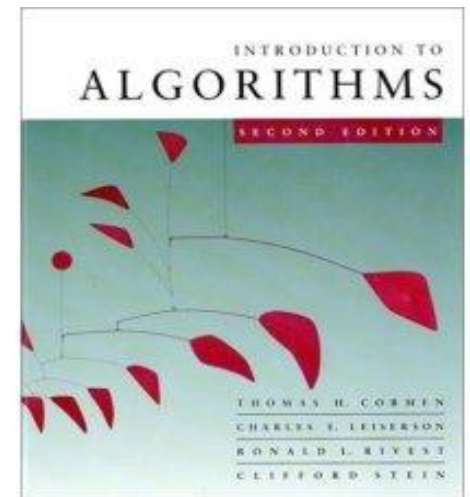




Algoritmi greedy

Algoritmi, Strutture Dati e Calcolo Parallelo

- ❑ T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein Introduction to Algorithms, Second Edition
- ❑ Queste trasparenze sono disponibili su <http://dei.polimi.it/upload/loiacono>
- ❑ Materiale rilasciato con licenza Creative Commons Attribution-NonCommercial-ShareAlike License (<http://creativecommons.org/licenses/by-nc-sa/2.5/>)



- ❑ Gli algoritmi greedy risolvono problemi di ottimizzazione che comportano una sequenza di decisioni
- ❑ Come funzionano?
 - ▶ Ad ogni passo prendono la decisione che **sembra** ottima (ovvero, è localmente ottima)
 - ▶ Applicano il medesimo principio al sottoproblema rimanente (cioè quello che si ottiene dopo aver preso la decisione localmente ottima)
 - ▶ **Sperabilmente**, trovano un **ottimo globale**
- ❑ Quando possiamo applicarli?
 - ▶ Quando è possibile **dimostrare** che esiste una **scelta ingorda**: cioè fra tutte le scelte ne esiste una che può essere **facilmente** individuata e porta sicuramente alla soluzione ottima”
 - ▶ Quando il problema ha **sottostruttura ottima**: dopo aver preso una decisione, si ottengono uno o più sottoproblemi con la stessa struttura di quello originale
- ❑ In alcuni casi, anche soluzioni non ottime possono essere comunque interessanti

Problema della selezione di attività

□ Problema:

- ▶ Siano date n attività in competizione tra loro per l'utilizzo di una certa risorsa
- ▶ Trovare l'allocazione ottimale della risorsa, cioè il più grande sottoinsieme di attività che possono condividere la risorsa senza creare conflitti

□ Definizioni:

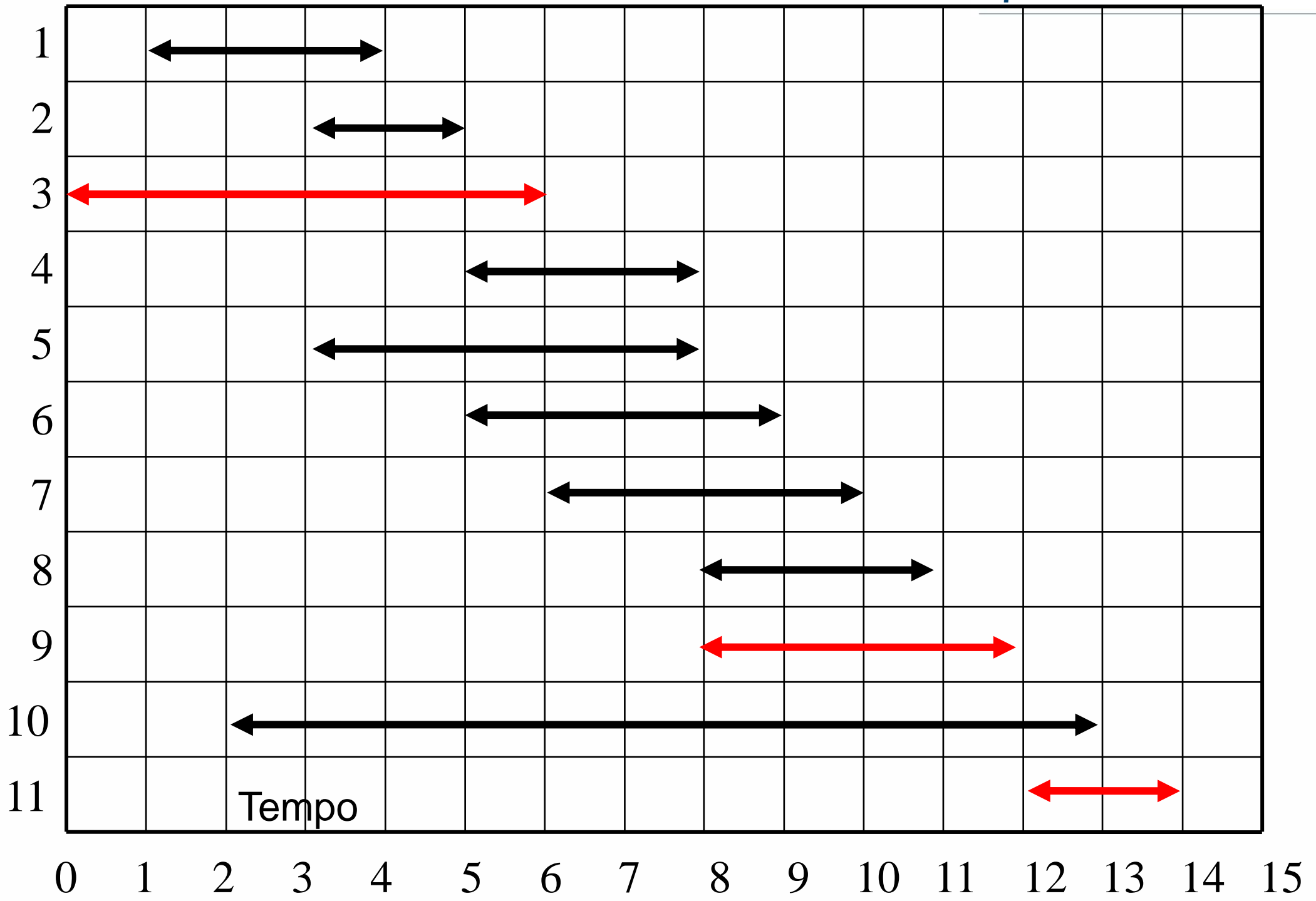
- ▶ $S = \{ 1, 2, \dots, n \}$ un insieme di attività
- ▶ Ad ogni attività i in S sono associati:
 - $s[i]$ → tempo di inizio
 - $f[i]$ → tempo di fine
 - $s[i]$ e $f[i]$ sono interi non negativi

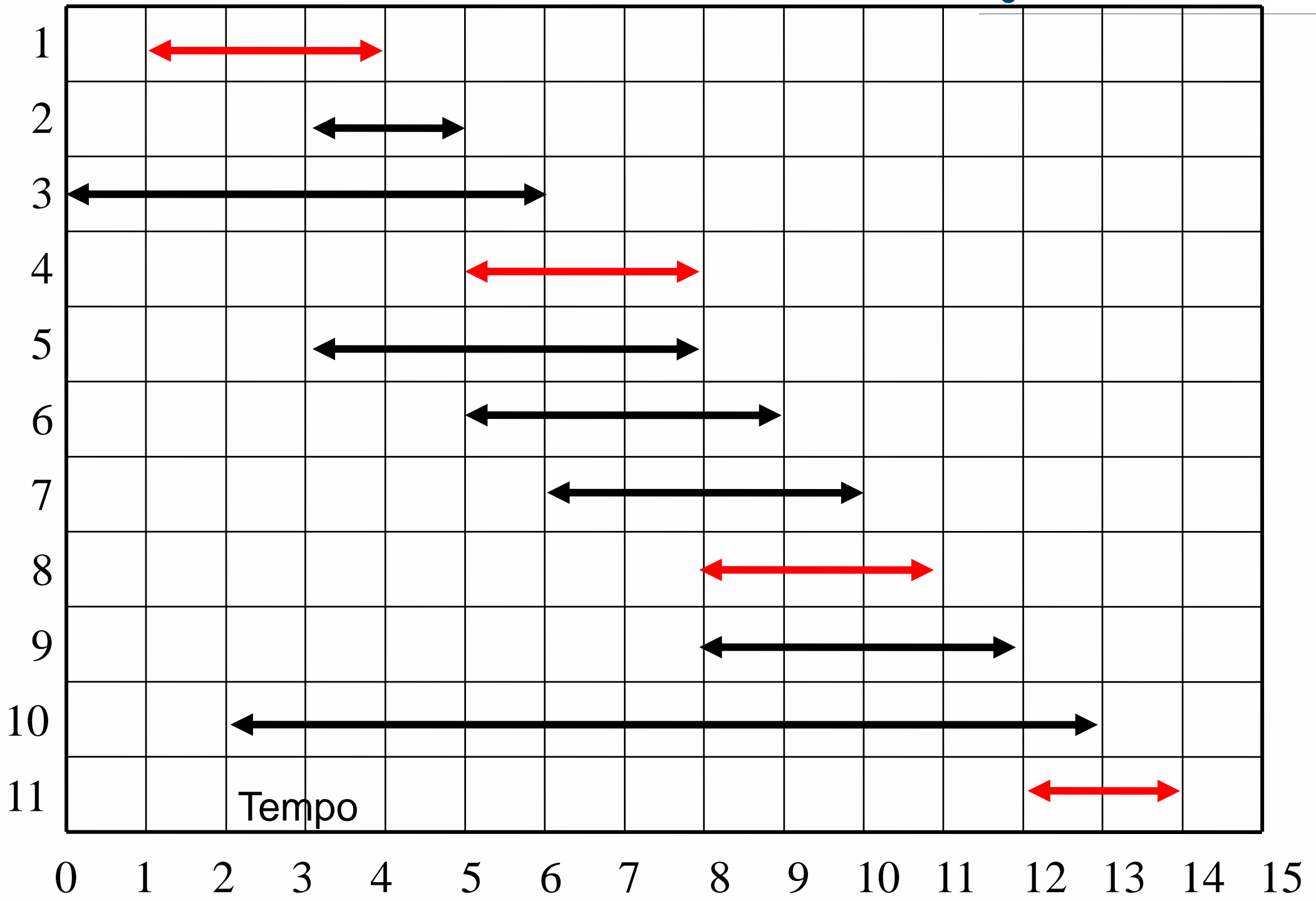
i	$s[i]$	$f[i]$
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	12	14

- Due attività i e j si dicono *compatibili* se gli intervalli $[s[i],f[i]]$ e $[s[j],f[j]]$ sono *disgiunti*, cioè non si sovrappongono
 - ▶ $f[j] \leq s[i]$ oppure $f[i] \leq s[j]$



- Problema della selezione delle attività
 - ▶ Dato l'insieme di attività $S = \{ 1, 2, \dots, n \}$, trovare il più grande sottoinsieme $A \subseteq S$ tale che tutte le attività in esso contenute sono compatibili due a due
- Soluzione "brute-force": $\Theta(2^n)$





- Si assuma che le attività siano ordinate per tempo di fine:

- ▶ $f[1] \leq f[2] \leq \dots \leq f[n]$

- **Sottoproblema $S[i,j]$**

- **$S[i,j] = \{ k \mid f[i] \leq s[k] < f[k] \leq s[j] \}$**

Il sottoinsieme di attività che

- ▶ iniziano dopo che i ha finito
 - ▶ finiscono prima che j abbia iniziato



- Aggiungiamo due attività "fittizie":

- ▶ $0 \rightarrow f[0] = 0$

- ▶ $n+1 \rightarrow s[n+1] = +\infty$

- Così $S[0,n+1]$ corrisponde al problema completo S

- Sia $A[i,j]$ è una soluzione ottimale di $S[i,j]$, e l'attività k sia inclusa in $A[i,j]$; allora:
- Il problema $S[i,j]$ viene suddiviso in due sottoproblemi:
 - $S[i,k]$: le attività di $S[i,j]$ che **finiscono prima di k**
 - $S[k,j]$: le attività di $S[i,j]$ che **iniziano dopo di k**
- i. $A[i,j]$ contiene le soluzioni ottimali di $S[i,k]$ e $S[k,j]$
 - $A[i,k] = A[i,j] \cap S[i,k]$ è la soluzione ottimale di $S[i,k]$
 - $A[k,j] = A[i,j] \cap S[k,j]$ è la soluzione ottimale di $S[k,j]$
- Dimostrazione (metodo cut-and-paste)
 - ▶ Sia $A'[i,k]$ una soluzione ottima di $S[i,k]$ tale che $|A[i,k]| < |A'[i,k]|$
 - ▶ Allora $A'[i,j] = A[i,j] - A[i,k] \cup A'[i,k]$ è tale che $|A[i,j]| < |A'[i,j]|$
 \Rightarrow Assurdo: $A[i,j]$ è una soluzione ottima

- ❑ Descrizione ricorsiva della soluzione:
 - ▶ $A[i,j] = A[i,k] \cup \{ k \} \cup A[k, j]$
 - ▶ Come determinare k ? Analizzando tutte le possibilità...
- ❑ Definizione: $c[i, j]$
 - ▶ La **dimensione** del più grande sottoinsieme $A[i,j] \subseteq S[i,j]$ di attività mutualmente compatibili
- ❑ Come si calcola?

$$c[i, j] = \begin{cases} 0 & \text{se } S[i, j] = \emptyset \\ \max_{i < k < j} \{c[i, k] + c[k, j] + 1\} & \text{se } S[i, j] \neq \emptyset \end{cases}$$

- ❑ La soluzione appena descritta ha complessità $\theta(n^3)$
 - ▶ Devo risolvere tutti i sottoproblemi con $i \neq j$, nel caso peggiore ci metterò tempo $O(n)$ per un sottoproblema
- ❑ Posso fare di meglio?
 - ▶ Riflettiamo meglio sul problema....
 - ▶ Siamo sicuri che sia necessario analizzare *tutti* i possibili valori per k ?

- Sia $S[i,j]$ un sottoproblema non vuoto, e m l'attività di $S[i,j]$ con il **minor tempo di fine**; allora
 - ▶ m è compresa in qualche soluzione ottima di $S[i,j]$
 - ▶ Il sottoproblema $S[i,m]$ è vuoto
- Conseguenze:
 - ▶ **Non è più necessario analizzare tutti i possibili valori di k**
 - Faccio una scelta “ingorda”, ma sicura: seleziono l'attività m con il minor tempo di fine
 - ▶ **Non è più necessario analizzare due sottoproblemi:**
 - Elimino tutte le attività che non sono compatibili con la scelta ingorda
 - Mi resta solo un sottoproblema da risolvere: $S[m,j]$
- **Esercizio: dimostrare con il metodo cut-and-paste che esiste almeno una soluzione ottima che contiene la scelta greedy**

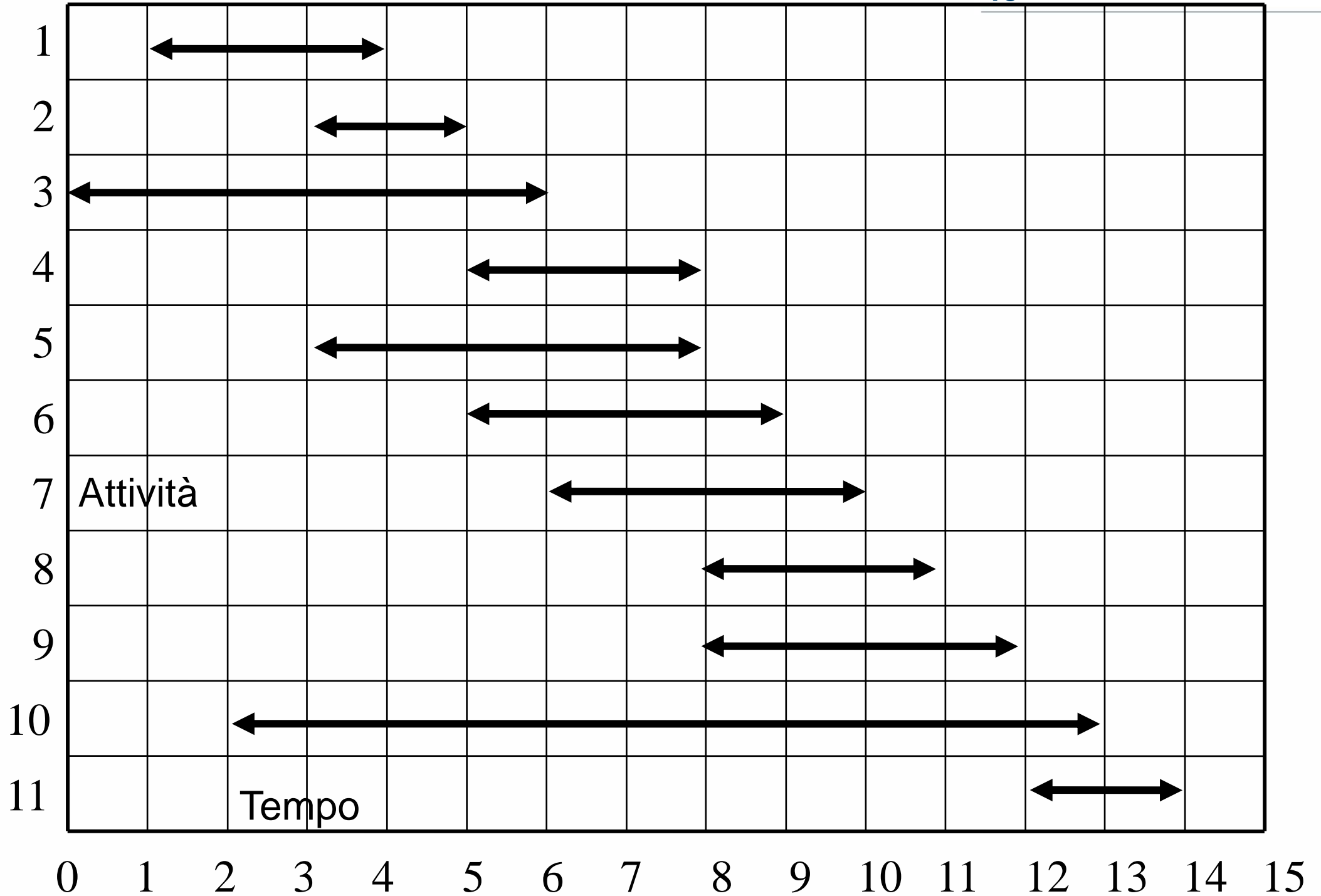
```
RGSA(s[0..n], f[0..n], i, len)  
  m := i+1  
  while m ≤ len and s[m] < f[i] do m := m+1  
  if m ≤ len then  
    return { m } ∪ RGSA(s, f, m, len)  
  else  
    return ∅
```

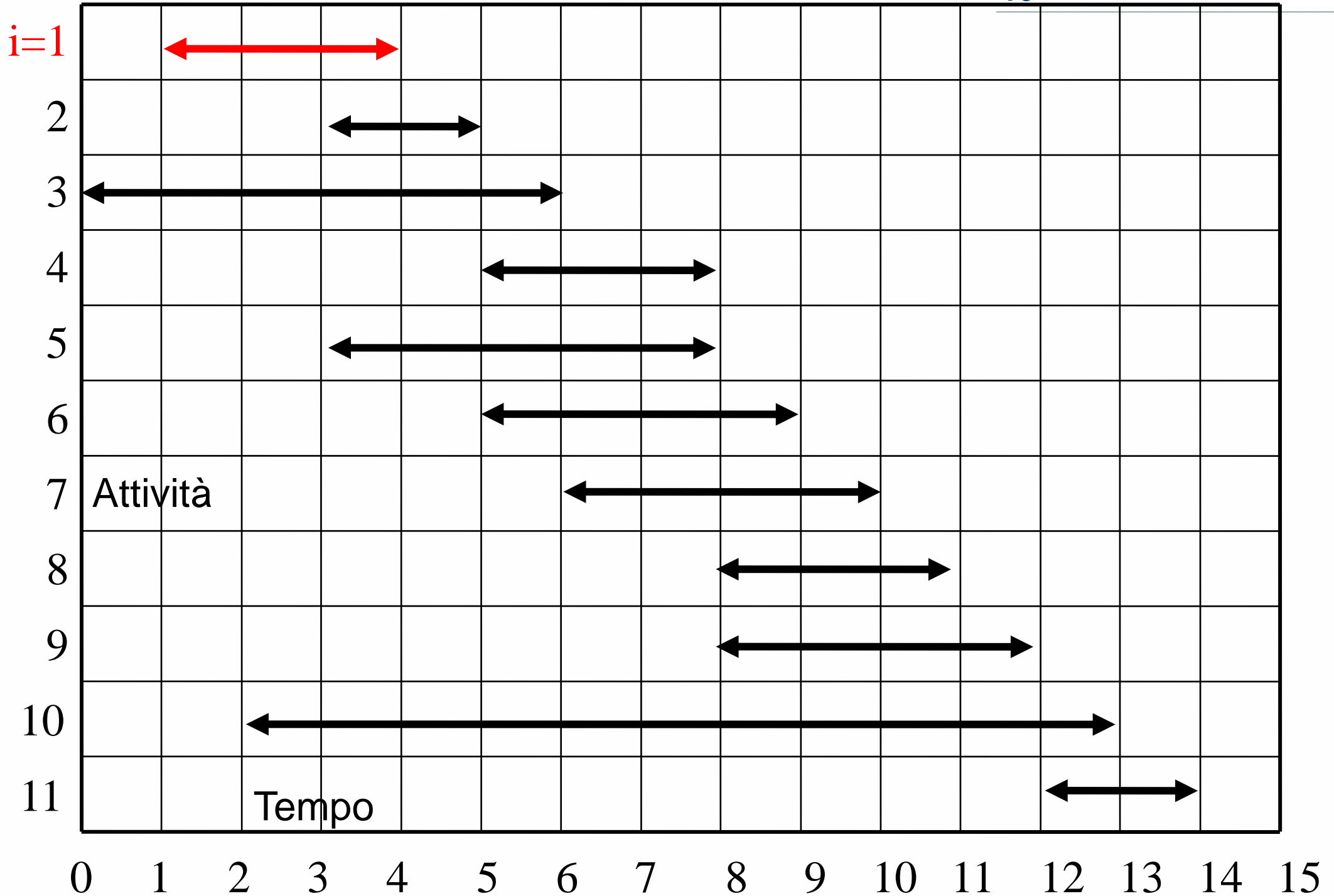
□ Descrizione:

- ▶ Chiamata iniziale: $RGSA(s, f, 0, n)$
- ▶ Cerca il nuovo valore di m : il primo compatibile con i
- ▶ Chiama ricorsivamente $RGSA()$ e unisci la soluzione a m

$i=0$

15



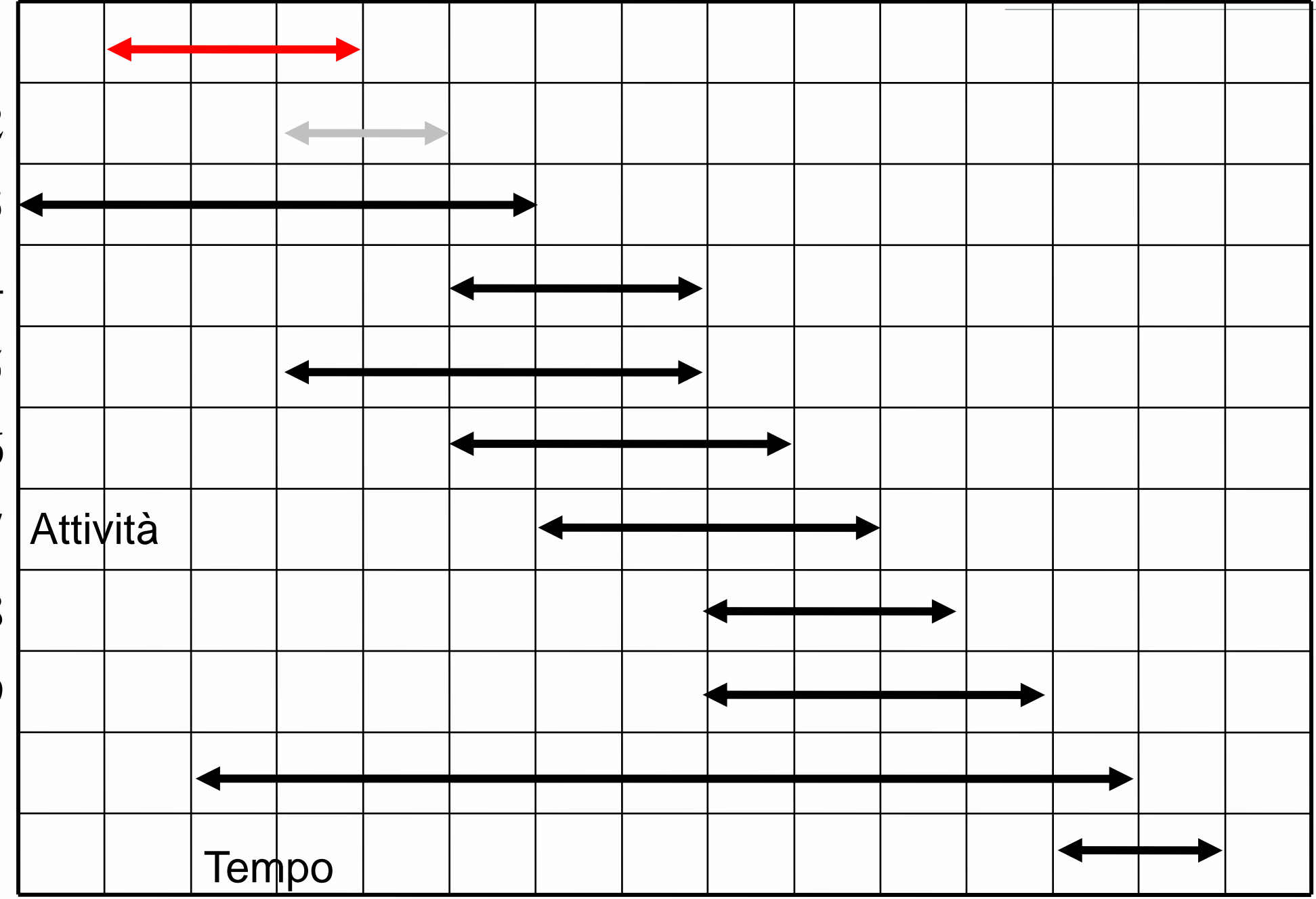


$i=1$

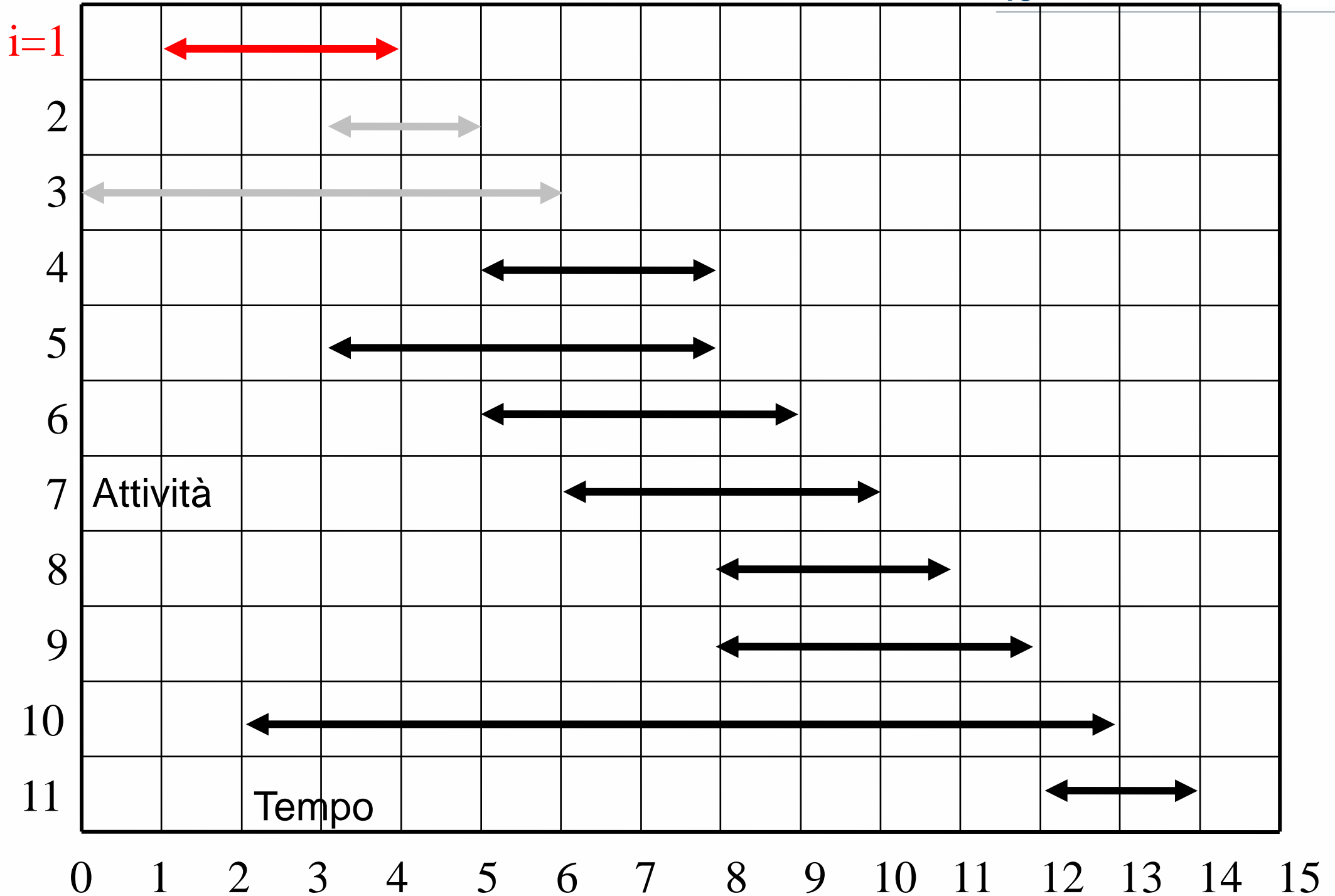
2
3
4
5
6
7
8
9
10
11

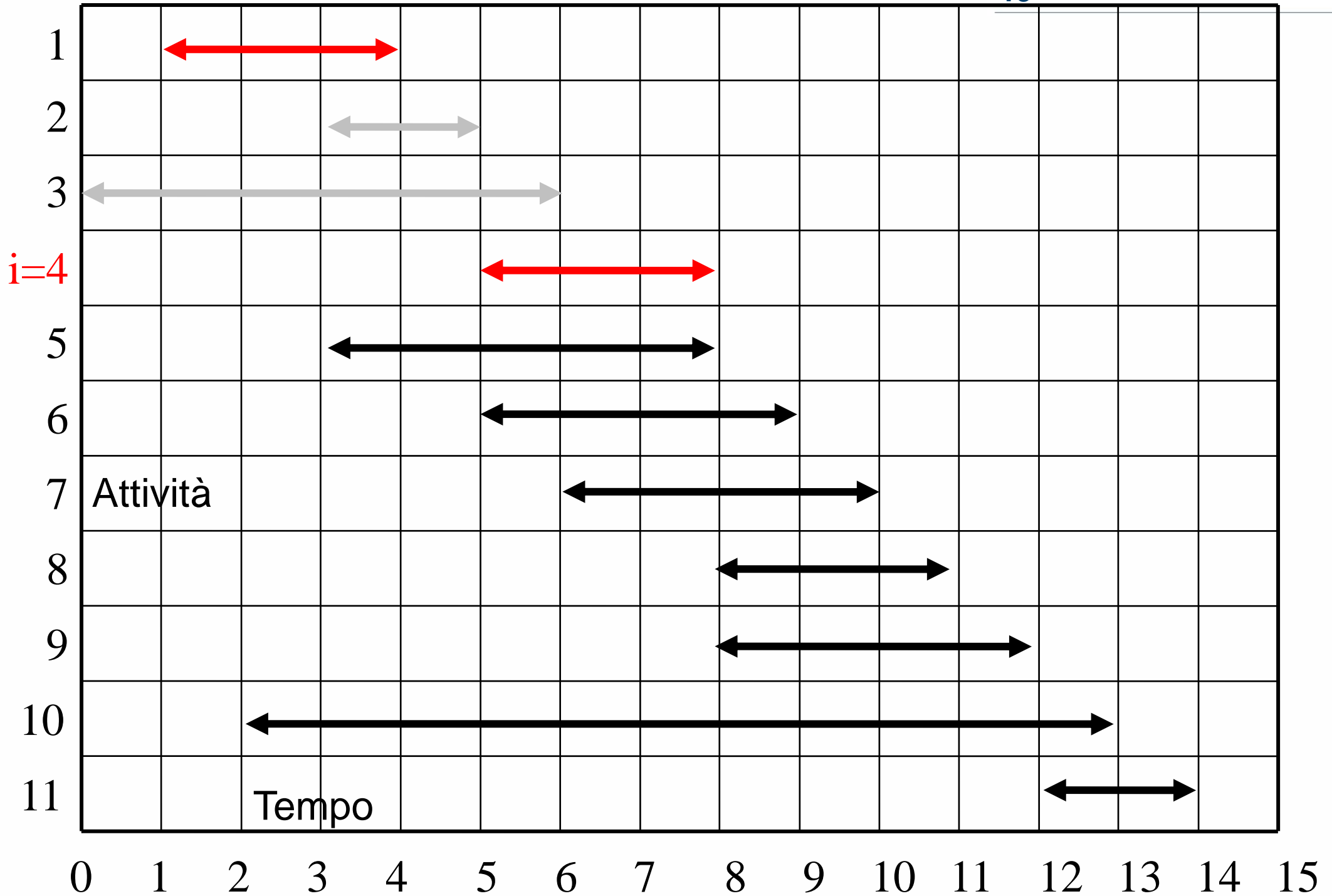
Attività

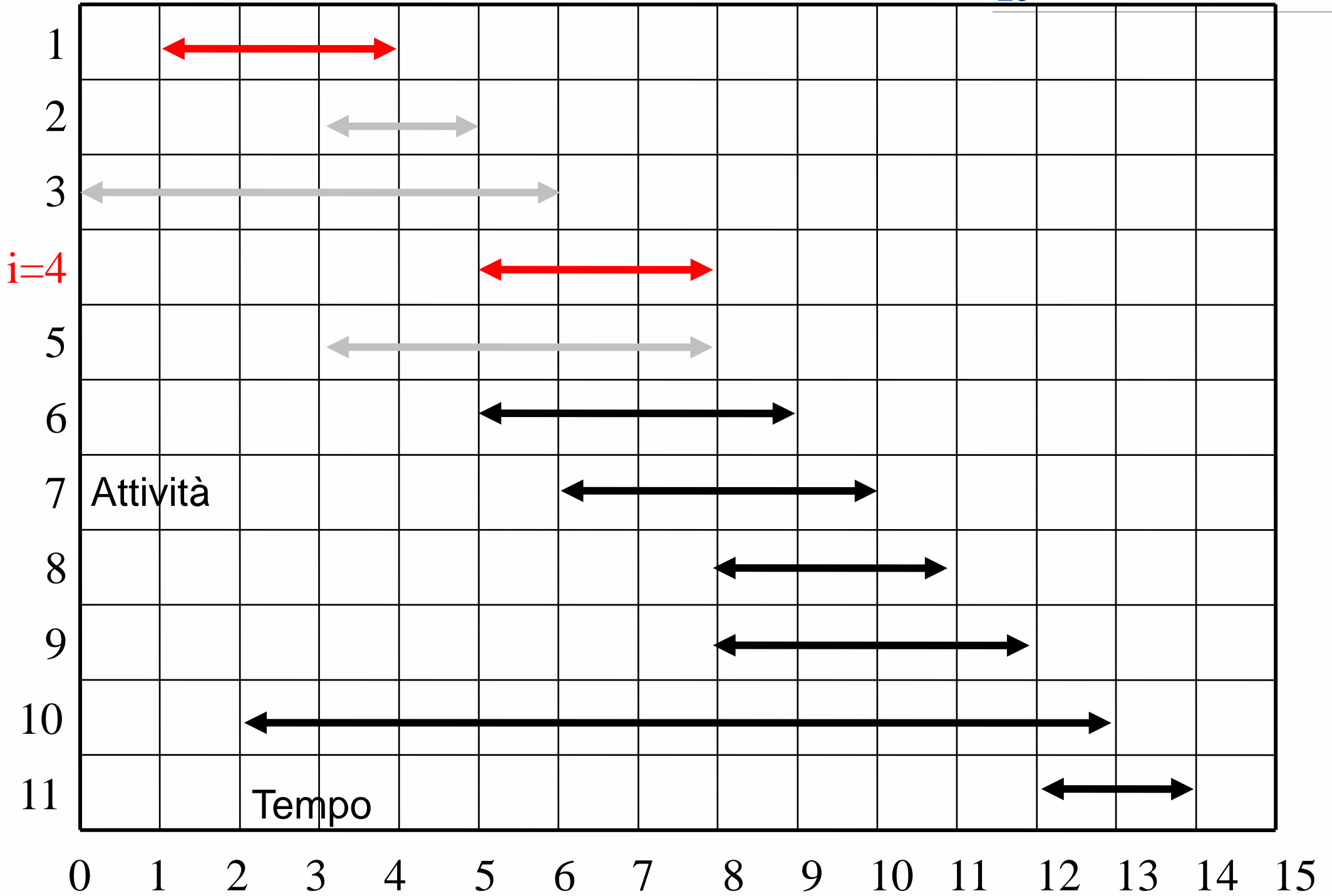
Tempo

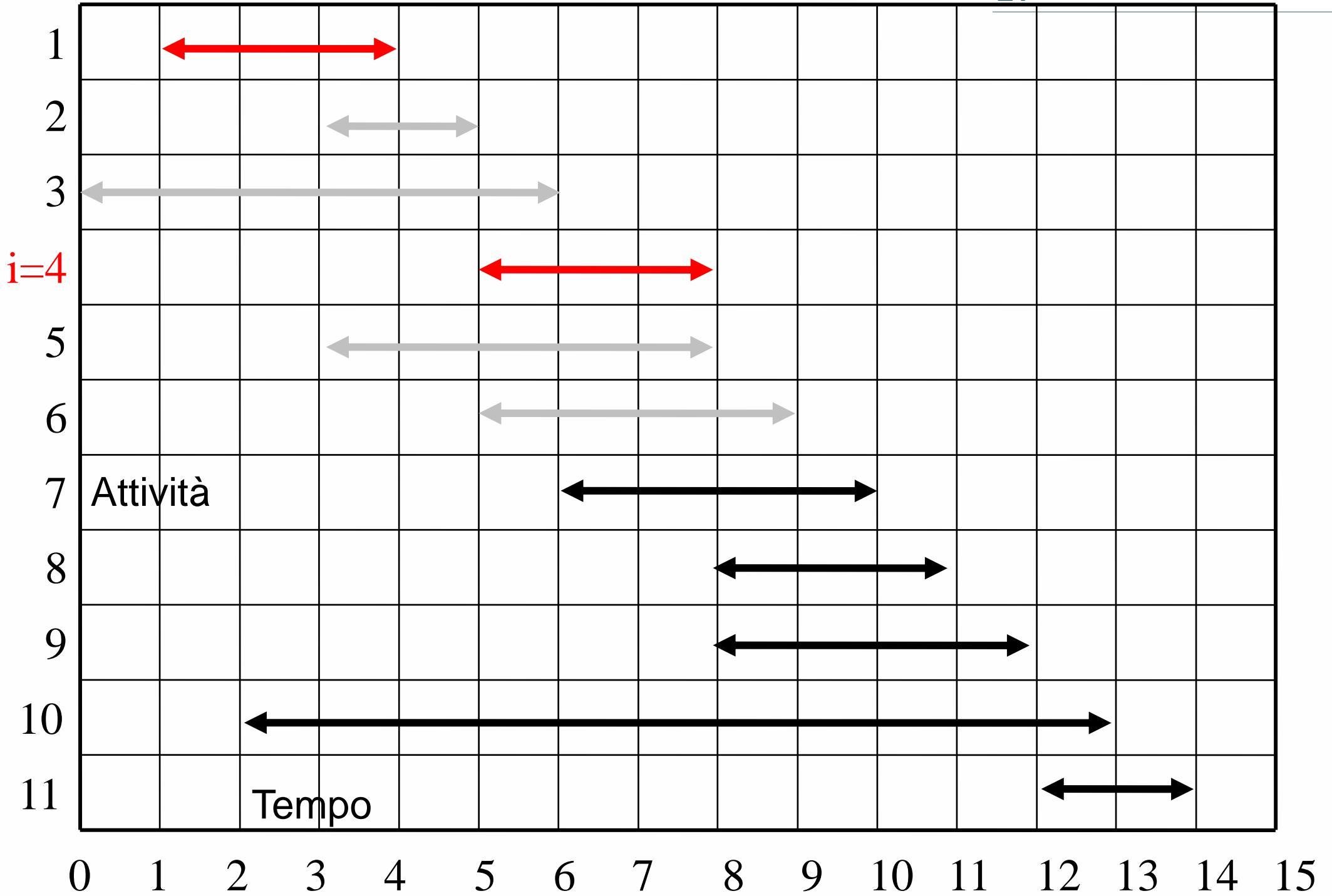


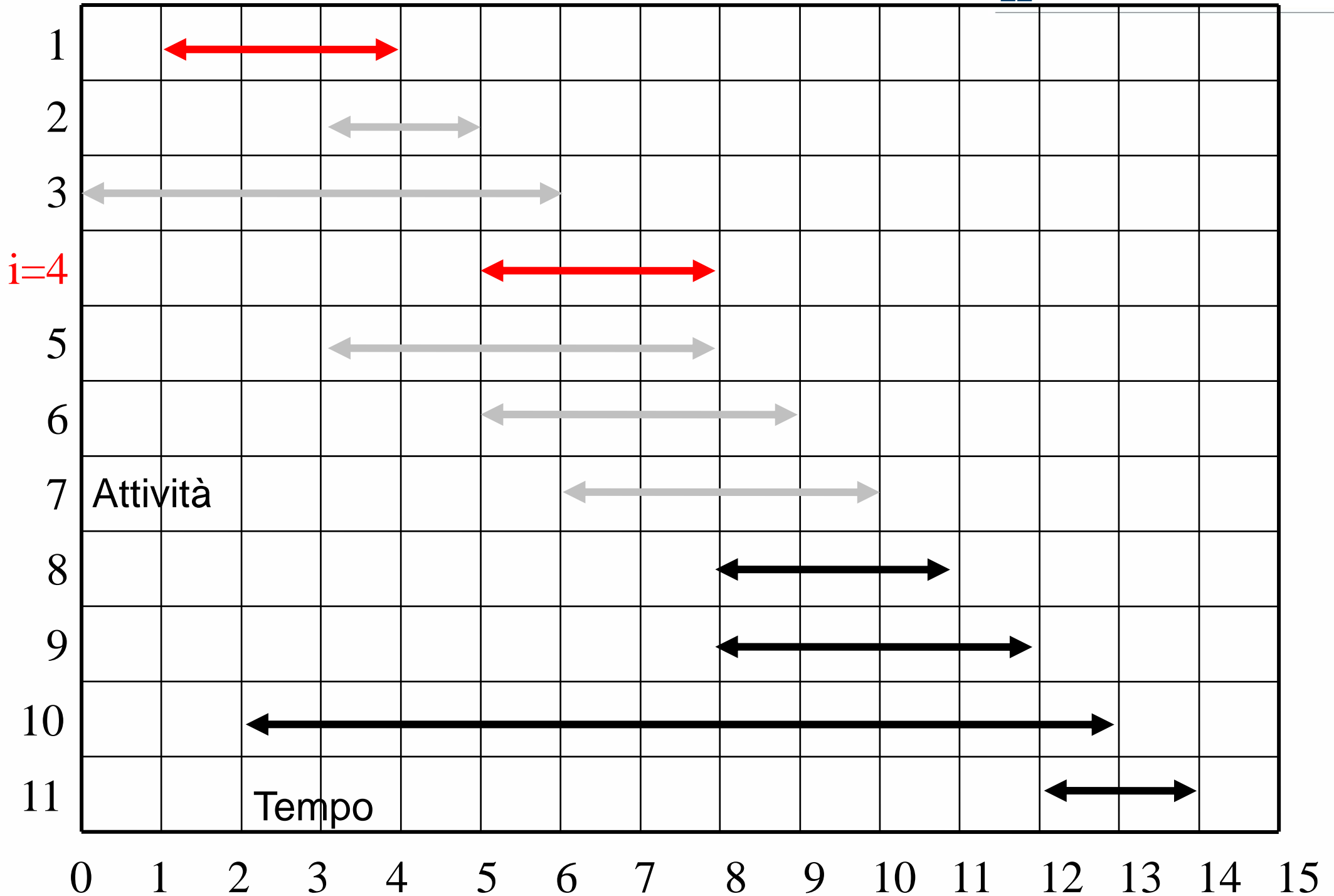
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

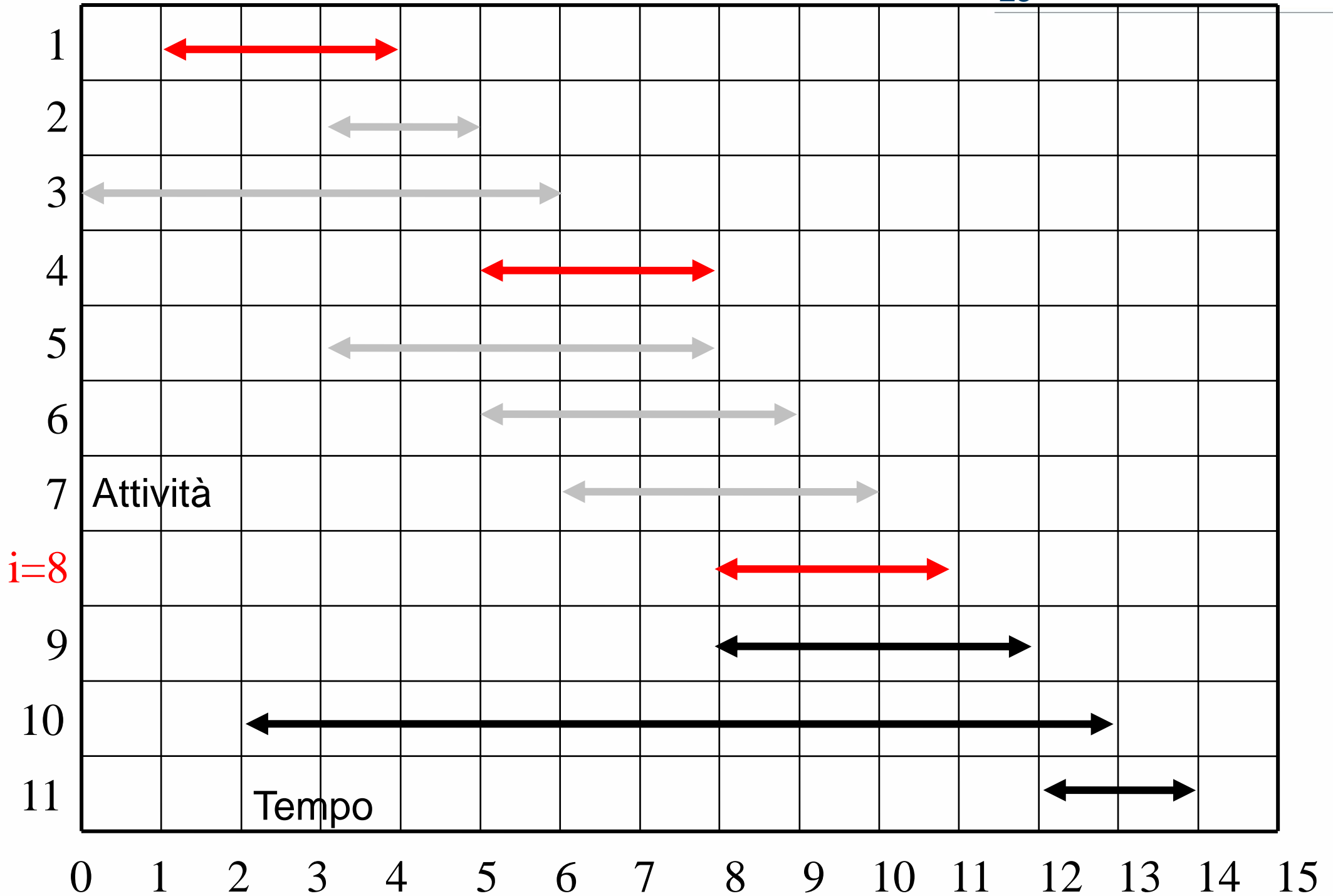


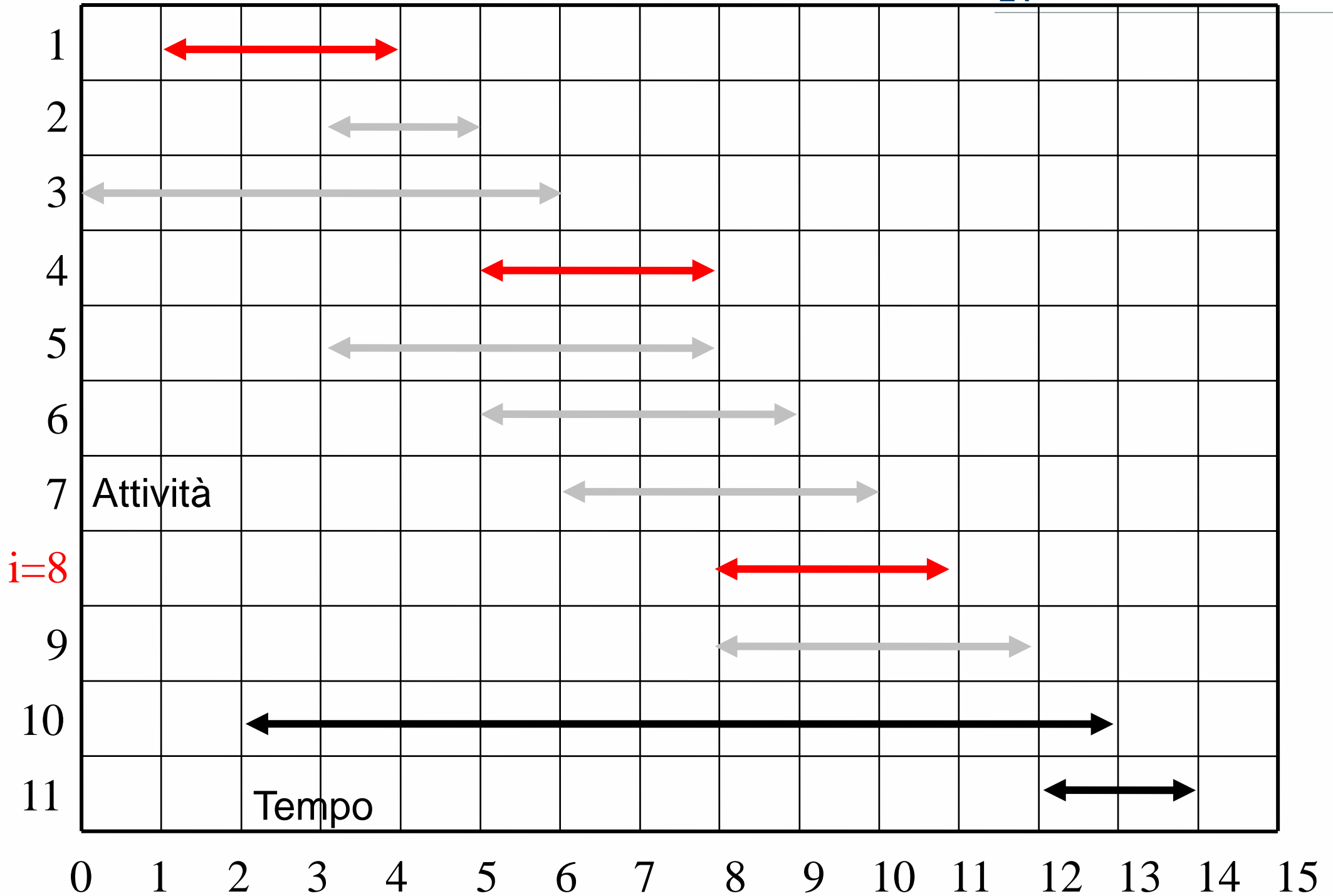


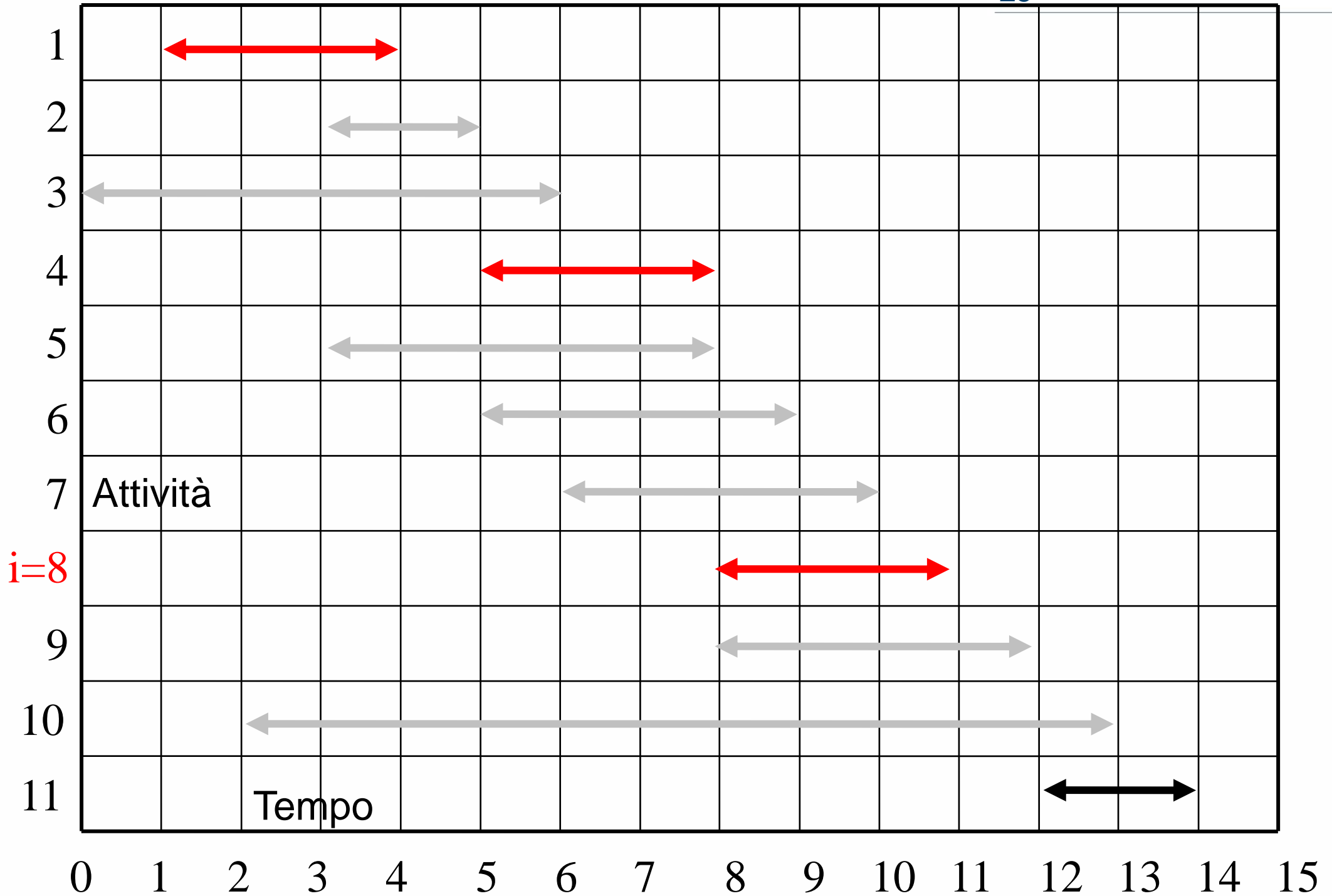


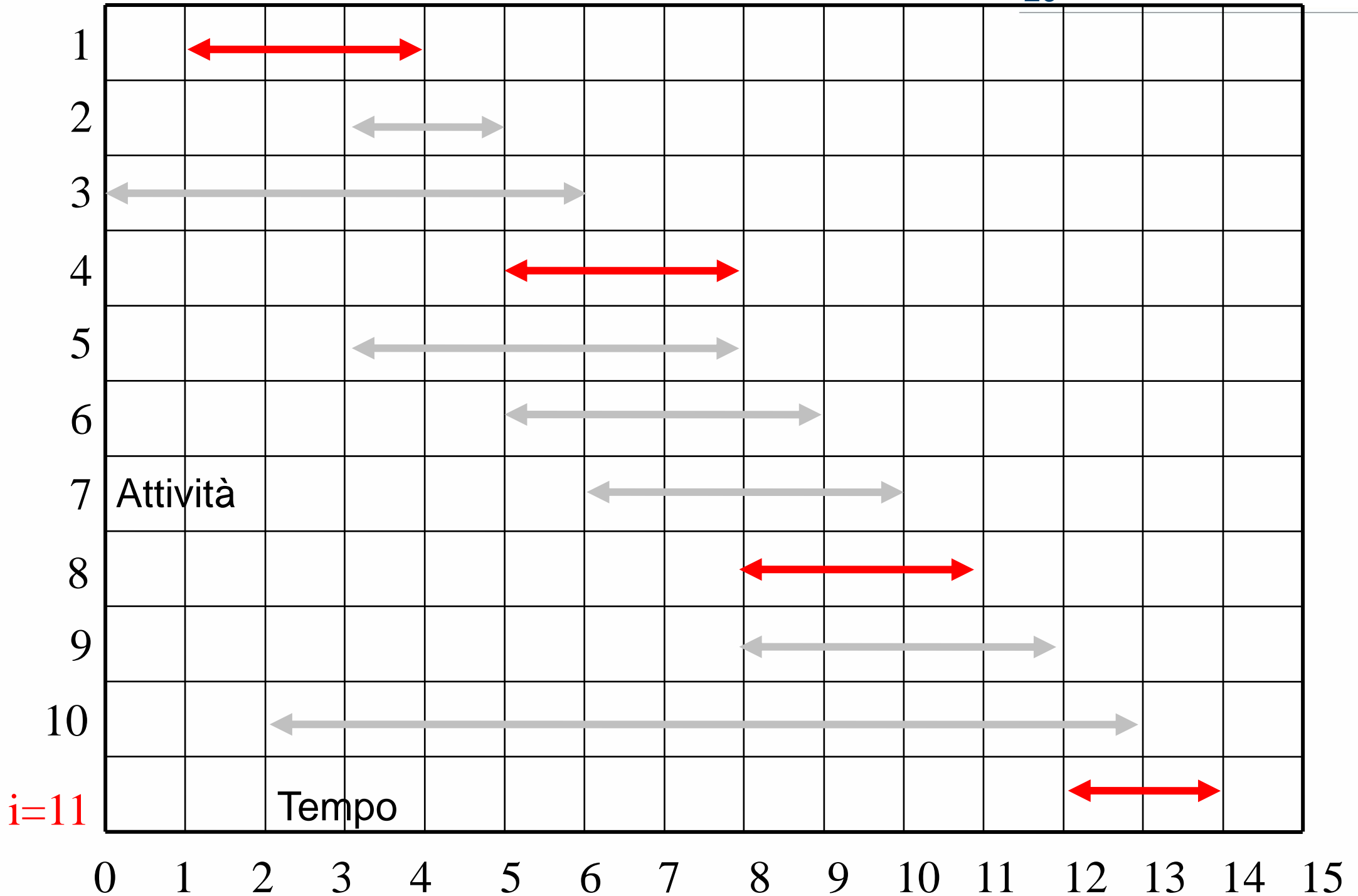












Greedy-Activity-Selection(s[1..n], f[1..n])

```
A := {1}
```

```
j := 1
```

```
for m := 2 to n do
```

```
    if s[m] ≥ f[j] then
```

```
        A := A ∪ {m}
```

```
        j := m
```

```
return A
```

□ Complessità

- ▶ Inserisce la prima attività: è sempre compatibile
- ▶ Se l'input è già ordinato: $\theta(n)$
- ▶ Altrimenti, è necessario un passo di ordinamento: $\theta(n \log n)$

In generale...

- ❑ Evidenziare i “passi di decisione”
 - ▶ Trasformare il problema di ottimizzazione in un problema di “scelte” successive
- ❑ Evidenziare una possibile scelta greedy
 - ▶ Dimostrare che esiste almeno una soluzione ottima che contiene la scelta greedy (“**principio della scelta greedy**”)
- ❑ Evidenziare la sottostruttura ottima
 - ▶ Dimostrare che la soluzione ottima del problema “residuo” dopo la scelta ingorda può essere unito a tale scelta
- ❑ Attenzione: se non vale il **principio della scelta greedy** non si ha più la garanzia che l’algoritmo troverà una soluzione ottima

□ Input

- ▶ Un numero intero positivo n

□ Output

- ▶ Il più piccolo numero intero di pezzi per dare un resto di n centesimi utilizzando monete da 50c, 10c, 5c e 1c.

□ Esempi

- ▶ $n = 78$, 6 pezzi: $50+20+5+1+1+1$
- ▶ $n = 18$, 5 pezzi: $10+5+1+1+1$

□ Domanda 1

- ▶ Descrivere un algoritmo greedy per dare il resto con le monete specificate
 - Sottostruttura ottima
 - Scelta ingorda
- ▶ Provare che tale algoritmo fornisce sempre una soluzione ottima

□ Domanda 2

- ▶ Supponete di avere un insieme di monete i cui valori siano potenze consecutive di c : $c^0, c^1, c^2, \dots, c^k$
- ▶ L'algoritmo individuato in precedenza funziona ancora?

□ Domanda 3

- ▶ Trovare un insieme di "pezzature" per cui la scelta greedy non è applicabile

Codici di Huffman

- ❑ Rappresentare i dati in modo efficiente
 - ▶ Impiegare il numero minore di bit per la rappresentazione
 - ▶ Goal: risparmio spazio su disco e tempo di trasferimento
- ❑ Una possibile tecnica di compressione: *codifica di caratteri*
 - ▶ Tramite *funzione di codifica* $f: f(c) = x$
 - c è un possibile carattere preso da un alfabeto Σ
 - x è una rappresentazione binaria
 - " c è rappresentato da x "

- ❑ Supponiamo di avere un file di n caratteri
 - ▶ Composto dai caratteri: a b c d e f
 - ▶ Con le seguenti frequenze: 45% 13% 12% 16% 9% 5%
- ❑ Codifica tramite ASCII (8 bit per carattere)
 - ▶ Dimensione totale: **8n bit**
- ❑ Codifica basata sull'alfabeto (3 bit per carattere)
 - ▶ Codifica: 000 001 010 011 100 101
 - ▶ Dimensione totale: **3n bit**
- ❑ Possiamo fare di meglio?

❑ Codifica a lunghezza variabile

▶ Caratteri:	a	b	c	d	e	f
▶ Codifica:	0	101	100	111	1101	1100

▶ Costo totale:

$$(0.45*1+0.13*3+0.12*3+0.16*3+0.09*4+0.05*4)*n=2.24n$$

❑ Codice "a prefisso" (meglio, "senza prefissi"):

- ▶ Nessun codice è prefisso di un altro codice
- ▶ Condizione necessaria per permettere la decodifica

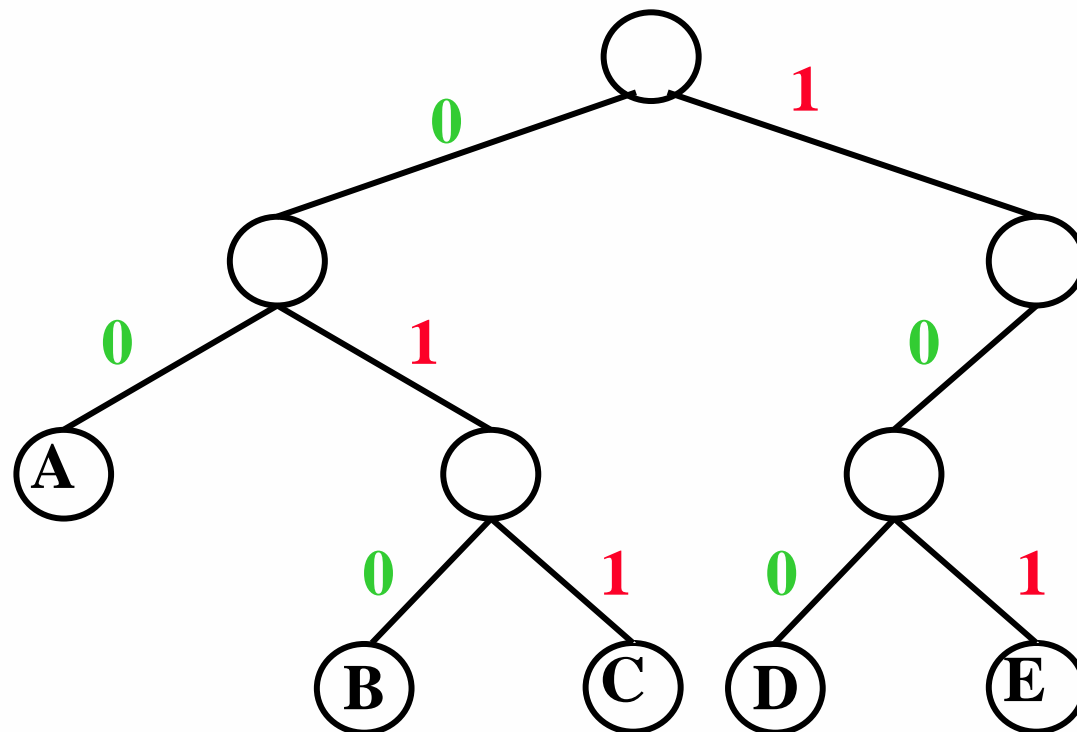
❑ Esempio:

- ▶ 0111111001011000 -> "ADDAABCA"
- ▶ Se $f(a) = 1$ $f(b) = 11$, $11 \rightarrow$ "b" oppure "aa" ??

- Rappresentazione come alberi binari
 - ▶ Figlio sinistro: 0 Figlio destro: 1
 - ▶ Caratteri dell'alfabeto sulle foglie

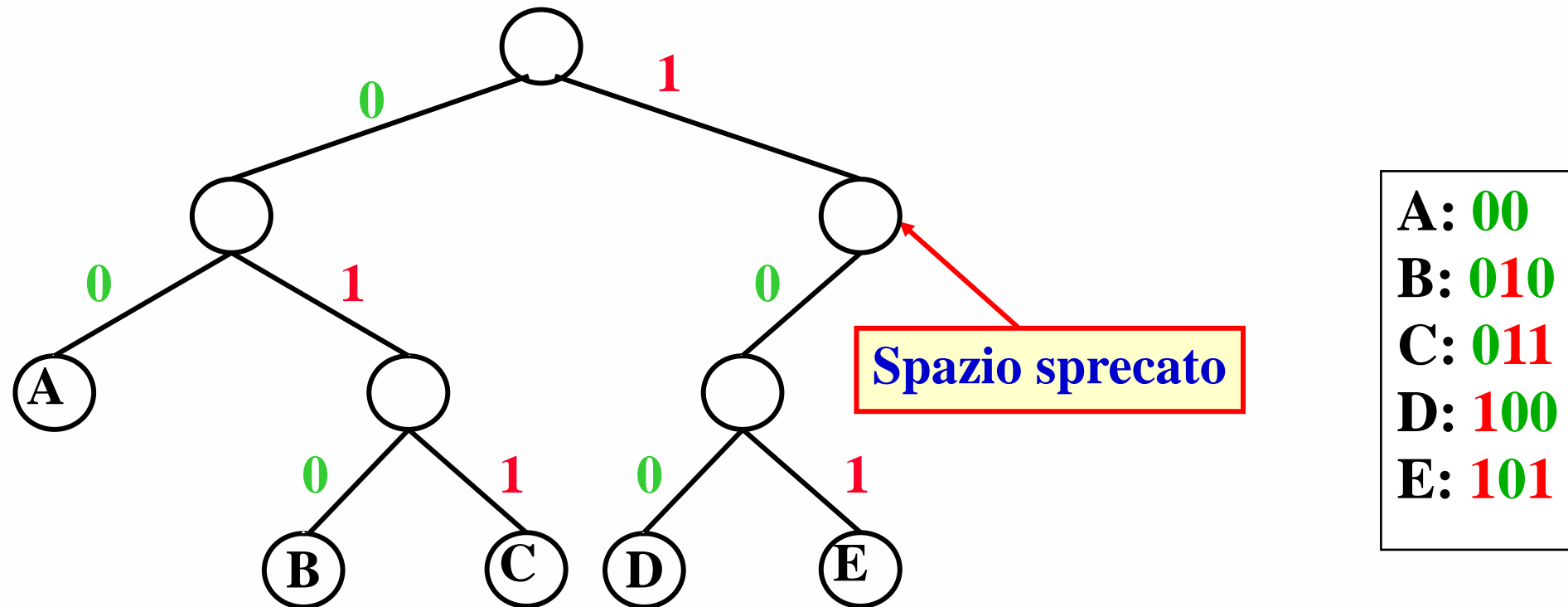
Algoritmo di decodifica:

1. parti dalla radice
2. leggi un bit alla volta percorrendo l'albero:
0: sinistra
1: destra
3. stampa il carattere della foglia
4. torna a 1

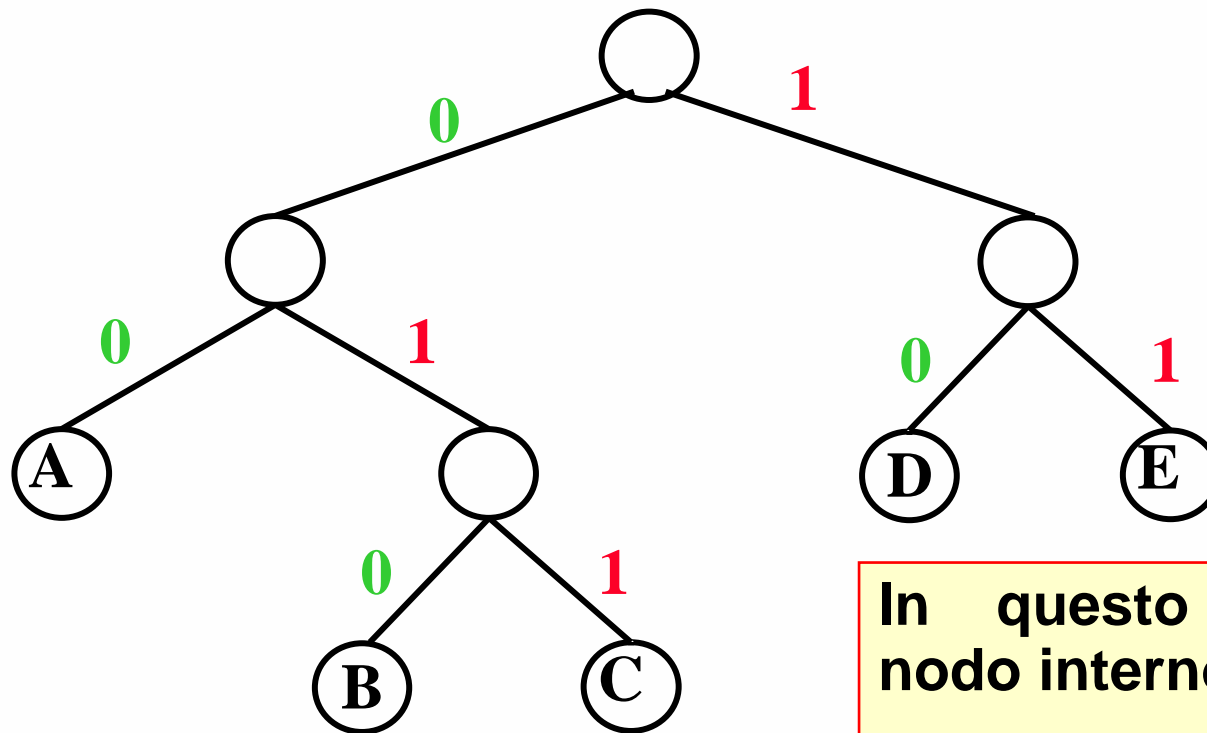


A: 00
 B: 010
 C: 011
 D: 100
 E: 101

- Non c'è motivo di avere un nodo interno con un solo figlio
 - ▶ Il "figlio unico" può essere sostituito al proprio genitore
 - Sia che sia una foglia che un nodo interno



- Non c'è motivo di avere un nodo interno con un solo figlio
 - ▶ Il "figlio unico" può essere sostituito al proprio genitore
 - Sia che sia una foglia che un nodo interno



In questo albero ogni nodo interno ha due figli

A: 00
 B: 010
 C: 011
 D: 100
 E: 101



A: 00
 B: 010
 C: 011
 D: 10
 E: 11

- Definizione: codice ottimo
 - ▶ Dato un file F , un codice C è ottimo per F se non esiste un altro codice tramite il quale F possa essere compresso impiegando un numero inferiore di bit.
- Nota:
 - ▶ Il codice ottimo dipende dal particolare file
 - ▶ Possono esistere più soluzioni ottime
- Teorema:
 - ▶ I codici a prefisso ottimi sono rappresentati da un albero in cui tutti i nodi interni hanno due figli

- Supponiamo di avere:
 - ▶ un file F composto da caratteri nell'**alfabeto** Σ
 - ▶ un **albero** T che rappresenta la codifica
- Quanti bit sono richiesti per codificare il file?
 - ▶ Per ogni $c \in \Sigma$, sia $d_T(c)$ la **profondità** della foglia che rappresenta c
 - ▶ Il codice per c richiederà allora $d_T(c)$ bit
- Se $f[c]$ è il numero di volte che c occorre in F , allora la dimensione della codifica è

$$C(F, T) = \sum_{c \in \Sigma} f[c] d_t(c)$$

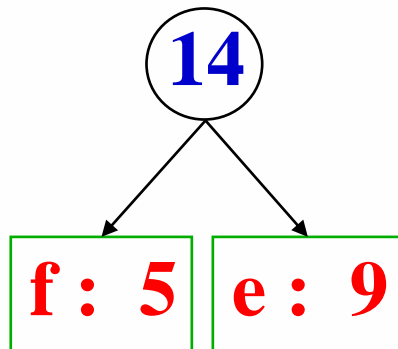
- ❑ Principio del codice di Huffman
 - ▶ Minimizzare la lunghezza dei caratteri che compaiono più frequentemente
 - ▶ Assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero
- ❑ Un codice è progettato per un file specifico
 - ▶ Si ottiene la frequenza di tutti i caratteri
 - ▶ **Si costruisce il codice**
 - ▶ Si rappresenta il file tramite il codice

Passo 1: Costruire una lista di nodi foglia per ogni carattere, etichettato con la propria frequenza

f : 5	e : 9	c : 12	b : 13	d : 16	a : 45
-------	-------	--------	--------	--------	--------

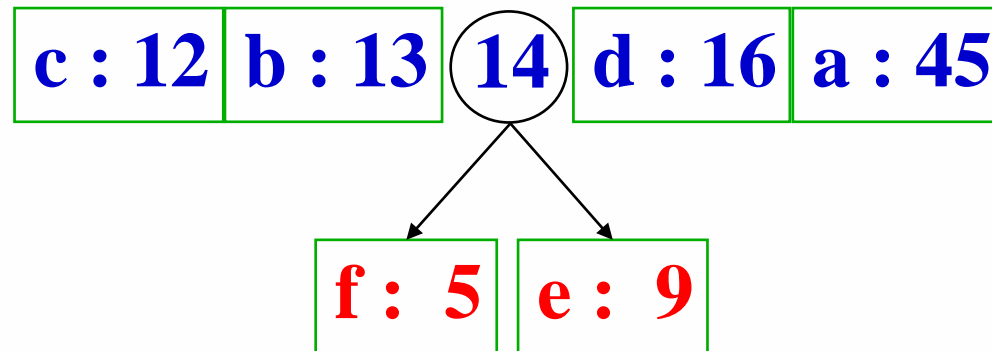
Passo 2: Rimuovere i due nodi “*più piccoli*”
(con frequenze minori)

Passo 3: Collegarli ad un nodo padre etichettato
con la frequenza combinata (sommata)

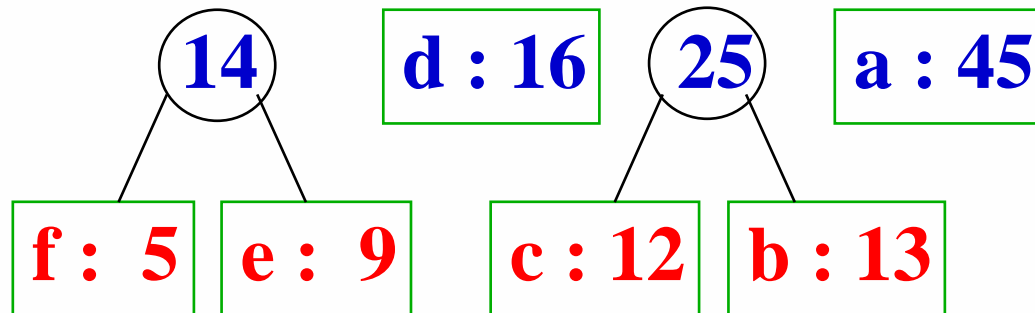


c : 12	b : 13	d : 16	a : 45
--------	--------	--------	--------

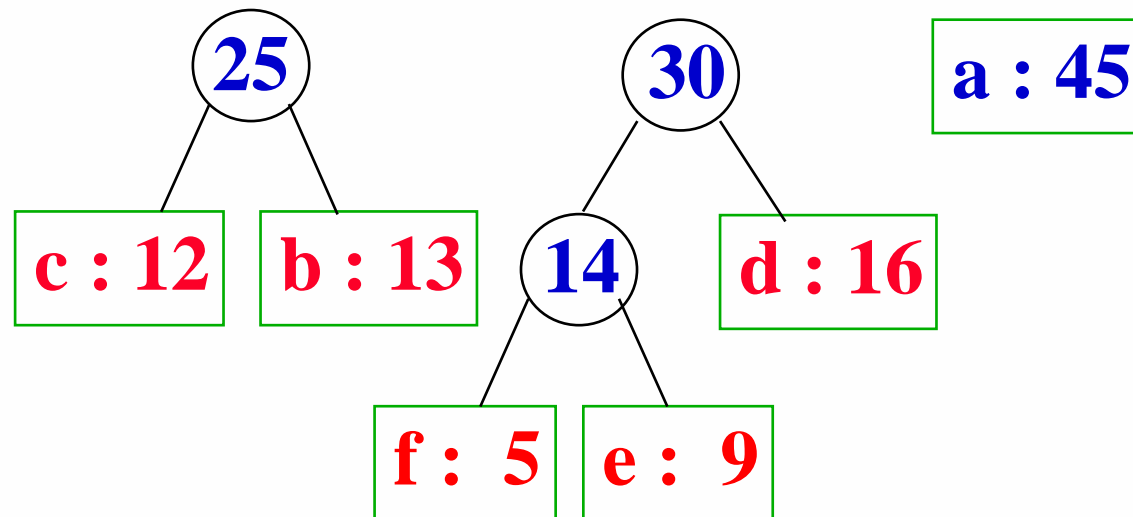
Passo 4: Aggiungere il nodo combinato alla lista.



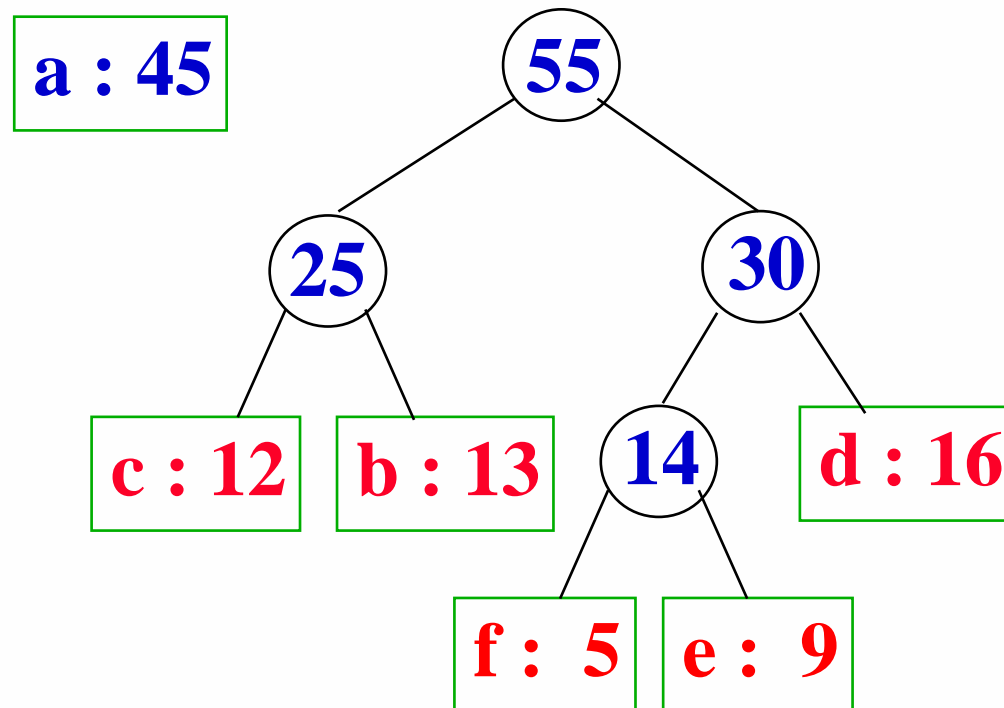
Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista



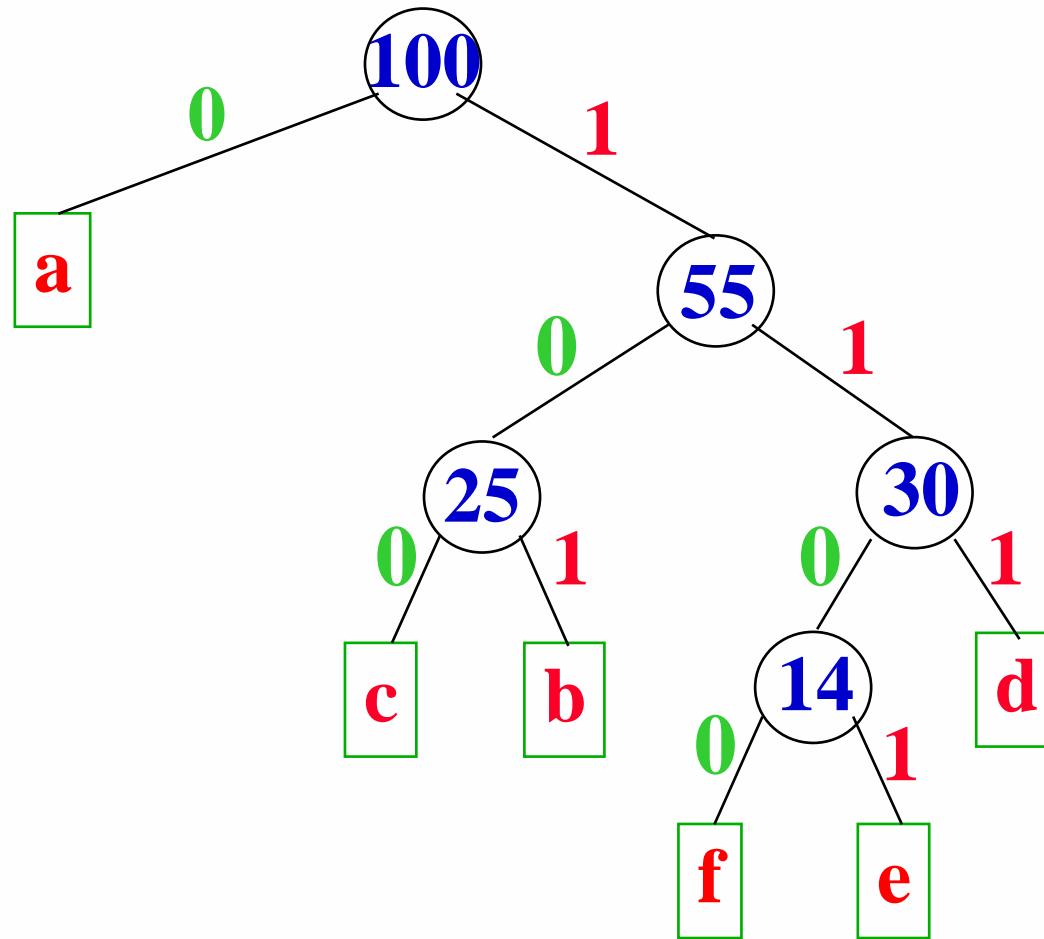
Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista



Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista



Al termine, si etichettano gli archi dell'albero con bit 0-1




```
Huffman(f[1..n], c[1..n])
  Q := new PriorityQueue()
  for i := 1 to n
    z = new Tree(f[i], c[i])
    Q.enqueue(z)
  for i := 1 to n - 1
    z1 = Q.dequeue()
    z2 = Q.dequeue()
    z = new Tree(z1.f + z2.f, '')
    z.left = z1
    z.right = z2
    Q.enqueue(z)
  return Q.dequeue()
```

Tree:

f // frequenza (key)
c // carattere
left // figlio sinistro
right // figlio destro

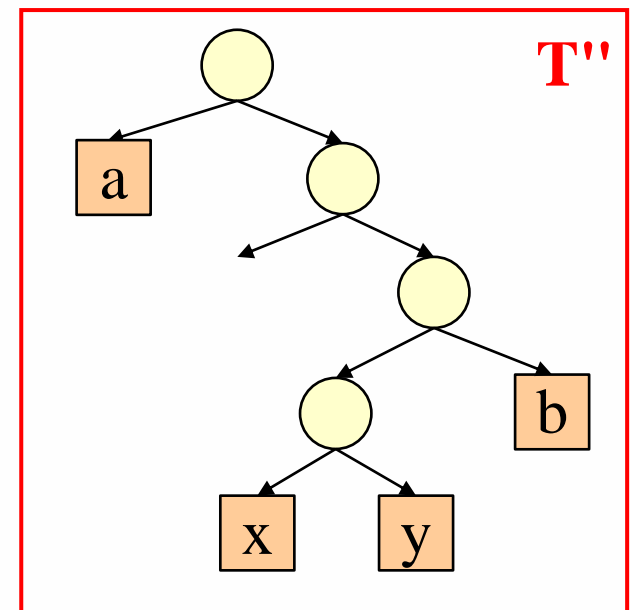
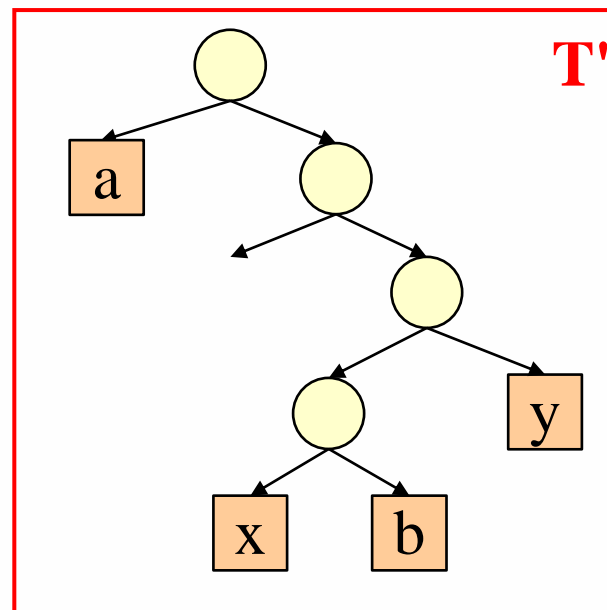
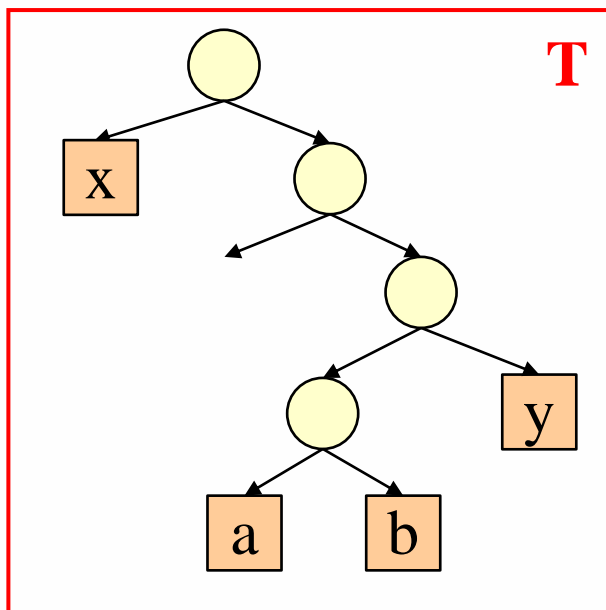
Complessità

$\theta(n \log n)$

- ❑ Teorema: L'output dell'algoritmo Huffman per un dato file è un codice a prefisso ottimo
- ❑ Schema della dimostrazione:
 - ▶ *Sottostruttura ottima*
 - Dato un problema sull'alfabeto Σ , è possibile costruire un sottoproblema con un alfabeto più piccolo
 - ▶ *Proprietà della scelta greedy*
 - Scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale

- Sia
 - ▶ Σ un alfabeto, f un array di frequenze
 - ▶ x, y i due caratteri che hanno frequenza più bassa
- Allora
 - ▶ Esiste un codice prefisso ottimo per Σ in cui x, y hanno *la stessa profondità massima* e i loro codici *differiscono solo per l'ultimo bit*
- Dimostrazione
 - ▶ Al solito, basata sulla trasformazione di una soluzione ottima
 - ▶ Supponiamo che esistano due caratteri a, b con profondità massima e questi siano diversi da x, y

- ❑ Assumiamo (senza perdere in generalità): $f[x] \leq f[y]$ $f[a] \leq f[b]$
- ❑ Poiché le frequenze di x e y sono minime: $f[x] \leq f[a]$ $f[y] \leq f[b]$
- ❑ Scambiamo x con a: otteniamo T'
- ❑ Scambiamo y con b: otteniamo T''



□ Otteniamo che

$$\begin{aligned}C(T) - C(T') &= \sum_{c \in \Sigma} f[c]d_T(c) - \sum_{c \in \Sigma} f[c]d_{T'}(c) \\&= (f[x]d_T(x) + f[a]d_T(a)) - (f[x]d_{T'}(x) + f[a]d_{T'}(a)) \\&= (f[x]d_T(x) + f[a]d_T(a)) - (f[x]d_T(a) + f[a]d_T(x)) \\&= (f[a] - f[x])(d_T(a) - d_T(x)) \\&\geq 0\end{aligned}$$

- In maniera analoga possiamo dimostrare che $C(T') - C(T'') \geq 0$
- Perciò $C(T'') \leq C(T)$ ed essendo T ottimo \Rightarrow anche T'' è ottimo

□ Sia

- ▶ Σ un alfabeto, f un array di frequenze
- ▶ x, y i due caratteri che hanno frequenza più bassa
- ▶ $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$
- ▶ $f[z] = f[x] + f[y]$
- ▶ O un albero che rappresenta *un codice a prefisso ottimo per Σ'*

□ Allora:

- ▶ *L'albero T (ottenuto da O sostituendo il nodo foglia z con un nodo interno con due figli x, y) rappresenta un codice a prefisso ottimo per Σ*

- Esprimiamo la relazione fra il costo di T e O
 - ▶ Per ogni $c \in \Sigma - \{x, y\} \rightarrow f[c] \cdot d_T(c) = f[c] \cdot d_O(c)$
Quindi tutte queste componenti sono uguali
 - ▶ $d_T(x) = d_T(y) = d_O(z) + 1$
 - da cui concludiamo
 - $f[x] \cdot d_T(x) + f[y] \cdot d_T(y) =$
 $(f[x] + f[y])(d_O(z) + 1) =$
 $f[z] \cdot d_O(z) + f[x] + f[y]$
 - da cui concludiamo
 - $C(f, T) = C(f, O) + f[x] + f[y]$
 $C(f, O) = C(f, T) - f[x] - f[y]$

- Per assurdo: supponiamo T non sia ottimo
 - ▶ Allora esiste un albero T' tale che $C(f, T') < C(f, T)$
 - ▶ Senza perdere in generalità, sappiamo che T' ha x e y come foglie sorelle
 - ▶ Sia T'' l'albero ottenuto da T' sostituendo il padre di x, y con un nodo z tale che $f[z] = f[x] + f[y]$
- Allora
 - ▶
$$\begin{aligned} C(f, T'') &= C(f, T') - f[x] - f[y] \\ &< C(f, T) - f[x] - f[y] \\ &= C(f, O) \end{aligned}$$
 - ▶ Il che è assurdo, visto che O è ottimo