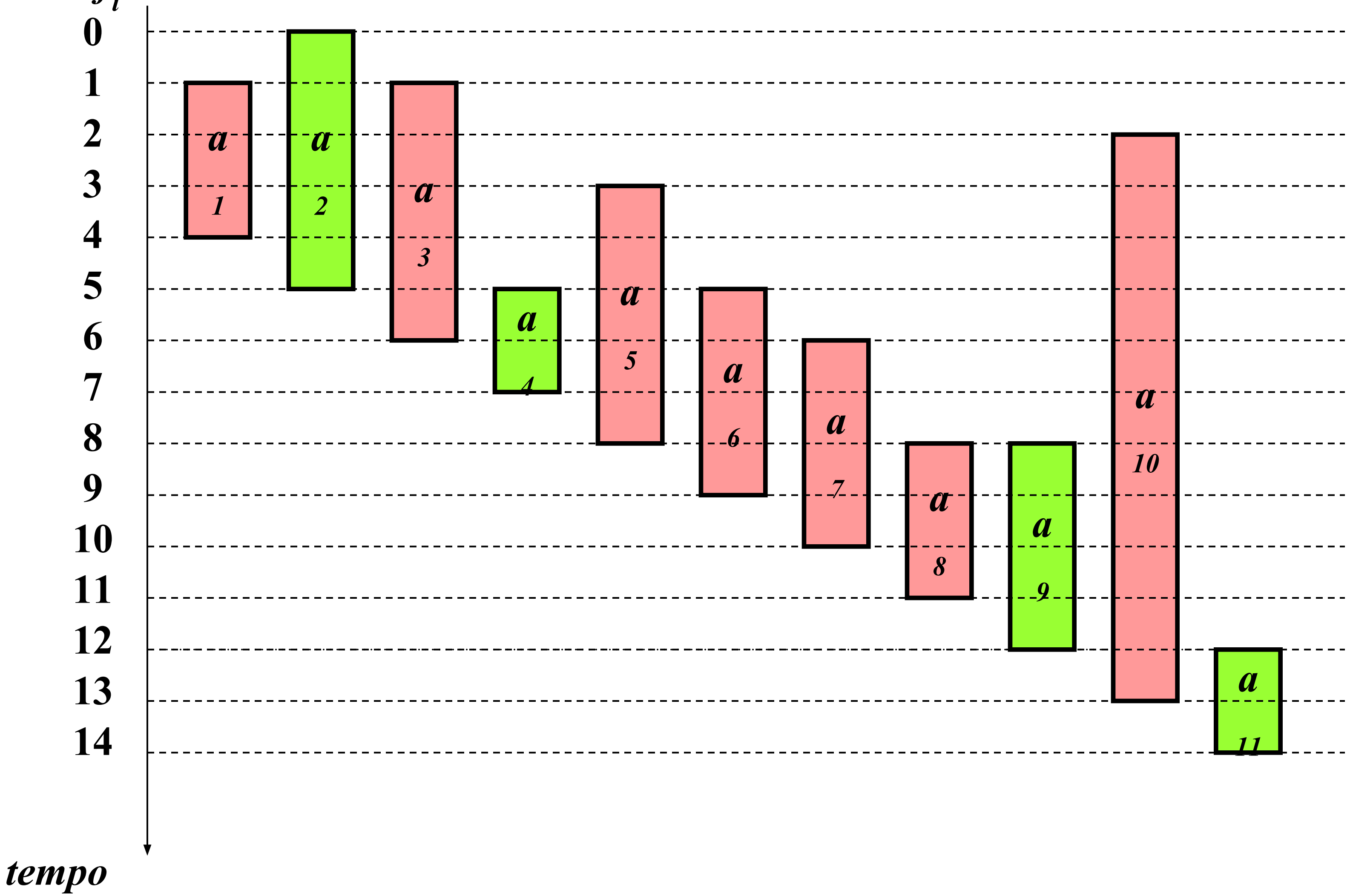
return A

La soluzione trovata contiene quattro attività.

1) La soluzione trovata con l’algoritmo avido è l’unica possibile che contiene quattro attività?

2) La soluzione trovata con l’algoritmo avido è ottima o esistono anche soluzioni con più di quattro attività?

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	0	1	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Cerchiamo di rispondere alla seconda domanda

2) La soluzione trovata con l'algoritmo avido è ottima o esistono anche soluzioni con più di quattro attività?

Greedy-Activity-Selector(s, f, n) $// f_1 \leq \ldots \leq f_n$
  $A = \{a_1\}, k = 1$
  for $m = 2$ **to** n
    if $s[m] \geq f[k]$
      $A = A \cup \{a_m\}, k = m$
  return A

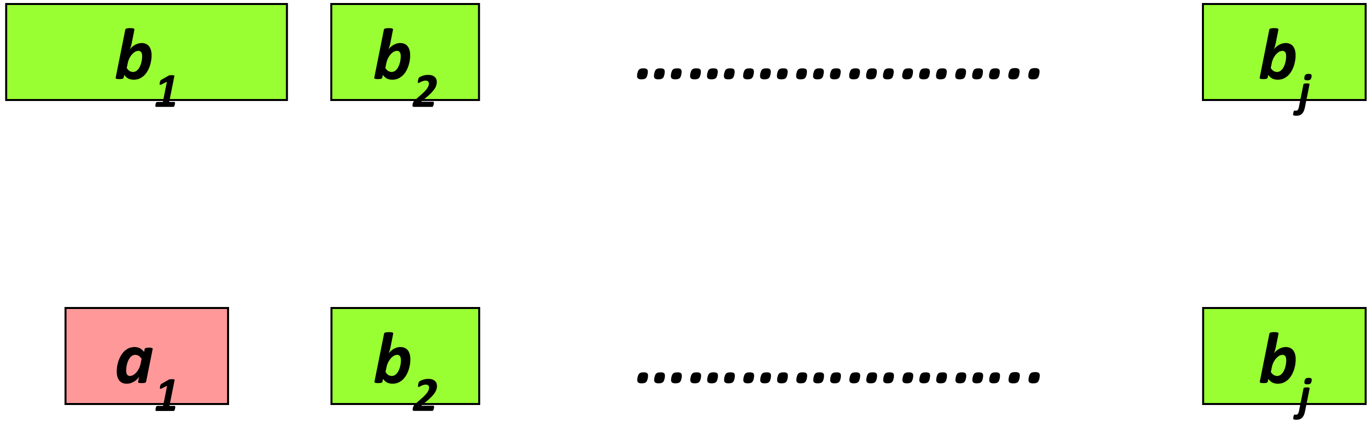
L’algoritmo comincia con scegliere la prima attività $\boldsymbol{a}_1$ (quella con tempo di fine minimo)

Siamo sicuri che questa scelta non possa compromettere il risultato?

In altre parole: esiste sempre una soluzione ottima che contiene $\boldsymbol{a}_1$?

La risposta è affermativa.

Sia $b_1,\ldots,b_j$ una qualsiasi soluzione ottima (ne esiste certamente almeno una) che supponiamo ordinata per tempo di fine



1. le attività $\boldsymbol{b_2, \ldots, b_m}$ sono compatibili con $\boldsymbol{b_1}$ e terminano dopo $\boldsymbol{b_1}$.

2. Esse hanno quindi tempo di inizio maggiore o uguale al tempo di fine $\boldsymbol{f}$ di $\boldsymbol{b_1}$.

3. Ma $\boldsymbol{a_1}$ ha il tempo di fine $\boldsymbol{f_1}$ minimo in assoluto e quindi termina prima di $\boldsymbol{b_1}$.

4. Quindi anche $\boldsymbol{a_1}$ è compatibile con $\boldsymbol{b_2, \ldots, b_m}$.

Dunque $\boldsymbol{a_1, b_2, \ldots, b_m}$ è una soluzione ottima che contiene $\boldsymbol{a_1}$.

k viene posto ad **1** ed aggiornato ad ***m*** ogni volta che si sceglie una nuova attività $\boldsymbol{a_m}$

```
Greedy-Activity-Selector(s, f, n)   // f_1 ≤ … ≤ f_n
  A = {a_1}, k = 1
  for m = 2 to n
    if s[m] ≥ f[k]
      A = A ∪ {a_m}, k = m
  return A
```

Siccome le attività sono ordinate per tempo di fine non decrescente, ***f*[*k*]** è il massimo tempo finale delle attività selezionate precedentemente.

Con il test:

$$\textbf{if } s[m] \geq f[k]$$
$$A = A \cup \{a_m\},\ k = m$$

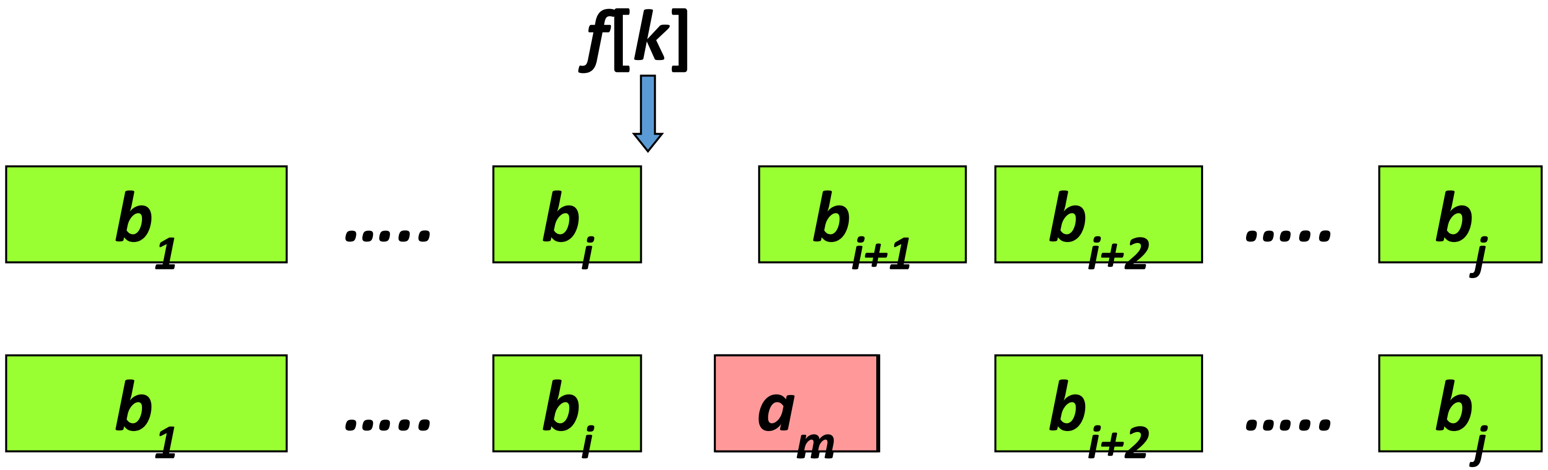
l'algoritmo seleziona la prima attività $\boldsymbol{a_m}$ il cui tempo di inizio $\boldsymbol{s[m]}$ è maggiore o uguale di $\boldsymbol{f[k]}$

Siamo sicuri che questa scelta non comprometta il risultato?

In altre parole: esiste sempre una soluzione ottima che contiene $\boldsymbol{a_m}$ e le attività finora scelte?

La risposta è ancora affermativa.

Assumiamo che esista una soluzione ottima $\boldsymbol{b_1,\dots,b_i,b_{i+1},\dots,b_j}$ che estende le attività $\boldsymbol{b_1,\dots,b_i}$ finora scelte e supponiamo $\boldsymbol{b_1,\dots,b_i}$ e $\boldsymbol{b_{i+1},\dots,b_j}$ ordinate per tempo di fine



1. Le attività scartate finora iniziano prima e hanno tempo di fine maggiore o uguale ad una delle attività precedentemente scelte.

2. Esse sono quindi incompatibili con almeno una delle attività $b_1, \ldots, b_k$ finora scelte.

3. $b_{k+1}, \ldots, b_m$ sono invece compatibili con tutte le attività $b_1, \ldots, b_k$ e quindi non sono tra quelle scartate precedentemente.

4. $b_{k+1}, \ldots, b_m$ hanno sia tempo di fine che tempo di inizio maggiore o uguale di $f[k]$.

5. L'attività $\boldsymbol{a_i}$ è quella con tempo di fine $\boldsymbol{f_i}$ minimo in assoluto tra quelle compatibili con $\boldsymbol{b_1}, \ldots, \boldsymbol{b_k}$ e quindi $\boldsymbol{f_i} \leq \boldsymbol{f}$.

6. Siccome $\boldsymbol{b_{k+1}}$ è compatibile con $\boldsymbol{b_{k+2}}, \ldots, \boldsymbol{b_m}$ i tempi di inizio di $\boldsymbol{b_{k+2}}, \ldots, \boldsymbol{b_m}$ sono maggiori o uguali di $\boldsymbol{f}$ e quindi anche di $\boldsymbol{f_i}$.

7. Dunque anche $\boldsymbol{a_i}$ è compatibile con $\boldsymbol{b_{k+2}}, \ldots, \boldsymbol{b_m}$

8. Pertanto $\boldsymbol{b_1}, \ldots, \boldsymbol{b_k}, \boldsymbol{a_i}, \boldsymbol{b_{k+2}}, \ldots, \boldsymbol{b_m}$ è una soluzione ottima contenente $\boldsymbol{a_i}$ e $\boldsymbol{b_1}, \ldots, \boldsymbol{b_k}$.

Sappiamo quindi che durante tutta l'esecuzione dell'algoritmo esiste sempre una soluzione ottima contenente le attività $\boldsymbol{b_1,\ldots,b_i}$ scelte fino a quel momento

Quando l'algoritmo termina non ci sono altre attività compatibili con $\boldsymbol{b_1,\ldots,b_i}$ e quindi le attività $\boldsymbol{b_1,\ldots,b_i}$ sono una soluzione ottima

L'algoritmo è ***avido*** perché ad ogni passo, tra tutte le attività compatibili con quelle già scelte, sceglie quella che termina prima.

Questa scelta è localmente ottima (avida) perché è quella che lascia più tempo a disposizione per le successive attività.

Elementi della strategia avida

Sottostruttura ottima: ogni soluzione ottima non elementare si compone di soluzioni ottime di sottoproblemi.

Proprietà della scelta avida: la scelta ottima localmente (avida) non pregiudica la possibilità di arrivare ad una soluzione globalmente ottima.

Codici di Huffman

I *__codici di Huffman__* vengono usati nella compressione dei dati.

Essi permettono un risparmio compreso tra il **20%** ed il **90%**.

Sulla base delle frequenze con cui ogni carattere appare nel file, l’algoritmo avido di Huffman trova un ***codice ottimo***, ossia un modo ottimale di associare ad ogni carattere una sequenza di bit, detta ***parola di codice***.

Sia dato un file di **120** caratteri con frequenze:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5

Usando un codice a lunghezza fissa occorrono **3** bit per rappresentare **6** caratteri. Ad esempio

carattere	a	b	c	d	e	f
cod. fisso	000	001	010	011	100	101

Per codificare il file occorrono **120×3 = 360** bit.

Possiamo fare meglio con un codice a lunghezza variabile che assegni codici più corti ai caratteri più frequenti.
Ad esempio con il codice

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. var.	0	101	100	111	1101	1100

Bastano **57×1 + 49×3 + 14×4 = 260** bit

Codici prefissi

Un ***codice prefisso*** è un codice in cui nessuna parola codice è prefisso (parte iniziale) di un’altra

Ogni codice a lunghezza fissa è ovviamente prefisso.

Ma anche il codice a lunghezza variabile che abbiamo appena visto è un codice prefisso.

Codifica e decodifica sono semplici con i codici prefissi.

Con il codice prefisso

carattere	a	b	c	d	e	f
cod. var.	0	101	100	111	1101	1100

la codifica della stringa **`abc`** è

`0101100`

La decodifica è pure semplice.

Siccome nessuna parola codice è prefisso di un’altra, la prima parola codice del file codificato risulta univocamente determinata.

Per la decodifica basta quindi:

1. individuare la prima parola codice del file codificato
2. tradurla nel carattere originale e aggiungere tale carattere al file decodificato
3. rimuovere la parola codice dal file codificato
4. ripetere l'operazione per i caratteri successivi

Ad esempio con il codice

carattere	a	b	c	d	e	f
cod. var.	0	101	100	111	1101	1100

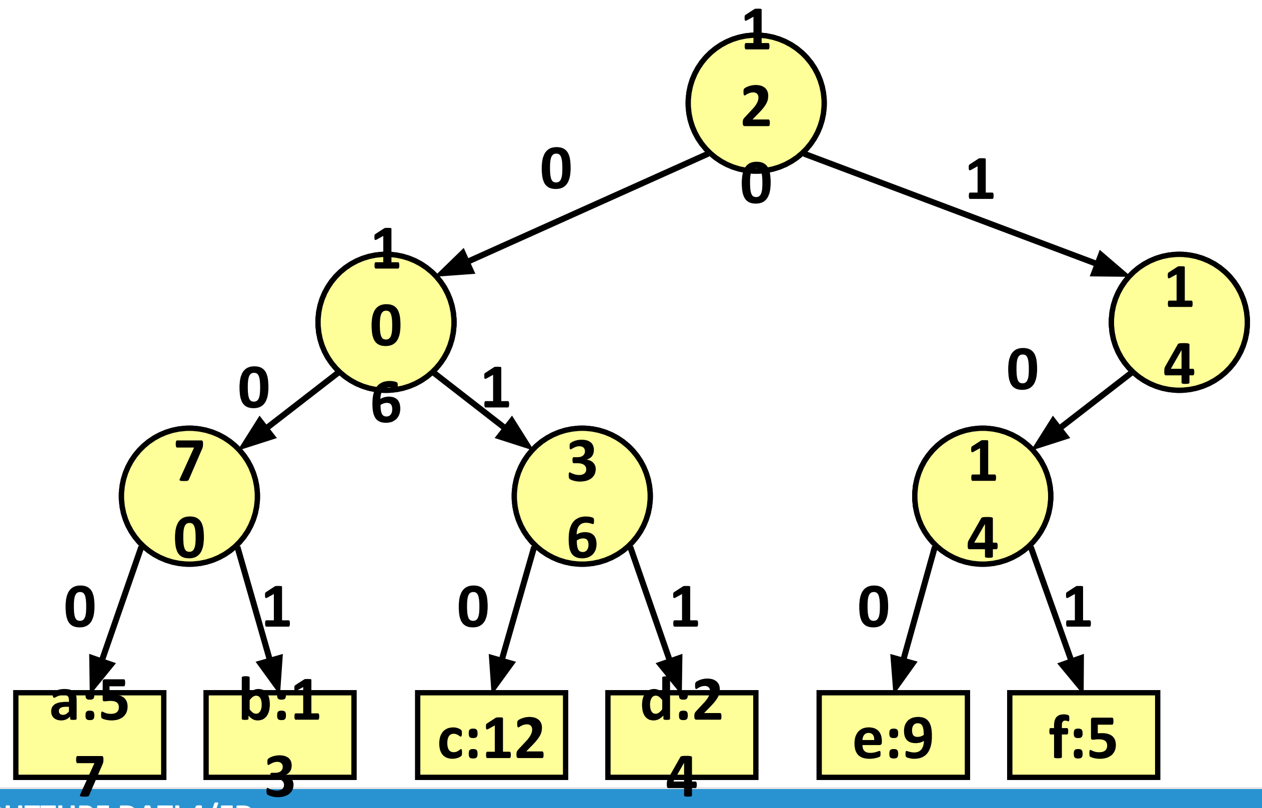
la suddivisione in parole codice della stringa di bit **001011101** è **0·0·101·1101** a cui corrisponde la stringa **aabe**

Per facilitare la suddivisione del file codificato in parole codice è comodo rappresentare il codice con un albero binario.

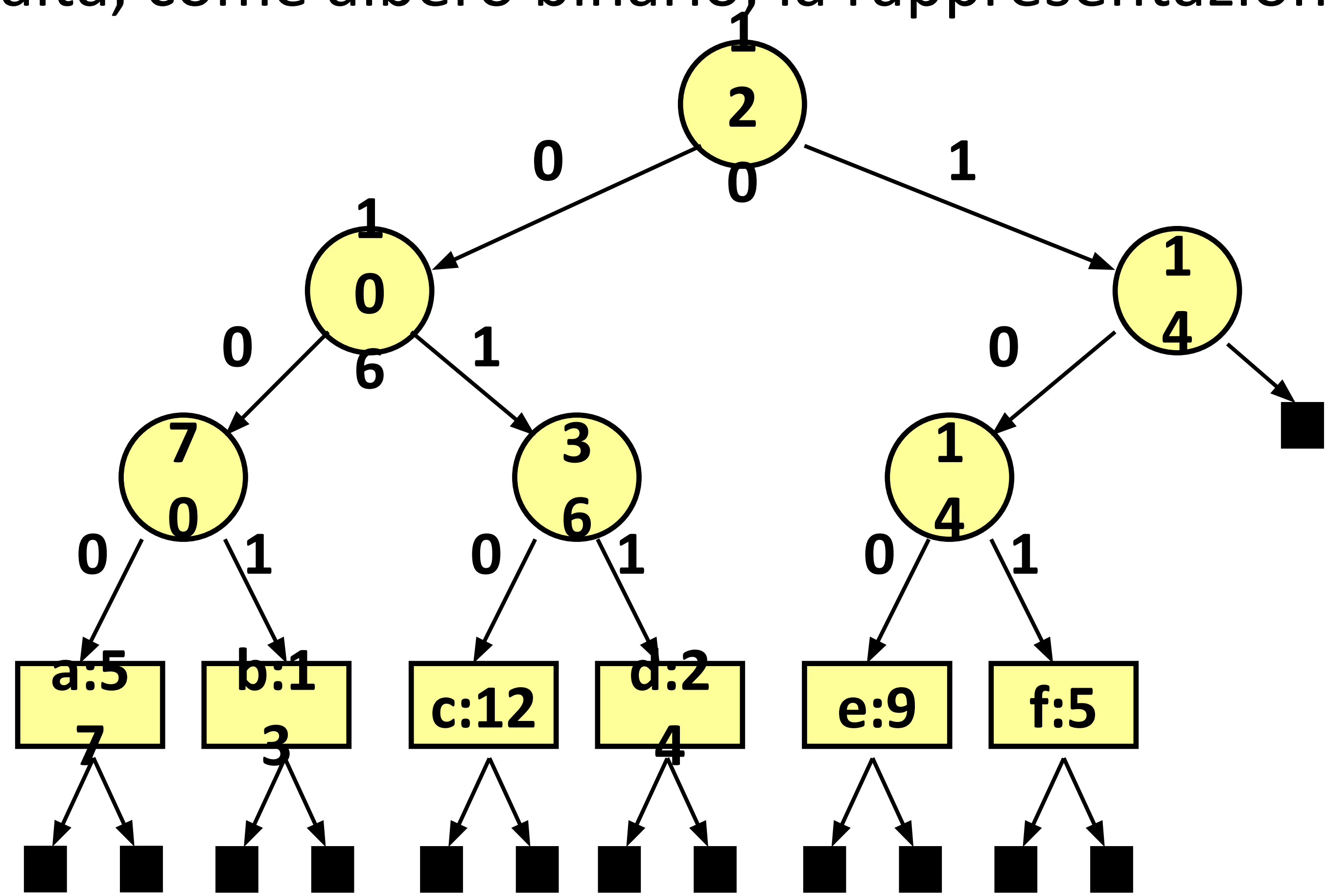
Esempio: il codice a lunghezza fissa

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. fisso	000	001	010	011	100	101

si rappresenta con l'albero



In realtà, come albero binario, la rappresentazione sarebbe

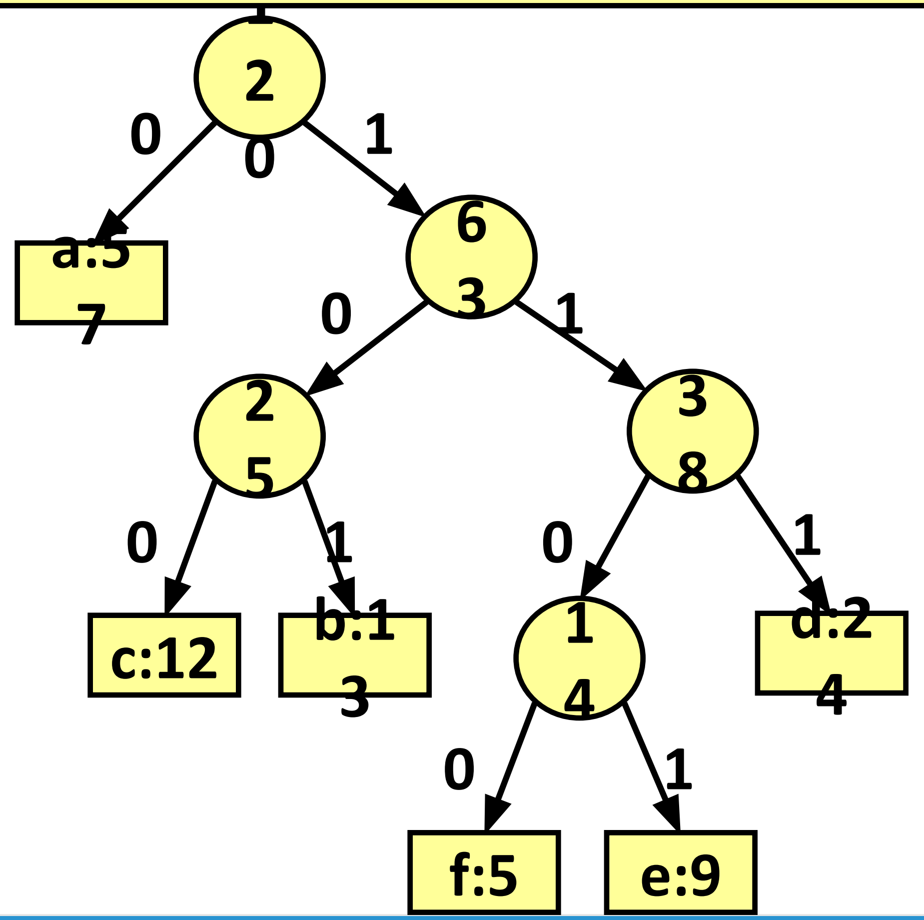


eliminiamo le foglie e chiamiamo foglie i nodi interni senza figli

Il codice a lunghezza variabile

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. var.	0	101	100	111	1101	1100

è rappresentato da



La lunghezza in bit del file codificato con il codice rappresentato da un albero ***T*** è:

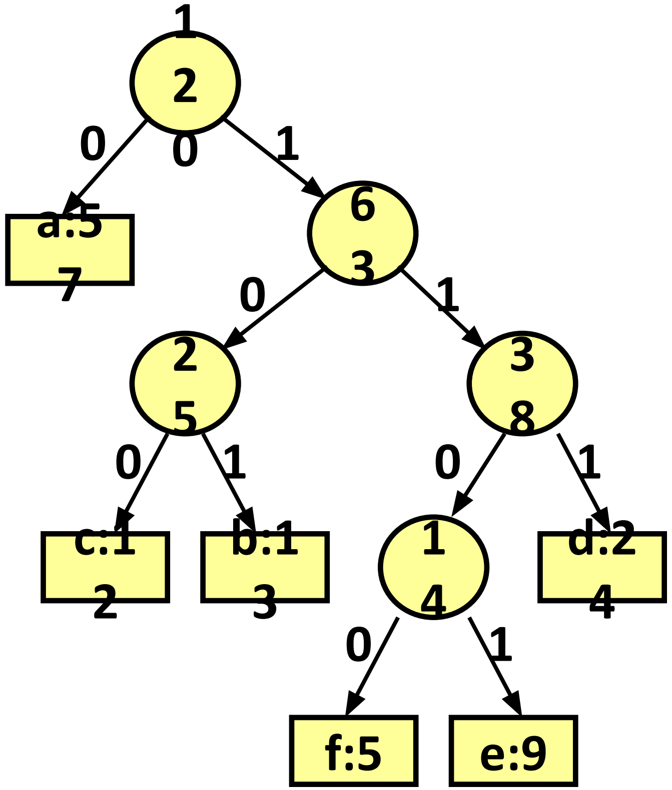
$$B(T) = \sum_{c \in \Sigma} f_c d_T(c)$$

in cui la sommatoria è estesa a tutti i caratteri ***c*** dell'alfabeto $\boldsymbol{\Sigma}$, $\boldsymbol{f_c}$ è la frequenza del carattere ***c*** e $\boldsymbol{d_T(c)}$ è la profondità della foglia che rappresenta il carattere ***c*** nell'albero ***T***

Nota: assumiamo che l'alfabeto $\boldsymbol{\Sigma}$ contenga almeno due caratteri. In caso contrario basta un numero per rappresentare il file: la sua lunghezza

La lunghezza in bit del file codificato è anche:

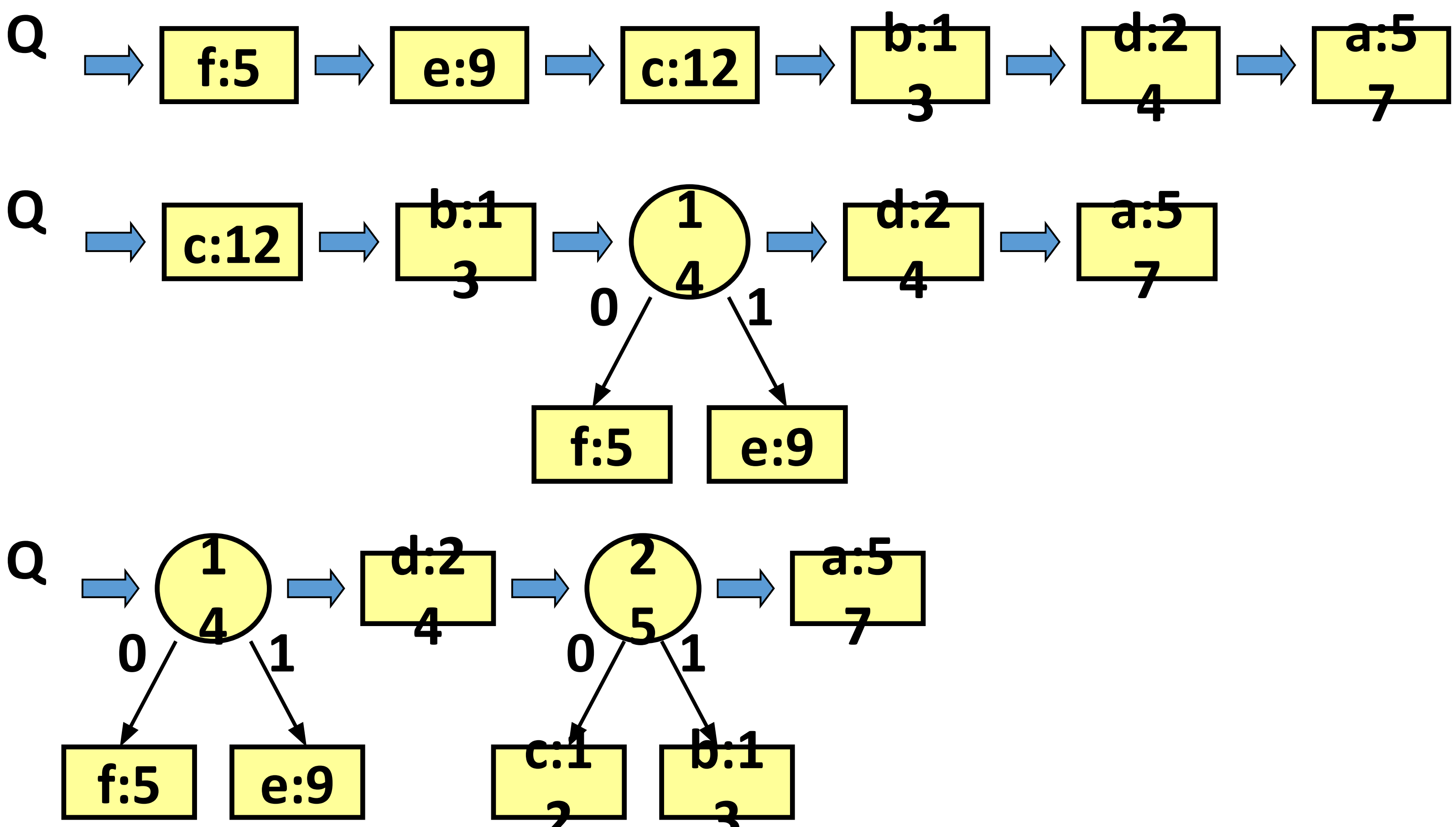
$$B(T) = \sum_{x \text{ nodo interno}} x.f$$

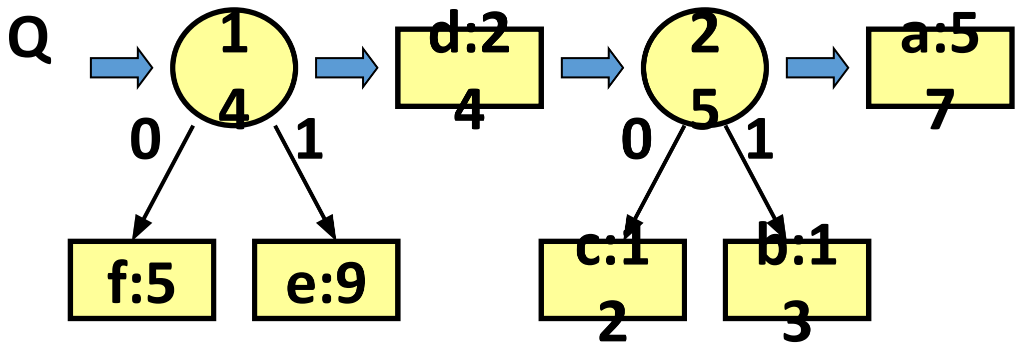


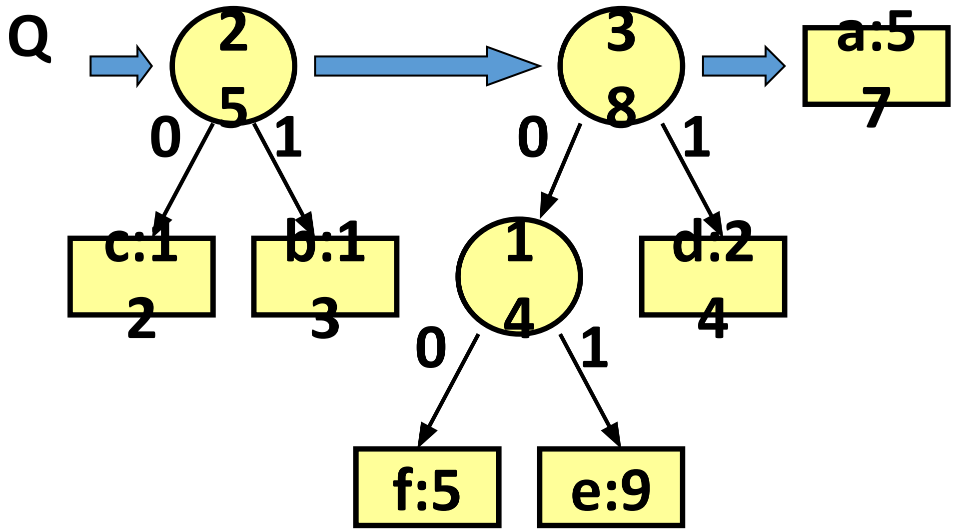
in cui la sommatoria è estesa alle frequenze ***x.f*** di tutti i nodi interni ***x*** dell’albero ***T.***

Costruzione dell’albero di Huffman:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5

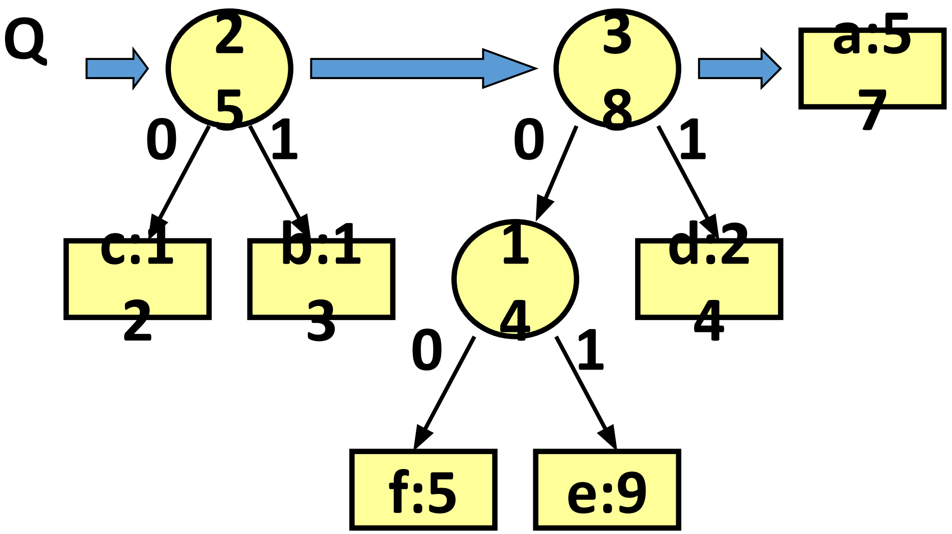


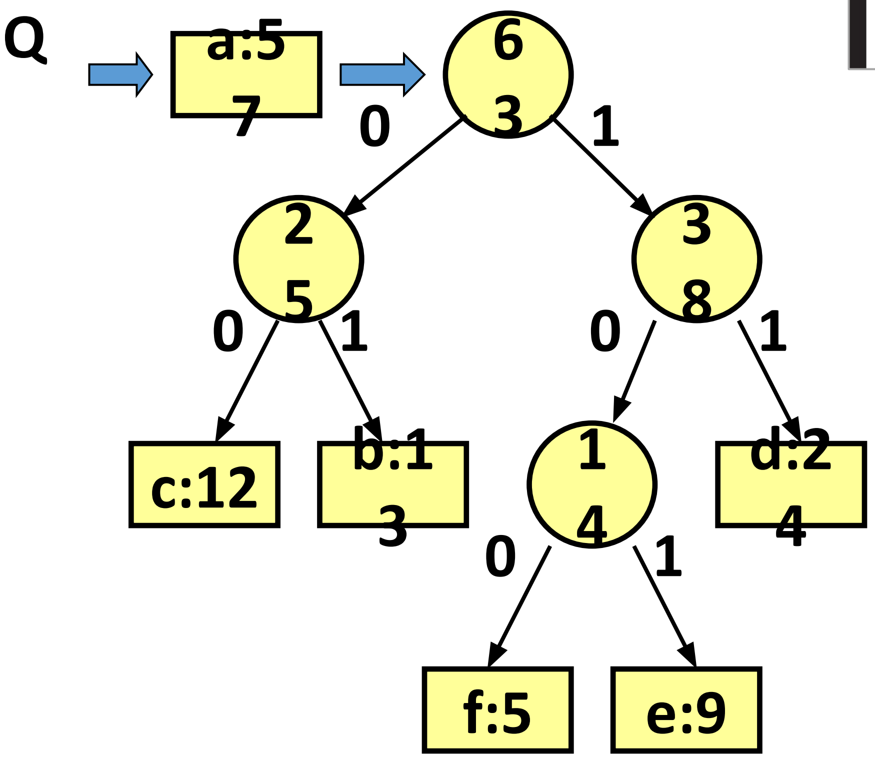

Q
1
4
d:2
4
2
5
a:5
7
0
1
0
1
f:5
e:9
c:1
2
b:1
3


Q
2
5
3
8
a:5
7
0
1
0
1
c:1
2
b:1
3
1
4
d:2
4
0
1
f:5
e:9

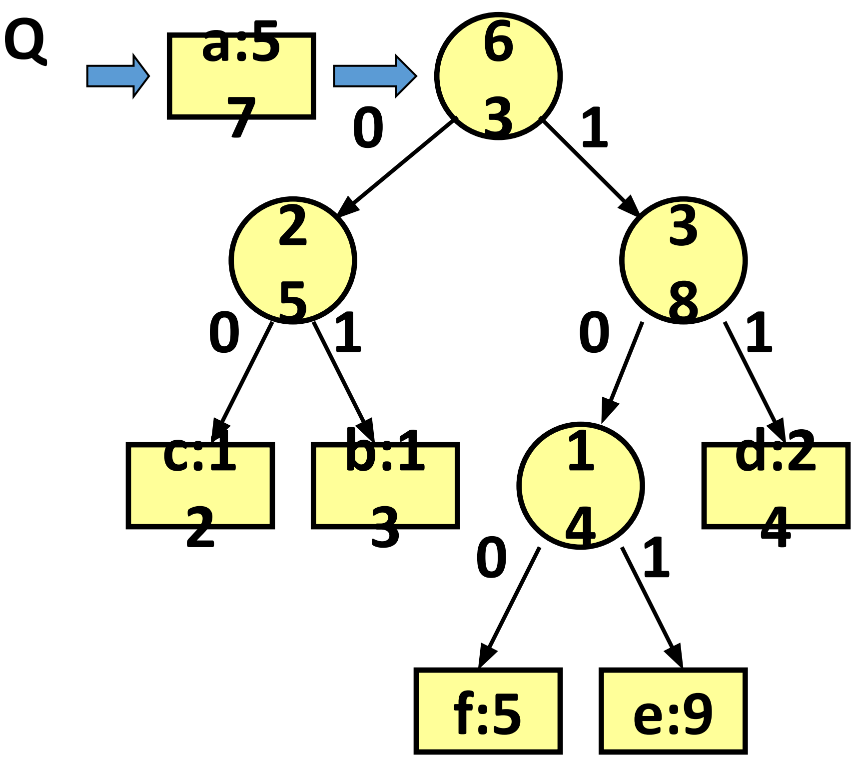

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein
Introduzione agli algoritmi e strutture dati
IV edizione

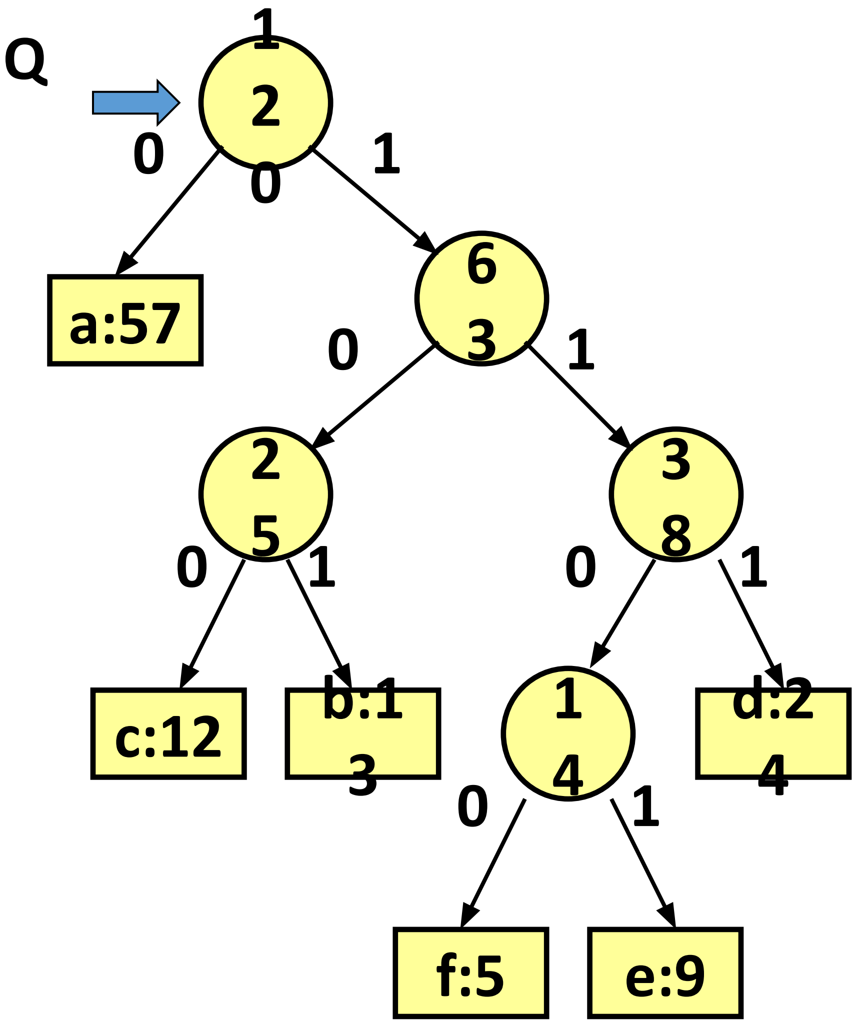

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein
Introduzione agli algoritmi e strutture dati
IV edizione
Edizione italiana a cura di Livio Colussi
Con la collaborazione di Achille Frigeri
Mc Graw Hill


Q
25
38
a:57
0
1
c:12
b:13
14
d:24
f:5
e:9


Q
a:57
63
0
1
25
38
c:12
b:13
14
d:24
f:5
e:9


Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein
Introduzione agli algoritmi e strutture dati
IV edizione
Edizione italiana a cura di Livio Colussi
Con la collaborazione di Achille Frigeri


Q
a:57
63
0
1
25
38
0
1
0
1
c:12
b:13
14
d:24
0
1
f:5
e:9


Q
120
0
1
a:57
63
0
1
25
38
0
1
0
1
c:12
b:13
14
d:24
0
1
f:5
e:9

Implementazione dell'algoritmo avido di Huffman, C è l'insieme dei caratteri, ognuno con la sua frequenza in un attributo *freq*.

```
Huffman(C)
    n = |C|, Q = C                  // Q coda di min-priorità
    for i = 1 to n-1                // costruita in tempo O(n)
        crea nuovo nodo z           // con Build-Min-Heap
        x = Extract-Min(Q)
        y = Extract-Min(Q)
        z.left = x
        z.right = y
        z.freq = x.freq + y.freq
        Insert(Q, z))
    return Extract-Min(Q)     // alla fine resta solo la radice
```

Assumendo che la coda ***Q*** venga realizzata con un min-heap, le operazioni ***Insert*** ed ***Extract-Min*** richiedono un tempo $\boldsymbol{O(\log n)}$.

Pertanto l'algoritmo richiede un tempo $\boldsymbol{O(n \log n)}$ (dove $\boldsymbol{n}$ è il numero di caratteri dell'alfabeto).

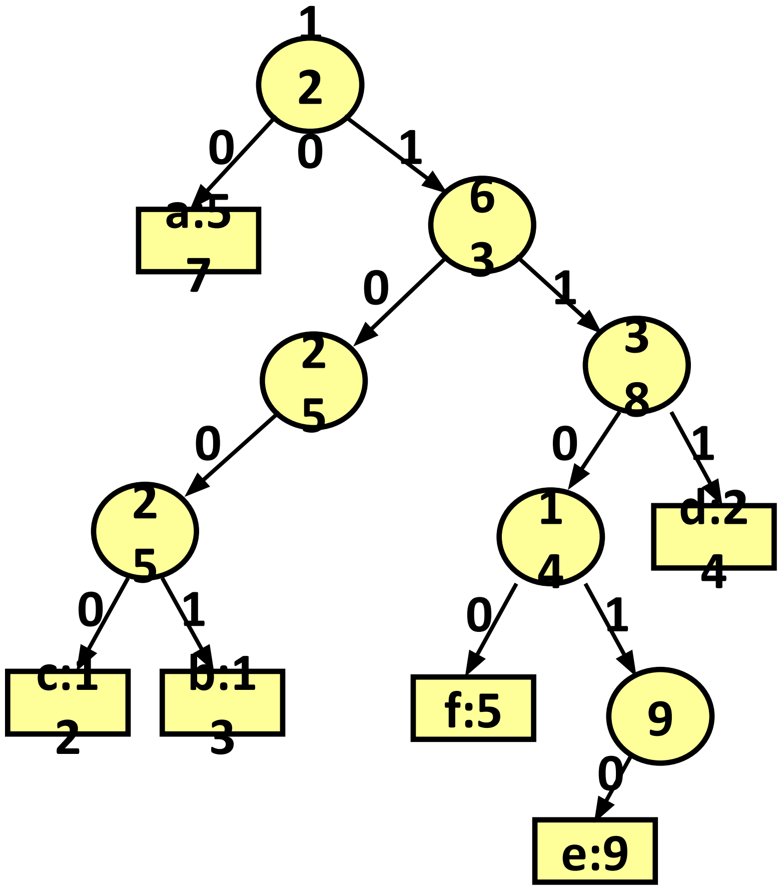
L'algoritmo è ***avido*** perché ad ogni passo costruisce il nodo interno avente frequenza minima possibile.

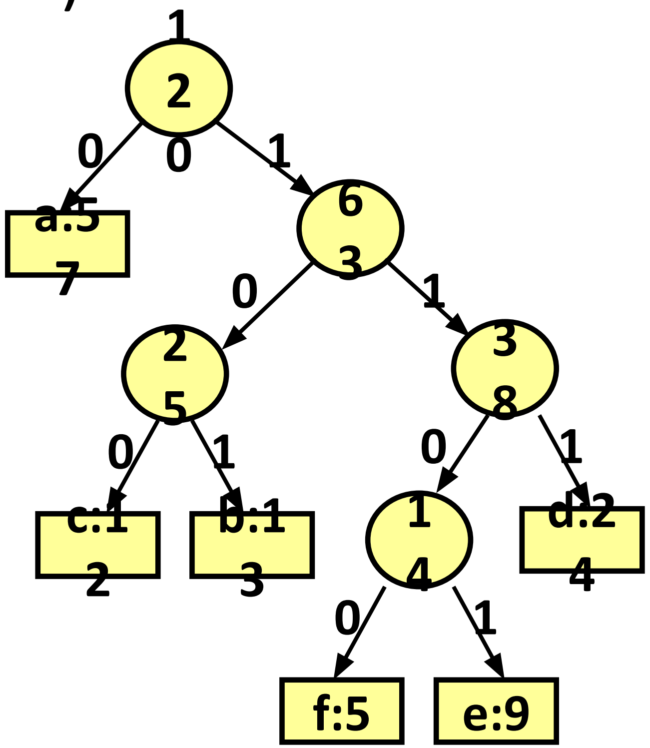
Ricordiamo infatti che

$$B(T) = \sum_{x \text{ nodo interno}} x.f$$

Siamo sicuri che in questo modo otteniamo sempre un codice ottimo?

Se ***T*** è ottimo ogni nodo interno ha due figli (altrimenti togliendo il nodo si otterrebbe un codice migliore)





Se ***T*** è ottimo esistono due foglie sorelle ***x*** ed ***y*** a profondità massima.

Proprietà (*sottostruttura ottima*)

Sia ***T*** l'albero di un codice prefisso ottimo per l'alfabeto **Σ** e siano ***a*** ed ***b*** i caratteri associati a due foglie sorelle ***x*** ed ***y*** di ***T***. Se consideriamo il padre ***z*** di ***x*** ed ***y*** come foglia associata ad un nuovo carattere ***c*** con frequenza

$$f_c = z.f = f_a + f_b$$

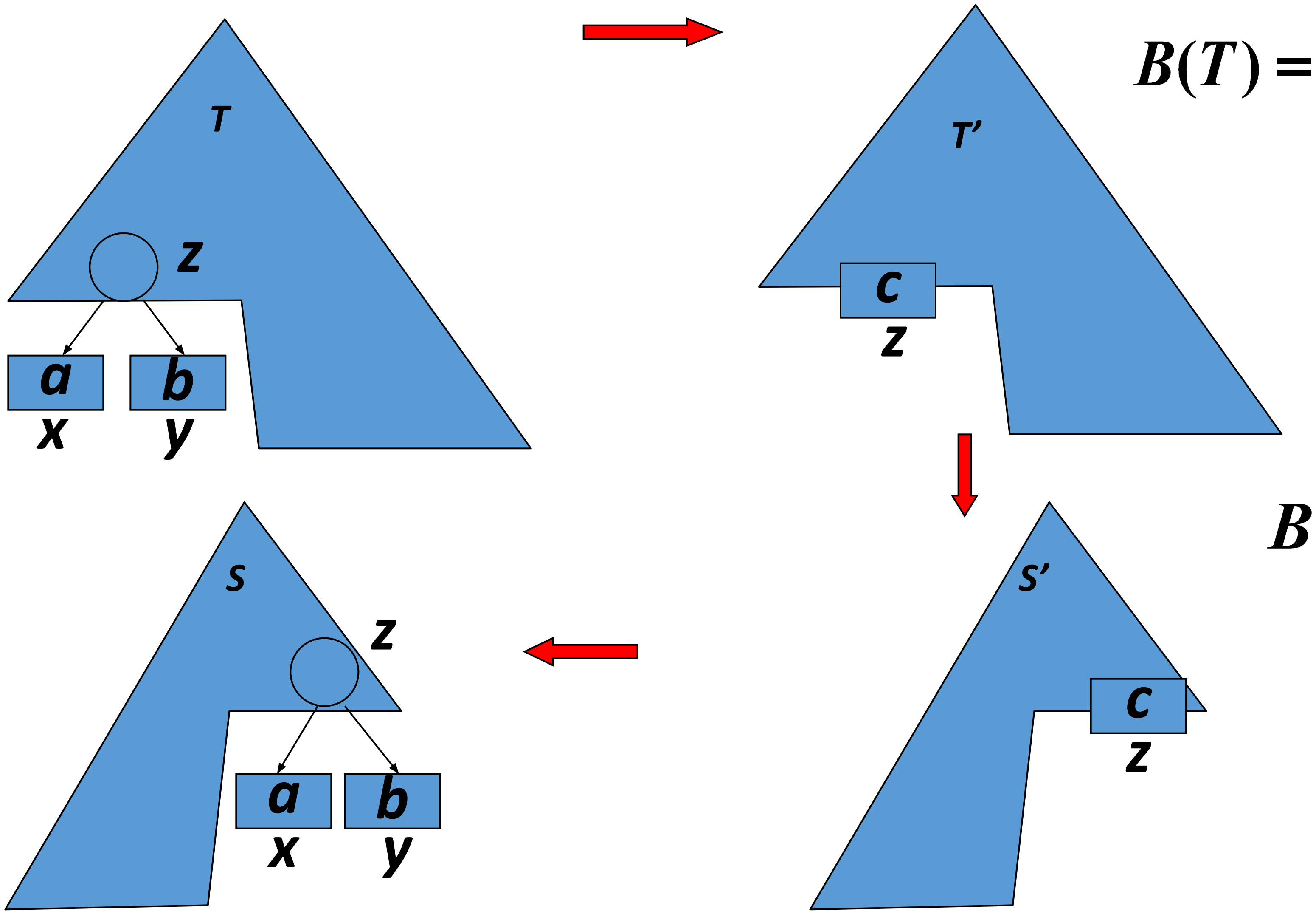
allora l'albero ***T'=T - {x,y}*** rappresenta un codice prefisso ottimo per l'alfabeto

$$\Sigma' = \Sigma - \{a,b\} \cup \{c\}$$

$$
\begin{aligned}
B(T) &= B(T') + f_a d_T(a) + f_b d_T(b) - f_c d_{T'}(c) \\
&= B(T') + (f_a + f_b)(d_{T'}(c) + 1) - (f_a + f_b) d_{T'}(c) \\
&= B(T') + f_a + f_b
\end{aligned}
$$

Supponiamo, per assurdo, esista un albero ***S'*** per ***Σ'*** tale che ***B*(*S'*) < *B*(*T'*)**.

Aggiungendo ad ***S'*** le foglie ***x*** ed ***y*** come figlie del nodo ***z*** (che in ***S'*** è una foglia) otterremmo un albero ***S*** per ***Σ*** tale che ***B*(*S*) < *B*(*T*)** contro l'ipotesi che ***T*** sia ottimo.



$$B(T) = B(T') + f_a + f_b$$

$$B(S') < B(T')$$

$$B(S) = B(S') + f_a + f_b < B(T') + f_a + f_b = B(T)$$

Proprietà (*scelta avida*)

Siano ***a*** e ***b*** due caratteri di **Σ** aventi frequenze f_a ed f_b minime

Esiste un codice prefisso ottimo in cui le parole codice di ***a*** e ***b*** hanno uguale lunghezza e differiscono soltanto per l'ultimo bit.

Se i codici di ***a*** e ***b*** differiscono soltanto per l'ultimo bit, nell'albero del codice le foglie ***a*** e ***b*** sono figlie dello stesso nodo, cioè sorelle.

Attenzione: la proprietà ***non*** dice che ciò è vero per ogni codice prefisso ottimo e ***tanto meno*** che se ciò è vero il codice è ottimo.

Dice ***solo*** che ciò è vero per ***almeno*** un codice ottimo.

Sappiamo che in ***T*** esistono due foglie sorelle a profondità massima.

Siano ***c*** e ***d*** i caratteri di tali foglie.

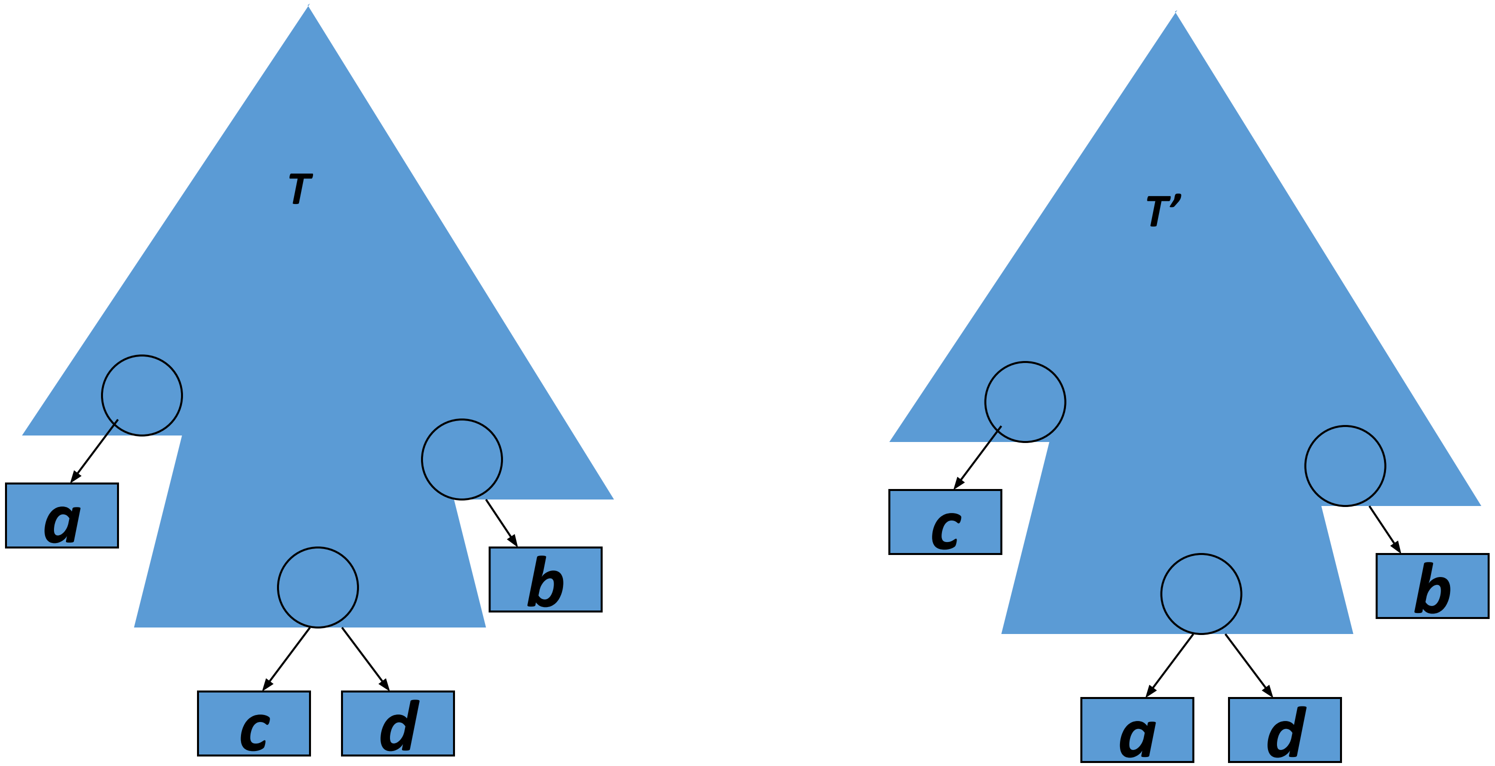
Mostriamo che scambiando ***c*** e ***d*** con ***a*** e ***b*** il codice rimane ottimo.

Possiamo supporre $f_c \leq f_d$ ed $f_a \leq f_b$.

a e ***b*** sono i due caratteri con frequenza minima in assoluto e quindi $f_a \leq f_c$ ed $f_b \leq f_d$.

Sia T' ottenuto da T scambiando la foglia c con la foglia a (con ricalcolo delle frequenze dei nodi interni)

$$f_a \leq f_c \text{ ed } f_b \leq f_d.$$



Allora:

$$
\begin{aligned}
B(T) - B(T') &= \sum_{k \in \Sigma} f_k d_T(k) - \sum_{k \in \Sigma} f_k d_{T'}(k) \\
&= f_a d_T(a) + f_c d_T(c) - f_a d_{T'}(a) - f_c d_{T'}(c) \\
&= f_a d_T(a) + f_c d_T(c) - f_a d_T(c) - f_c d_T(a) \\
&= [f_c - f_a][d_T(c) - d_T(a)] \\
&\geq 0
\end{aligned}
$$

Siccome ***T*** è ottimo ***B*(*T*) = *B*(*T'*)** e quindi anche ***T'*** è ottimo.

Scambiando poi le foglie ***d*** e ***b*,** si ottiene ancora un albero ottimo ***T''*** in cui ***a*** e ***b*** sono foglie sorelle.

Teorema. L'algoritmo di Huffman produce un codice prefisso ottimo

```
Huffman(C)
  n = |C|, Q = C                  // Q coda di min-priorità
  for i = 1 to n-1                // costruita in tempo O(n)
     crea nuovo nodo z            // con Build-Min-Heap
     x = Extract-Min(Q)
     y = Extract-Min(Q)
     z.left = x
     z.right = y
     z.freq = x.freq + y.freq
     Insert(Q, z))
  return Extract-Min(Q)     // alla fine resta solo la radice
```

Conseguenza della sottostruttura ottima e della proprietà della scelta avida

__Esercizio__ __1.__ Problema dello "zaino" frazionario. Dati n tipi di merce $M_1,\ldots,M_n$ in quantità rispettive $q_1,\ldots,q_n$ e con costi unitari $c_1,\ldots,c_n$, si vuole riempire uno zaino di capacità Q in modo che il contenuto abbia costo massimo. Mostrare che il seguente algoritmo risolve il problema:

```
Riempi-Zaino(q, c, n, Q)    // c1 ≥ c2 ≥ ... ≥ cn
  Spazio = Q
  for i = 1 to n
     if Spazio ≥ q[i]
          z[i] = q[i], Spazio = Spazio – z[i]
     else z[i] = Spazio, Spazio = 0
  return z
```

Esercizio 2. Problema dello "zaino" 0-1.

Sono dati n tipi di oggetti $O_1,\ldots,O_n$ in numero illimitato. Un oggetto di tipo O_i occupa un volume v_i e costa c_i.

Si vuole riempire uno zaino di capacità Q in modo che il contenuto abbia costo massimo. Mostrare che il seguente algoritmo *non* risolve il problema.

Riempi-Zaino(v, c, n, Q) // $c_1/v_1 \geq c_2/v_2 \geq \ldots \geq c_n/v_n$

  $Spazio = Q$
  for $i = 1$ to n
    $z[i] = \lfloor Spazio/v[i] \rfloor$
    $Spazio = Spazio - z[i]v[i]$
  return z

Esercizio 3

Siano $\boldsymbol{a_1,\ldots,a_n}$ attività didattiche aventi tempi di inizio $\boldsymbol{s_1,\ldots,s_n}$ e tempi di fine $\boldsymbol{f_1,\ldots,f_n}$ e supponiamo di avere un insieme sufficiente di aule in cui svolgerle.

Trovare un algoritmo per programmare tutte le attività usando il minimo numero possibile di aule.

Esercizio 4

Siano $a_1,\dots,a_n$ attività didattiche aventi tempi di inizio $s_1,\dots,s_n$ e tempi di fine $f_1,\dots,f_n$ e abbiamo a disposizione m aule $A_1,\dots, A_m$

Trovare un algoritmo avido per programmare il massimo numero di attività nelle m aule.

Esercizio 5

Dimostrare che ogni algoritmo di compressione che accorcia qualche sequenza di bit deve per forza allungarne qualche altra.

<u>Dimostrazione</u>

Supponiamo per assurdo che l'algoritmo accorci qualche sequenza ma non ne allunghi nessuna.

Sia ***x*** la più corta sequenza che viene accorciata dall'algoritmo e sia ***m*** la sua lunghezza.

Le sequenze di lunghezza minore di ***m*** sono $\mathbf{2^{m}-1}$ e non vengono accorciate o allungate.

Ognuna di esse viene codificata con una sequenza diversa e quindi le loro codifiche sono anch’esse $\mathbf{2^m-1}$ e sono tutte di lunghezza minore di ***m.***

Dunque ognuna delle $\mathbf{2^m-1}$ sequenze più corte di ***m*** è codifica di qualche sequenza più corta di ***m.***

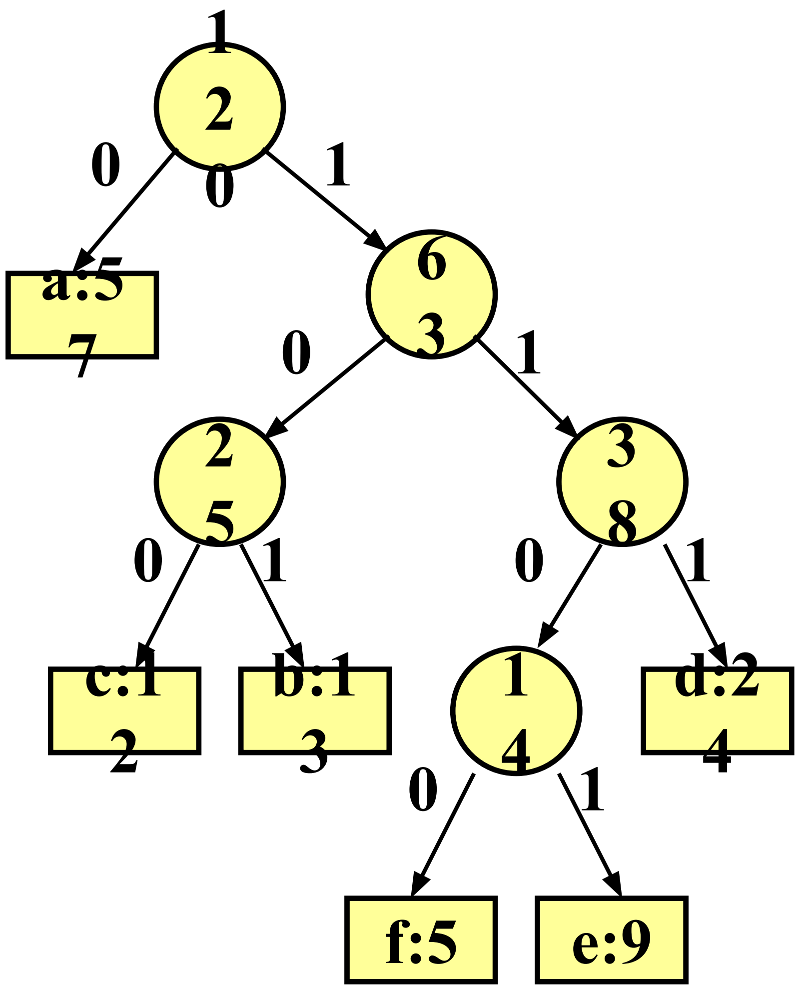
Dunque la codifica di ***x*** coincide con la codifica di un’altra sequenza: assurdo.

Esercizio 6. Sia $\boldsymbol{\Sigma} = \{c_1, \ldots, c_n\}$ un insieme di caratteri ed $f_1, \ldots, f_n$ le loro frequenze

Rappresentare un codice prefisso ottimo per $\boldsymbol{\Sigma}$ con una sequenza di

$$2n - 1 + n \lceil \log n \rceil \text{ bit}$$

Suggerimento: usare $2n - 1$ bit per la struttura dell'albero ed $n \lceil \log n \rceil$ bit per elencare i caratteri nell'ordine in cui compaiono nelle foglie (usando il codice del file non compresso)

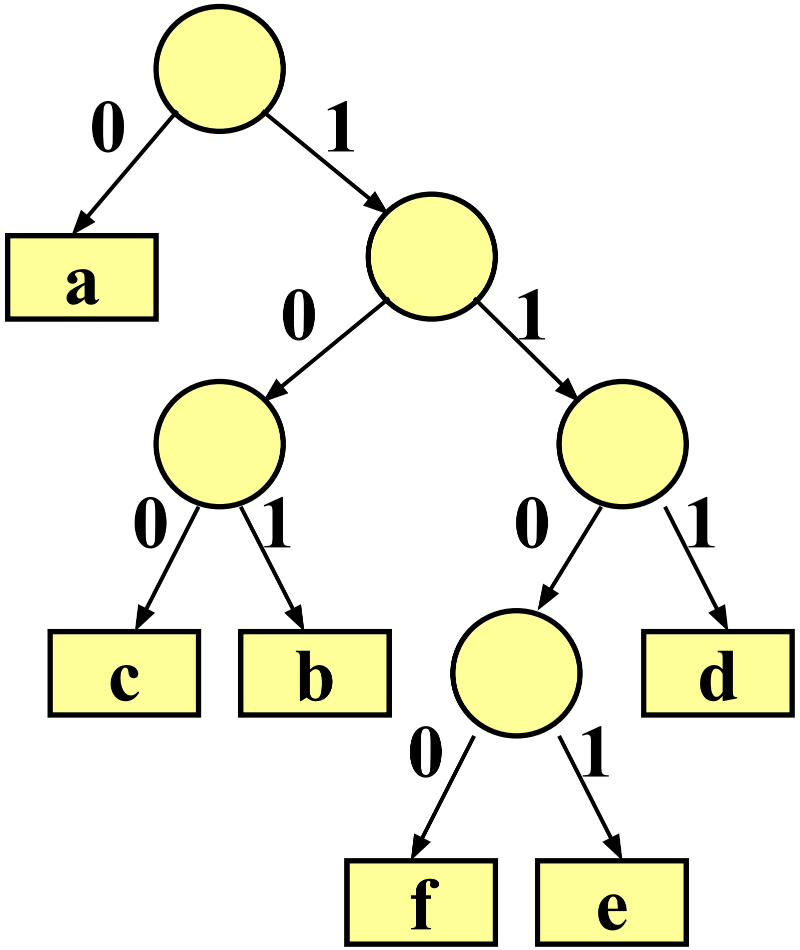


Parti dalla radice e ripeti:

1. se il nodo è interno metti uno **0** e scendi sul figlio sinistro;
2. se è una foglia metti un **1** e risali verso sinistra finché puoi: se arrivi alla radice fermati altrimenti risali di un passo verso destra e scendi sul figlio destro

01001100111

01001100111



Crea la radice e ripeti:

1. se incontri uno **0** crea il figlio sinistro e scendi su tale figlio;
2. se incontri un **1** risali verso sinistra finché puoi: se arrivi alla radice fermati altrimenti risali di un passo verso destra, crea un figlio destro e scendi su tale figlio

Esercizio 7.

Un cassiere vuole dare un resto di ***n*** centesimi di euro usando il minimo numero di monete.

a. Descrivere un algoritmo avido per fare ciò con tagli da **1¢**, **2¢**, **5¢**, **10¢**, **20¢**, **50¢**, **1€** e **2€**.

b. Dimostrare che l'algoritmo avido funziona anche con monete di tagli $\mathbf{1, c, c^2, \ldots, c^k}$ dove $\mathbf{c > 1}$ e $\mathbf{k \geq 0}$.

c. Trovare un insieme di tagli di monete per i quali l'algoritmo avido non funziona.