

Algoritmi e Strutture Dati

Algoritmi greedy

Alberto Montresor

Università di Trento

2025/03/27

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

- 1 Introduzione
- 2 Insieme indipendente di intervalli
- 3 Resto
- 4 Scheduling
- 5 Zaino frazionario
- 6 Compressione di Huffman
- 7 Alberi di copertura di peso minimo

Introduzione

Problemi di ottimizzazione

- Gli algoritmi per problemi di ottimizzazione eseguono una sequenza di decisioni

Programmazione dinamica

- In maniera bottom-up, valuta tutte le decisioni possibili
- Evitando però di ripetere sotto-problemi (decisioni) già percorse

Algoritmi greedy (ingordi, golosi)

- Seleziona una sola delle possibile decisioni...
- ... quella che sembra ottima (ovvero, è localmente ottima)
- È però necessario dimostrare che si ottien un ottimo globale

Quando applicare la tecnica greedy?

Se è possibile dimostrare che esiste una scelta ingorda

"Fra le molte scelte possibili, ne può essere facilmente individuata una che porta sicuramente alla soluzione ottima."

Se il problema ha sottostruttura ottima

"Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale."

Note

- Non tutti i problemi hanno una scelta ingorda
- In alcuni casi, soluzioni non ottime possono essere comunque interessanti

Insieme indipendente massimale di intervalli

Input

Sia $S = \{1, 2, \dots, n\}$ un insieme di intervalli della retta reale. Ogni intervallo $[a_i, b_i[$, con $i \in S$, è chiuso a sinistra e aperto a destra.

- a_i : tempo di inizio
- b_i : tempo di fine

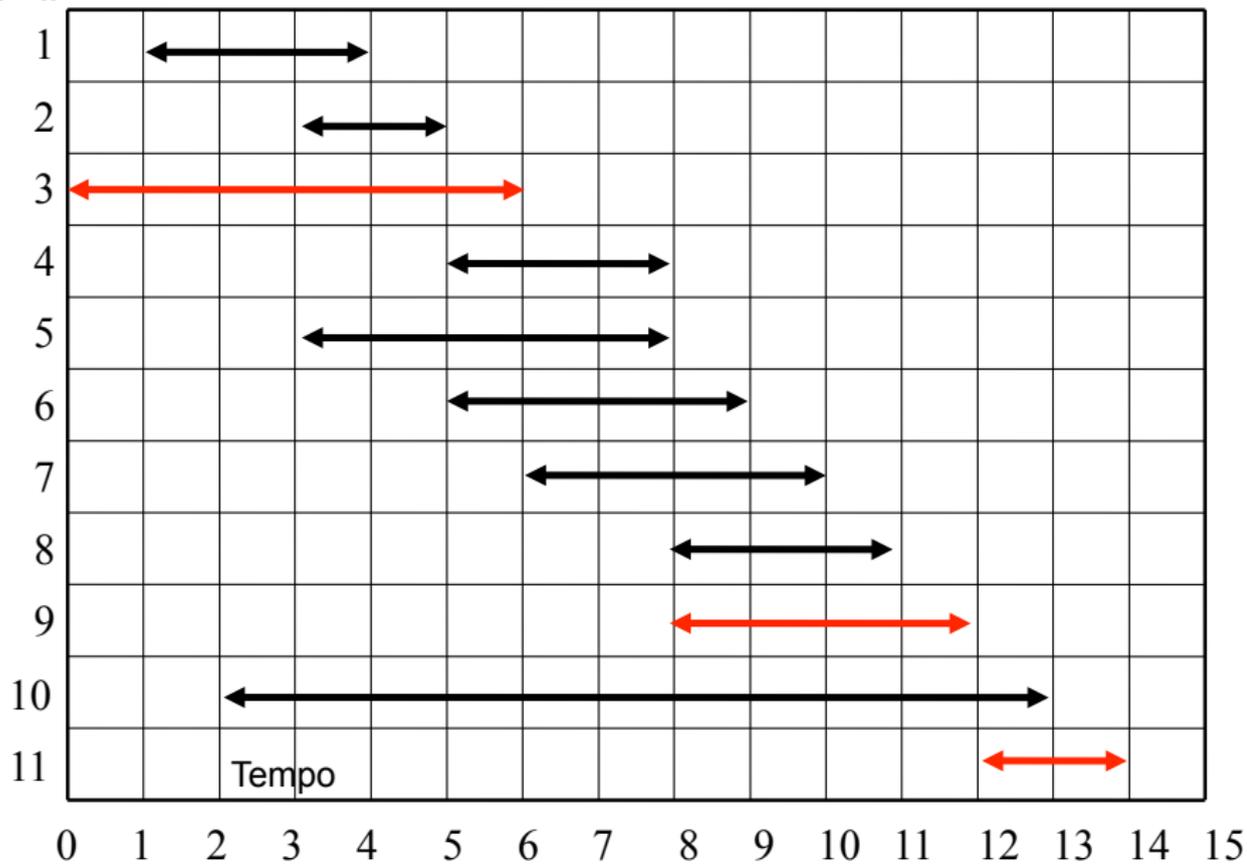
Definizione del problema

Un **insieme indipendente massimale** è un sottoinsieme di massima cardinalità formato da intervalli tutti disgiunti tra loro.

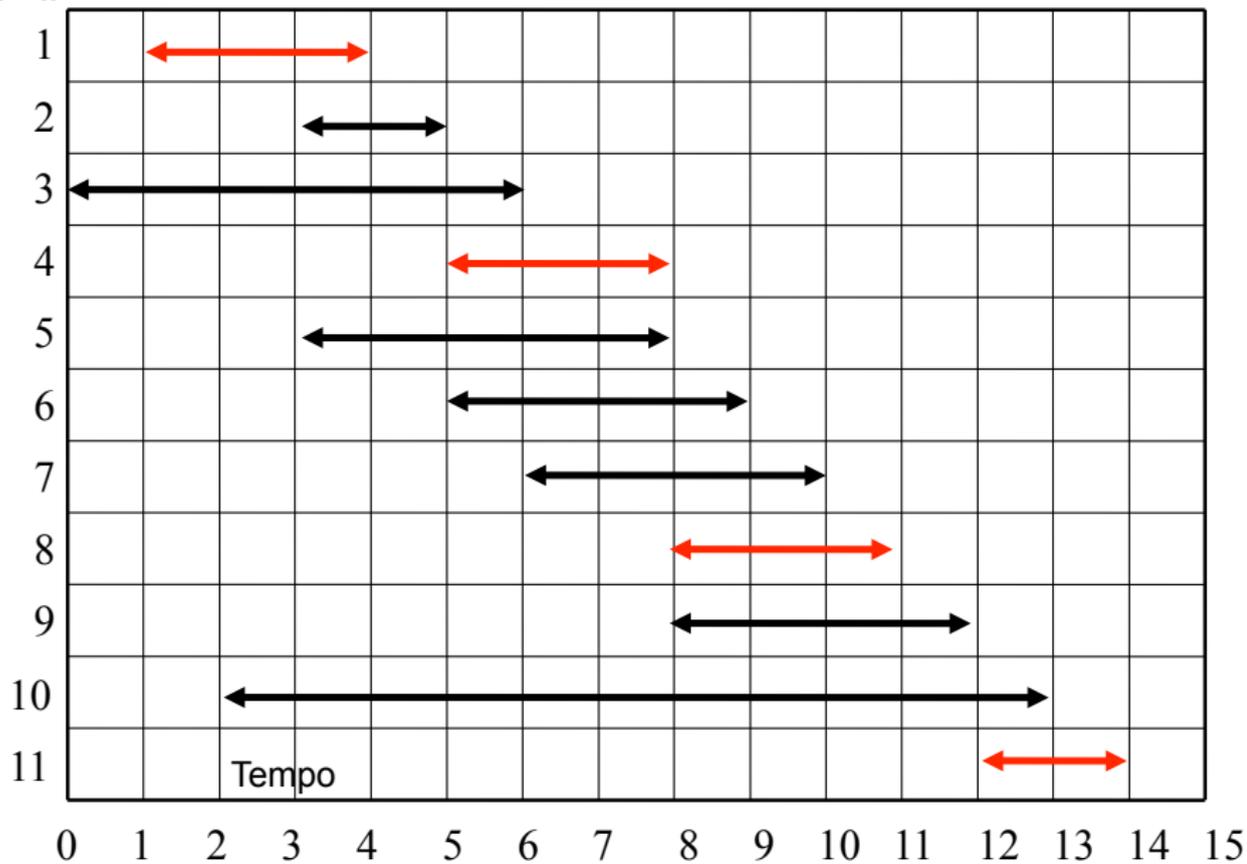
i	a_i	b_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Confronta con **Insieme indipendente di intervalli pesati**

Attività



Attività



Come affrontare il problema

Iniziamo con programmazione dinamica

- Individuiamo una sottostruttura ottima
- Scriviamo una definizione ricorsiva per la dimensione della soluzione ottima
- Scriviamo una versione iterativa bottom-up dell'algoritmo

Passiamo poi alla tecnica greedy

- Cerchiamo una possibile scelta ingorda
- Dimostriamo che la scelta ingorda porta alla soluzione ottima
- Scriviamo un algoritmo ricorsivo o iterativo che effettua sempre la scelta ingorda

Sottostruttura ottima

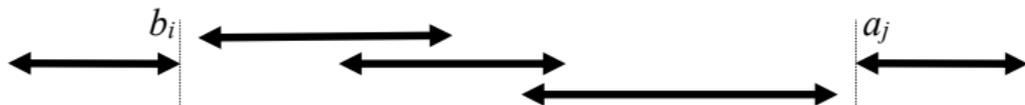
- Si assuma che gli intervalli siano ordinati per tempo di fine:

$$b_1 \leq b_2 \leq \dots \leq b_n$$

- Definiamo il **sottoproblema** $S[i \dots j]$ come l'insieme di intervalli che iniziano dopo la fine di i e finiscono prima dell'inizio di j :

$$S[i \dots j] = \{k \mid b_i \leq a_k < b_k \leq a_j\}$$

- Aggiungiamo due intervalli fittizi:
 - Intervallo 0: $b_0 = -\infty$
 - Intervallo $n + 1$: $a_{n+1} = +\infty$
- Il problema iniziale corrisponde al problema $S[0, n + 1]$



Sottostruttura ottima

Teorema

Supponiamo che $A[i \dots j]$ sia una soluzione ottimale di $S[i \dots j]$ e sia k un intervallo che appartiene a $A[i \dots j]$; allora

- Il problema $S[i \dots j]$ viene suddiviso in due sottoproblemi
 - $S[i \dots k]$: gli intervalli di $S[i \dots j]$ che finiscono prima di k
 - $S[k \dots j]$: gli intervalli di $S[i \dots j]$ che iniziano dopo di k
- $A[i \dots j]$ contiene le soluzioni ottimali di $S[i \dots k]$ e $S[k \dots j]$
 - $A[i \dots j] \cap S[i \dots k]$ è la soluzione ottimale di $S[i \dots k]$
 - $A[i \dots j] \cap S[k \dots j]$ è la soluzione ottimale di $S[k \dots j]$

Dimostrazione

Utilizzando il metodo cut-and-paste

Definizione ricorsiva del costo della soluzione

Definizione ricorsiva della soluzione

$$A[i \dots j] = A[i \dots k] \cup \{k\} \cup A[k \dots j]$$

Definizione ricorsiva del suo costo

- Come determinare k ? Analizzando tutte le possibilità
- Sia $DP[i][j]$ la dimensione del più grande sottoinsieme $A[i \dots j] \subseteq S[i \dots j]$ di intervalli indipendenti

$$DP[i][j] = \begin{cases} 0 & S[i \dots j] = \emptyset \\ \max_{k \in S[i \dots j]} \{DP[i][k] + DP[k][j] + 1\} & \text{altrimenti} \end{cases}$$

Verso una soluzione ingorda

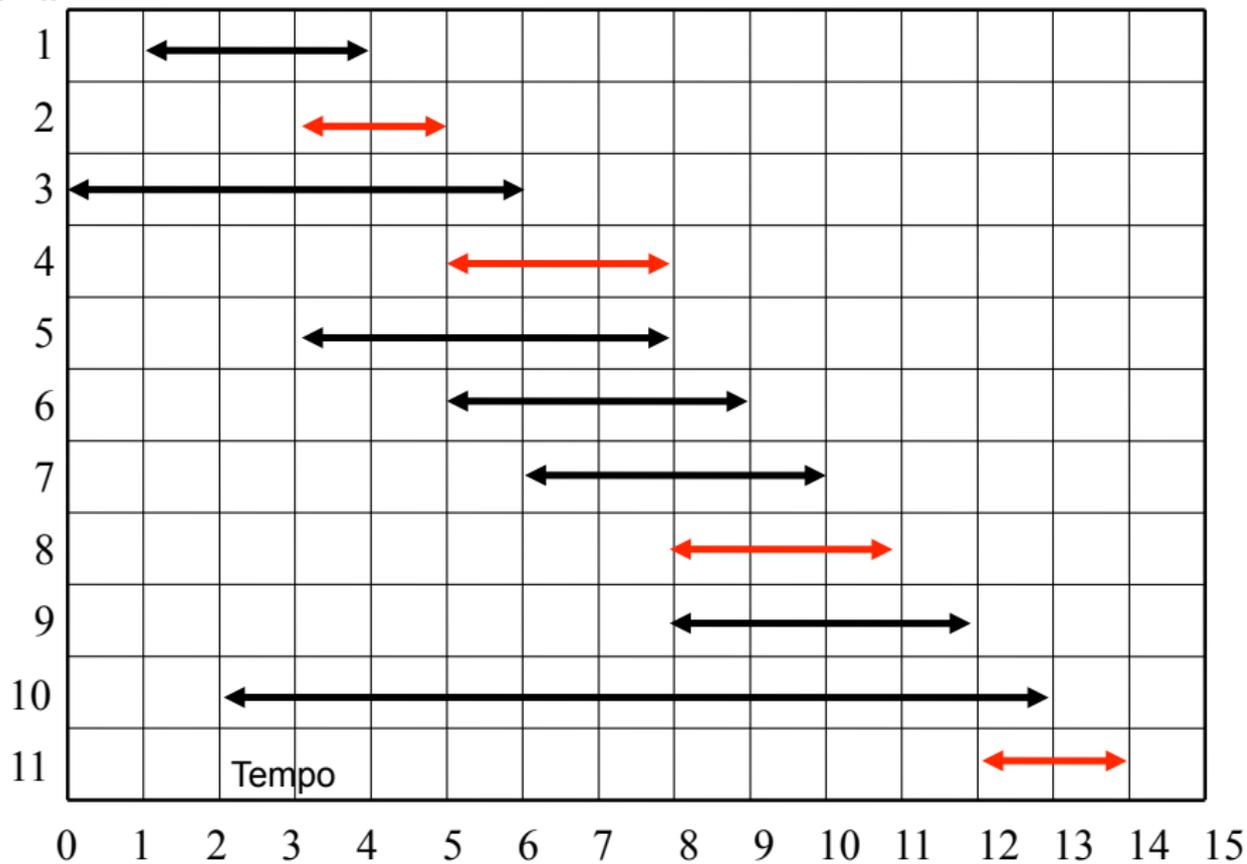
Programmazione dinamica

- La definizione precedente ci permette di scrivere un algoritmo basato su programmazione dinamica o su memoization
- Complessità $O(n^3)$: bisogna risolvere tutti i problemi con $i < j$, con costo $O(n)$ per sottoproblema nel caso peggiore

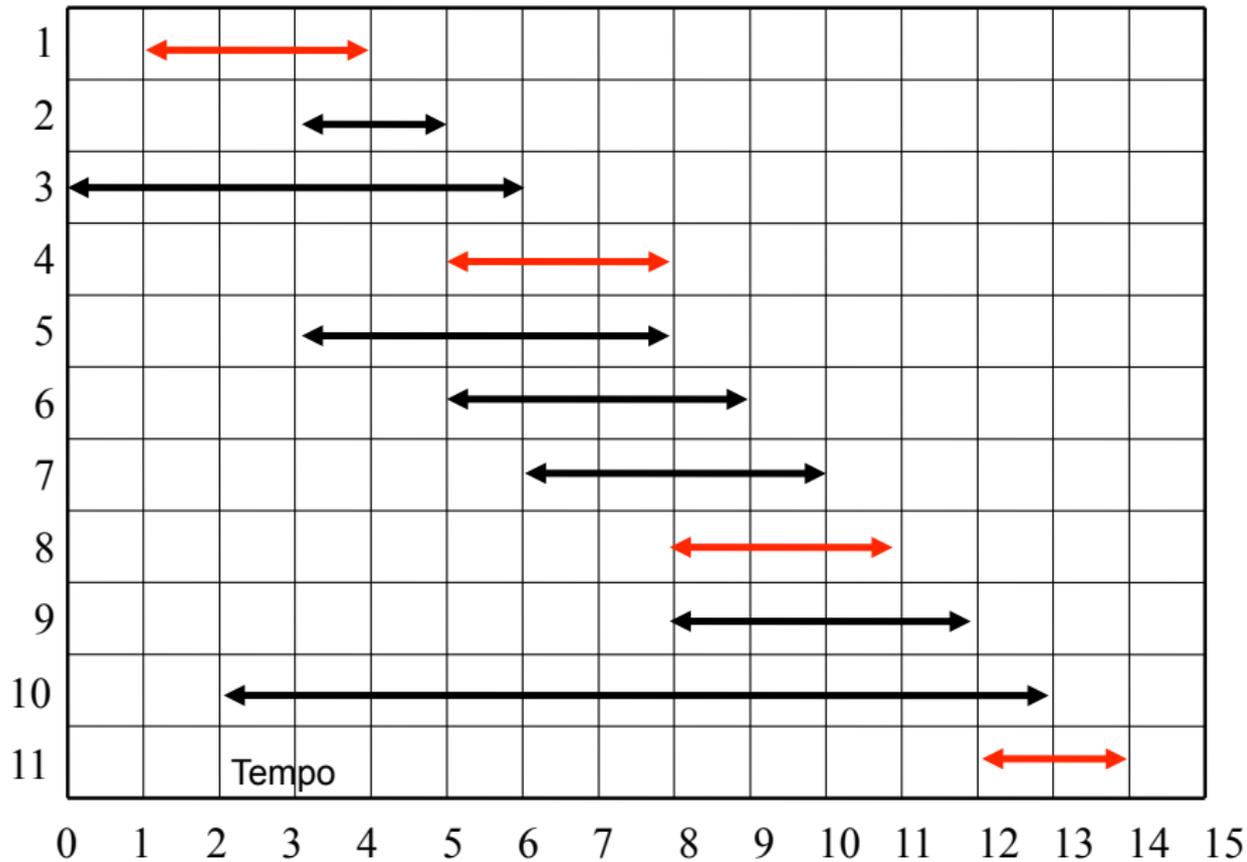
Possiamo fare di meglio?

- Abbiamo visto una soluzione $O(n \log n)$ nel caso di intervalli pesati
- È possibile utilizzare quella soluzione con pesi pari a 1
- Questa soluzione è peggiore, ma...
- Siamo sicuri che sia necessario analizzare tutti i possibili valori k ?

Attività



Attività



Scelta ingorda (Greedy Choice)

Teorema

Sia $S[i \dots j]$ un sottoproblema non vuoto, e m l'intervallo di $S[i \dots j]$ con il **minor tempo di fine**, allora:

- ① il sottoproblema $S[i \dots m]$ è vuoto
- ② m è compreso in qualche soluzione ottima di $S[i \dots j]$

Dimostrazione ①

Sappiamo che: $a_m < b_m$ (Definizione di intervallo)

Sappiamo che: $\forall k \in S[i \dots j] : b_m \leq b_k$ (m ha minor tempo di fine)

Ne consegue: $\forall k \in S[i \dots j] : a_m < b_k$ (Transitività)

Se nessun intervallo in $S[i \dots j]$ termina prima di a_m , allora $S[i \dots m] = \emptyset$

Scelta ingorda (Greedy Choice)

Teorema

Sia $S[i \dots j]$ un sottoproblema non vuoto, e m l'intervallo di $S[i \dots j]$ con il **minor tempo di fine**, allora:

- ① il sottoproblema $S[i \dots m]$ è vuoto
- ② m è compreso in qualche soluzione ottima di $S[i \dots j]$

Dimostrazione ②

- Sia $A'[i \dots j]$ una soluzione ottima di $S[i \dots j]$
- Sia m' l'intervallo con minor tempo di fine in $A'[i \dots j]$
- Sia $A[i \dots j] = (A'[i \dots j] - \{m'\}) \cup \{m\}$ una nuova soluzione ottenuta togliendo m' e aggiungendo m ad $A'[i \dots j]$
- $A[i \dots j]$ è una **soluzione ottima che contiene m** , in quanto ha la stessa dimensione di $A'[i \dots j]$ e gli intervalli sono indipendenti.

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: $S[m \dots j]$

Algoritmo

```
SET independentSet(int[] a, int[] b)
```

```
{ ordina a e b in modo che  $b[1] \leq b[2] \leq \dots \leq b[n]$  }
```

```
SET S = Set()
```

```
S.insert(1)
```

```
int last = 1
```

```
% Ultimo intervallo inserito
```

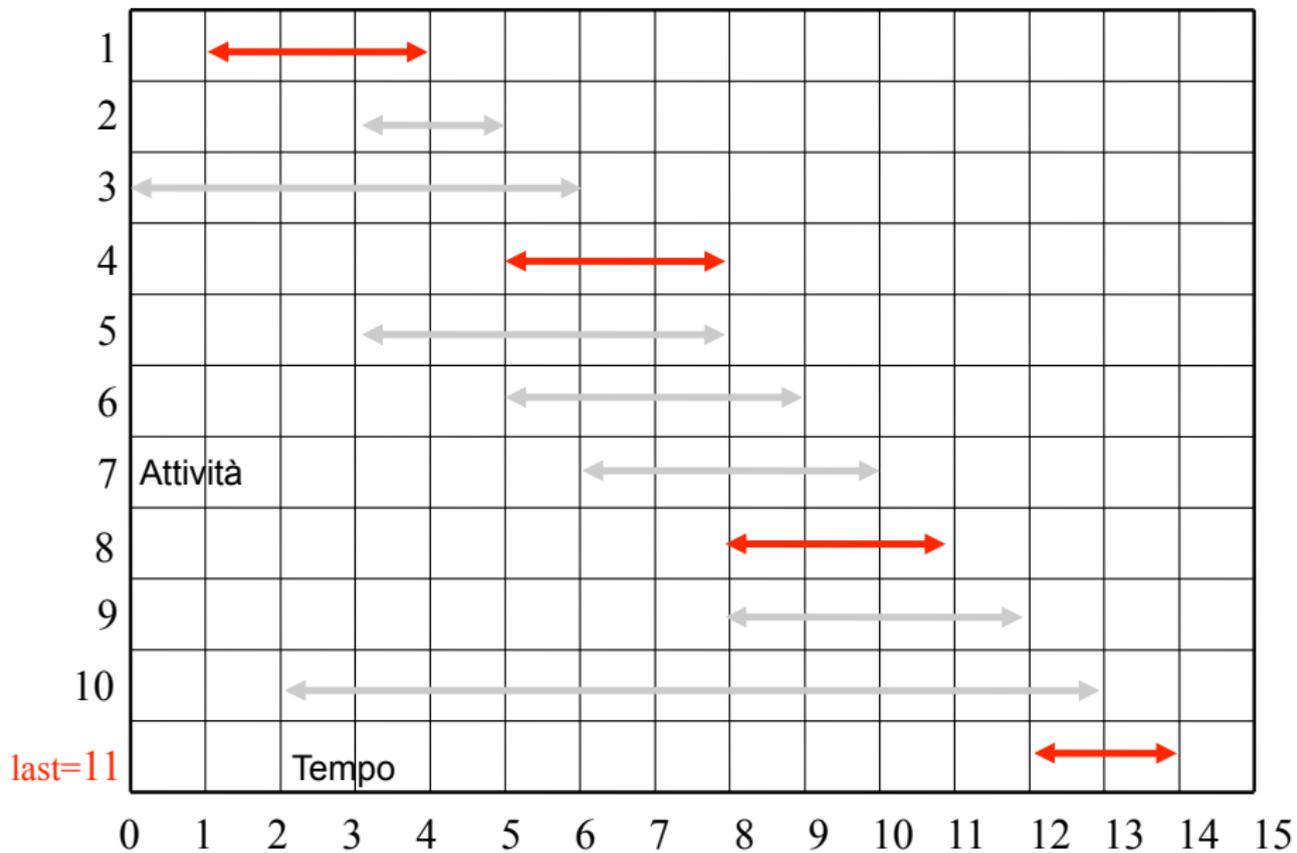
```
for i = 2 to n do
```

```
    if  $a[i] \geq b[last]$  then
        S.insert(i)
        last = i
```

```
% Controllo indipendenza
```

```
return S
```

Complessità: $O(n \log n)$ se input non è ordinato
 $O(n)$ se l'input è già ordinato.



Approccio a partire da Programmazione Dinamica

- Abbiamo cercato di risolvere il problema della selezione delle attività tramite programmazione dinamica:
 - Abbiamo individuato una sottostruttura ottima
 - Abbiamo scritto una definizione ricorsiva per la dimensione della soluzione ottima
- Abbiamo dimostrato la proprietà della scelta greedy:
 - Per ogni sottoproblema, esiste almeno una soluzione ottima che contiene la scelta greedy
 - Abbiamo scritto un algoritmo iterativo che effettua sempre la scelta ingorda

Problema del resto (Money change)

Input

- Un insieme di "tagli" di monete, memorizzati in un vettore di interi positivi $t[1 \dots n]$.
- Un intero R rappresentante il resto che dobbiamo restituire.

Definizione del problema

Trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli disponibili, assumendo di avere un numero illimitato di monete per ogni taglio.

Formalmente, trovare un vettore x di interi non negativi tale che:

$$R = \sum_{i=1}^n x[i] \cdot t[i] \quad \text{e} \quad m = \sum_{i=1}^n x[i] \text{ ha valore minimo}$$

Soluzione basata su programmazione dinamica

Sottostruttura ottima

- Sia $S(i)$ il problema di dare un resto pari ad i
- Sia $A(i)$ una soluzione ottima del problema $S(i)$, rappresentata da un multi-insieme; sia $j \in A(i)$
- Allora, $S(i - t[j])$ è un sottoproblema di $S(i)$, la cui soluzione ottima è data da $A(i) - \{j\}$.

Definizione ricorsiva

- Tabella di programmazione dinamica: $DP[0 \dots R]$
- $DP[i]$: **minimo n. di monete per risolvere il problema $S(i)$**

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min_{1 \leq j \leq n \wedge t[j] \leq i} \{DP[i - t[j]]\} + 1 & i > 0 \end{cases}$$

Algoritmo

```

int[] moneyChange(int[] t, int n, int R)


---


int[] DP = new int[0...R]           % Value of the solution
int[] coin = new int[0...R]       % Coin to be used for a specific value
DP[0] = 0
for i = 1 to R do
    | DP[i] = +∞
    | for j = 1 to n do
    | | if i ≥ t[j] and DP[i - t[j]] + 1 < DP[i] then
    | | | DP[i] = DP[i - t[j]] + 1
    | | | coin[i] = j
    |
[...]
```

Algoritmo

```
int[] moneyChange(int[] t, int n, int R)
```

```
[...]
```

```
% Solution reconstruction
```

```
int[] x = new int[1 ...n] = {0} % Output vector, initialized to zero
```

```
while R > 0 do
```

```
    x[coin[R]] = x[coin[R]] + 1
    R = R - t[coin[R]]
```

```
return x
```

Comlessità?

$O(nR)$

Scelta greedy

Domanda

È possibile pensare ad una soluzione greedy?

Risposta

Selezionare la moneta j più grande tale per cui $t[j] \leq R$, e poi risolvere il problema $S(R - t[j])$.

Esempi

- Tagli: 200, 100, 50, 20, 10, 5, 2, 1
- Tagli: 50, 10, 5, 1
- Tagli: 10, 8, 1
- Tagli: $c^k, c^{k-1}, \dots, c, 1$ ($c \in \mathbb{Z}^+$)

Algoritmo

```
int[] moneyChange(int[] t, int n, int R)
```

```
int[] x = new int[1 ...n]
```

```
{ Ordina le monete in modo decrescente }
```

```
for i = 1 to n do
```

```
     $x[i] = \lfloor R/t[i] \rfloor$   
     $R = R - x[i] \cdot t[i]$ 
```

```
return x
```

Complessità: $O(n \log n)$ se input non è ordinato
 $O(n)$ se l'input è già ordinato.

Dimostrazione scelta greedy $t = [50, 10, 5, 1]$

- Sia x una qualunque soluzione ottima; quindi

$$\sum_{i=1}^4 x[i] \cdot t[i] = R \quad m = \sum_{i=1}^4 x[i] \quad \text{è minimo}$$

- Sappiamo che $t[k] \cdot x[k] < t[k-1]$, altrimenti basterebbe sostituire un certo numero di monete di taglia $t[k]$ con quelle del taglio $t[k-1]$.

$$t[2] \cdot x[2] = 10 \cdot x[2] < t[1] = 50 \Rightarrow x[2] < 5$$

$$t[3] \cdot x[3] = 5 \cdot x[3] < t[2] = 10 \Rightarrow x[3] < 2$$

$$t[4] \cdot x[4] = 1 \cdot x[4] < t[3] = 5 \Rightarrow x[4] < 5$$

Dimostrazione scelta greedy $t = [50, 10, 5, 1]$

- Sia m_k la somma delle monete di taglio inferiore a $t[k]$:

$$m_k = \sum_{i=k+1}^4 x[i] \cdot t[i]$$

- Se dimostriamo che $\forall k : m_k < t[k]$, allora la soluzione (ottima) è proprio quella calcolata dall'algoritmo

$$\begin{array}{rclclclclcl}
 m_4 & = & 0 & & & < & 1 & & = & t[4] \\
 m_3 & = & x[4] \cdot 1 + m_4 & \leq & 4 \cdot 1 + m_4 & < & 4 + 1 & = & 5 & = & t[3] \\
 m_2 & = & x[3] \cdot 5 + m_3 & \leq & 1 \cdot 5 + m_3 & < & 5 + 5 & = & 10 & = & t[2] \\
 m_1 & = & x[2] \cdot 10 + m_2 & \leq & 4 \cdot 10 + m_2 & < & 40 + 10 & = & 50 & = & t[1]
 \end{array}$$

Approccio greedy, senza programmazione dinamica

- Evidenziare i "passi di decisione"
 - Trasformare il problema di ottimizzazione in un problema di "scelte" successive
- Evidenziare una possibile scelta ingorda
 - Dimostrare che tale scelta rispetto il "principio della scelta ingorda"
- Evidenziare la sottostruttura ottima
 - Dimostrare che la soluzione ottima del problema "residuo" dopo la scelta ingorda può essere unito a tale scelta
- Scrittura codice: top-down, anche in maniera iterativa
 - Nota: può essere necessario pre-processare l'input

Scheduling

Input

Supponiamo di avere un processore e n job da eseguire su di esso, ognuno caratterizzato da un tempo di esecuzione $t[i]$ noto a priori.

Problema

Trovare una sequenza di esecuzione (permutazione) che minimizzi il **tempo di completamento medio**.

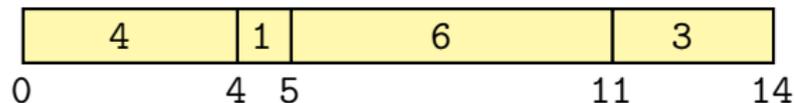
Tempo di completamento

Dato un vettore $A[1 \dots n]$ contenente una permutazione di $\{1, \dots, n\}$, il **tempo di completamento** dell' h -esimo job nella permutazione è:

$$T_A(h) = \sum_{i=1}^h t[A[i]]$$

Esempio

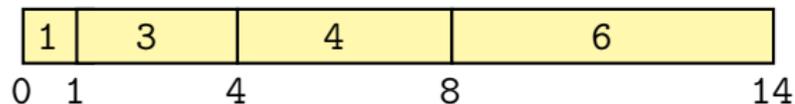
Esempio



Tempo di completamento medio:

$$(4 + 5 + 11 + 14)/4 = 34/4 = 8.5$$

Shortest job first



Tempo di completamento medio:

$$(1 + 4 + 8 + 14)/4 = 27/4 = 6.75$$

Dimostrazione di correttezza

Teorema - Scelta greedy

Esiste una soluzione ottima A in cui il job con minor tempo di fine m si trova in prima posizione ($A[1] = m$).

Teorema – Sottostruttura ottima

Sia A una soluzione ottima di un problema con n job, in cui il job con minor tempo di fine m si trova in prima posizione. La permutazione dei seguenti $n - 1$ job in A è una soluzione ottima al sottoproblema in cui il job m non viene considerato.

Dimostrazione – Scelta greedy

Trasformazione soluzione ottima

- Si consideri una permutazione ottima A :

$$A = \begin{array}{cccccccc} 1 & 2 & & m-1 & m & m+1 & & n-1 & n \\ \hline A[1] & A[2] & \dots & A[m-1] & A[m] & A[m+1] & \dots & A[n-1] & A[n] \end{array}$$

- Sia m la posizione in A in cui si trova il job con **minor tempo di fine**
- Si consideri una permutazione A' in cui i job in posizione 1, m vengono scambiati:

$$A' = \begin{array}{cccccccc} 1 & 2 & & m-1 & m & m+1 & & n-1 & n \\ \hline A[m] & A[2] & \dots & A[m-1] & A[1] & A[m+1] & \dots & A[n-1] & A[n] \end{array}$$

- Il tempo di completamento medio di A' è minore o uguale al tempo di completamento medio di A

Dimostrazione – Scelta greedy

La soluzione trasformata è anch'essa ottima

- Il tempo di completamento medio di A' è minore o uguale al tempo di completamento medio di A



- Job in posizione $1, \dots, m-1$ in A' hanno tempo di completamento \leq dei job in posizione $1, \dots, m-1$ in A
- Job in posizione m, \dots, n in A' hanno tempo di completamento $=$ dei job in posizione m, \dots, n in A
- Poichè A è ottima, A' non può avere tempo di completamento medio minore e quindi anche A' è ottima.

Problema dello zaino

Input

- Un intero positivo C - la capacità dello zaino
- n oggetti, tali che l'oggetto i -esimo è caratterizzato da
 - un profitto $p_i \in \mathbb{Z}^+$
 - un peso $w_i \in \mathbb{Z}^+$

Zaino 0/1

Trovare un sottoinsieme S di $\{1, \dots, n\}$ di oggetti tale che il loro peso totale non superi la capacità massima e il loro profitto totale sia massimo.

Zaino reale (o Zaino frazionario)

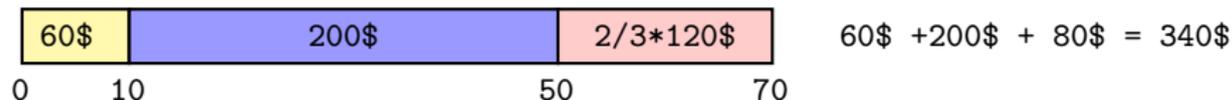
È possibile prendere frazioni di oggetti.

Esempio

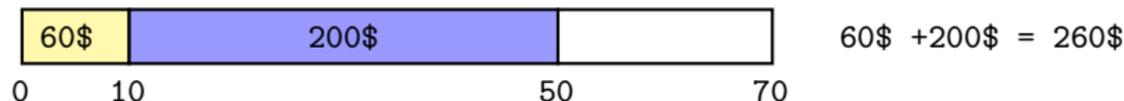
Consideriamo i tre oggetti a lato ed una capacità di 70

i	p_i	w_i	p_i/w_i
1	60\$	10	6\$
2	200\$	40	5\$
3	120\$	30	4\$

Approccio 3: Ordinati per **profitto specifico** p_i/w_i decrescente



Approccio 3 non funziona per Zaino 0/1



Algoritmo

```
float[] zaino(float[] p, float[] w, float C, int n)
```

```
float[] x = new float[1 ...n]
```

```
{ ordina p e w in modo che  $p[1]/w[1] \geq p[2]/w[2] \geq \dots \geq p[n]/w[n]$  }
```

```
for i = 1 to n do
```

```
    [ x[i] = min(C/w[i], 1)
      C = C - x[i] · w[i]
```

```
return x
```

Complessità: $O(n \log n)$ se input non è ordinato
 $O(n)$ se l'input è già ordinato.

$x[i] \in [0, 1]$ rappresenta la porzione dell'oggetto i -esimo che deve essere prelevata.

Correttezza

Informalmente

- Assumiamo che gli oggetti siano ordinati per profitto specifico decrescente
- Sia x una soluzione ottima
- Supponiamo che $x[1] < \min(C/w[i], 1) < 1$
- Allora possiamo costruire una nuova soluzione in cui $x'[1] = \min(C/w[i], 1)$ e la proporzione di uno o più oggetti è ridotta di conseguenza
- Otteniamo così una soluzione x' di profitto uguale o superiore, visto che il profitto specifico dell'oggetto 1 è massimo

Problema della compressione

Rappresentare i dati in modo efficiente

- Impiegare il numero minore di bit per la rappresentazione
- Obiettivo: 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 "

Possibili codifiche

Esempio

- Supponiamo di avere un file di n caratteri
- Composto da caratteri nell'alfabeto `abcdef`
- Di cui conosciamo la frequenza relativa

Caratteri	a	b	c	d	e	f	Dim.
Frequenza	45%	13%	12%	16%	9%	5%	
ASCII	01100001	01100010	01100011	01100100	01100101	01100110	$8n$
Codifica 1	000	001	010	011	100	101	$3n$

Possiamo fare di meglio?

Possibili codifiche

Esempio

- Supponiamo di avere un file di n caratteri
- Composto da caratteri nell'alfabeto **abcdef**
- Di cui conosciamo la frequenza relativa

Caratteri	a	b	c	d	e	f	Dim.
Frequenza	45%	13%	12%	16%	9%	5%	
ASCII	01100001	01100010	01100011	01100100	01100101	01100110	$8n$
Codifica 1	000	001	010	011	100	101	$3n$
Codifica 2	0	101	100	111	1101	1100	$2.24n$

Costo totale: $(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot n = 2.24n$

Codifica a prefissi

Codice a prefisso

In un codice a prefisso (meglio sarebbe "senza prefissi"), **nessun codice è prefisso di un altro codice** (condizione necessaria per la decodifica).

Esempio 1

- Codice: "a" \rightarrow 0, "b" \rightarrow 10, "c" \rightarrow 11
- "babaca": $10 \cdot 0 \cdot 10 \cdot 0 \cdot 11 \cdot 0$

Esempio 2

- Codice: "a" \rightarrow 0, "b" \rightarrow 1, "c" \rightarrow 11
- 111111?

Rappresentazione ad albero per la codifica

Alcune domande

- È possibile che il testo codificato sia più lungo della rappresentazione con 3 bit?
- Esistono testi "difficili" per questa codifica?
- Come organizzare un algoritmo per la decodifica?

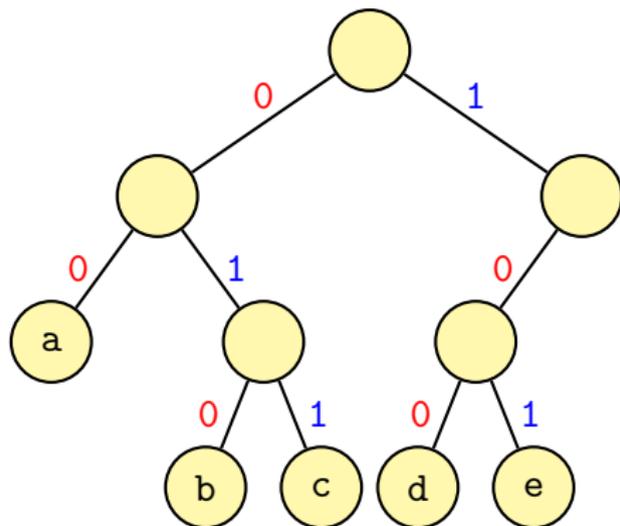
Cenni storici

- David Huffman, 1952
- Algoritmo ottimo per costruire codici prefissi
- Oggi utilizzato come complemento di altri metodi di compressione (e.g. in `pkzip`, `zip`, `winrar`)

Rappresentazione ad albero per la decodifica

Alberi binari di decodifica

- Figlio sinistro/destro: 0 / 1
- Caratteri dell'alfabeto sulle foglie



a	b	c	d	e
00	010	011	100	101

Algoritmo di decodifica

parti dalla radice

while file non è finito **do**

 leggi un bit

if bit è zero **then**

 vai a sinistra

else

 vai a destra

if nodo foglia **then**

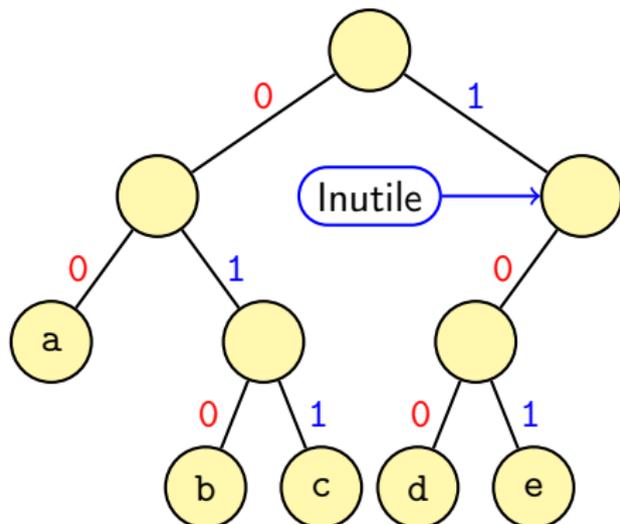
 stampa il carattere

 torna alla radice

Rappresentazione ad albero per la decodifica

Alberi binari di decodifica

- Figlio sinistro/destro: 0 / 1
- Caratteri dell'alfabeto sulle foglie



a	b	c	d	e
00	010	011	100	101

Algoritmo di decodifica

parti dalla radice

while file non è finito **do**

 leggi un bit

if bit è zero **then**

 vai a sinistra

else

 vai a destra

if nodo foglia **then**

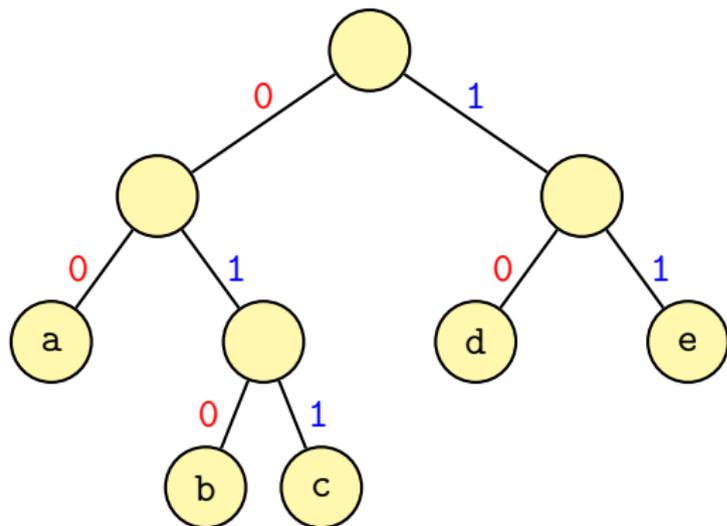
 stampa il carattere

 torna alla radice

Rappresentazione ad albero per la decodifica

Alberi binari di decodifica

- Figlio sinistro/destro: 0 / 1
- Caratteri dell'alfabeto sulle foglie



a	b	c	d	e
00	010	011	10	11

Algoritmo di decodifica

parti dalla radice

while file non è finito **do**

 leggi un bit

if bit è zero **then**

 vai a sinistra

else

 vai a destra

if nodo foglia **then**

 stampa il carattere

 torna alla radice

Definizione formale del problema

Input

- un file F composto da caratteri nell'alfabeto Σ

Quanti bit sono richiesti per codificare il file?

- Sia T un albero che rappresenta la codifica
- 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 occorrenze di c in F , allora la dimensione della codifica è

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

Algoritmo di Huffman

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

Ogni 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
- Si aggiunge al file una rappresentazione del codice, per la decodifica

Funzionamento algoritmo

- Costruire un nodo foglia per ogni carattere, etichettato con la propria frequenza

f : 5

e : 9

c : 12

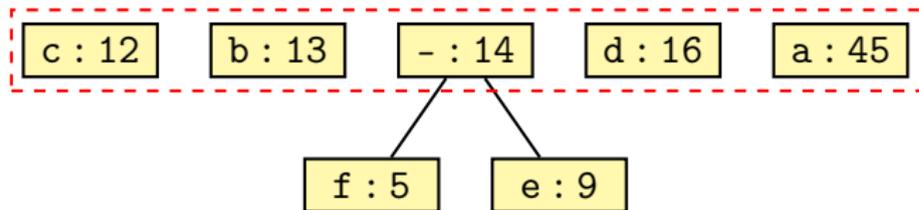
b : 13

d : 16

a : 45

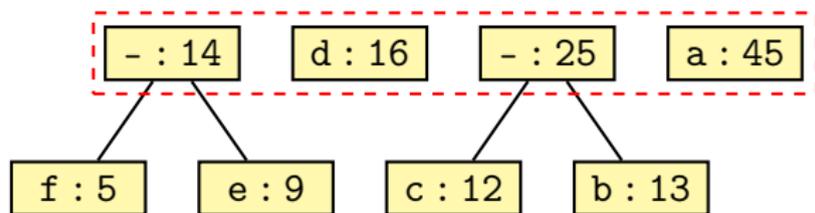
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato all'insieme



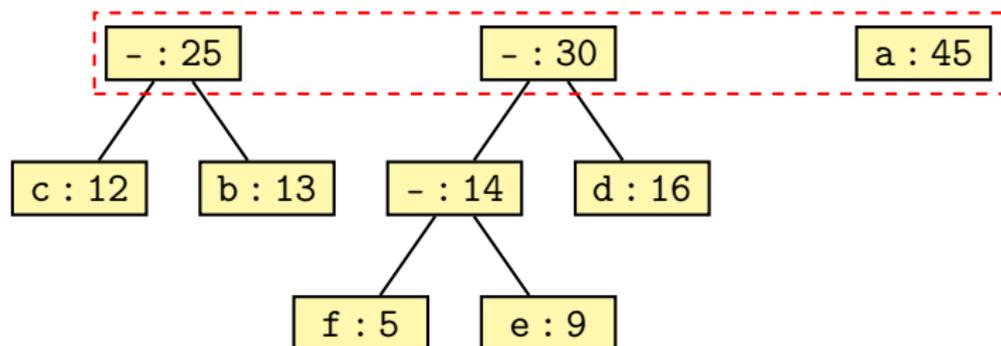
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato all'insieme



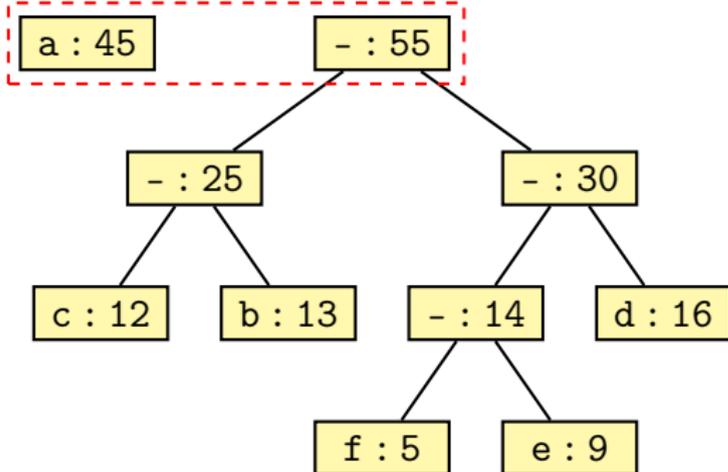
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato all'insieme



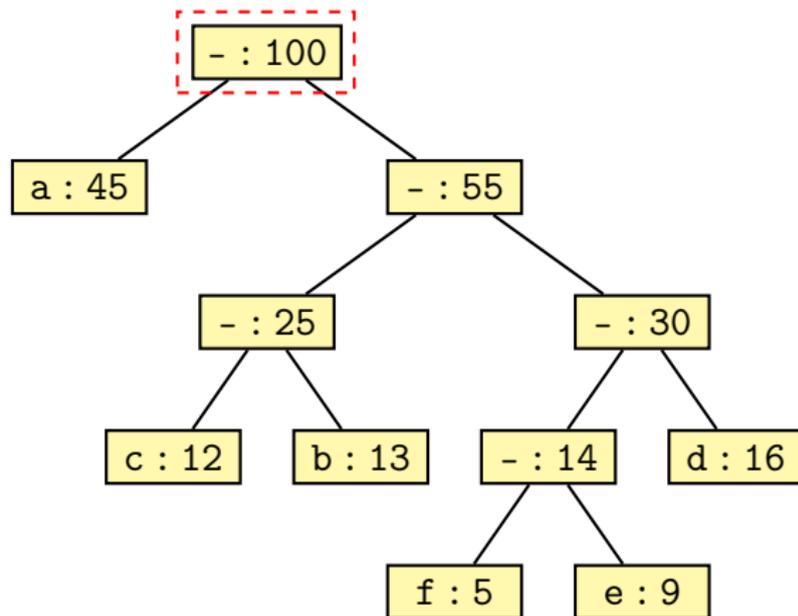
Funzionamento algoritmo

- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta "-" e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato all'insieme



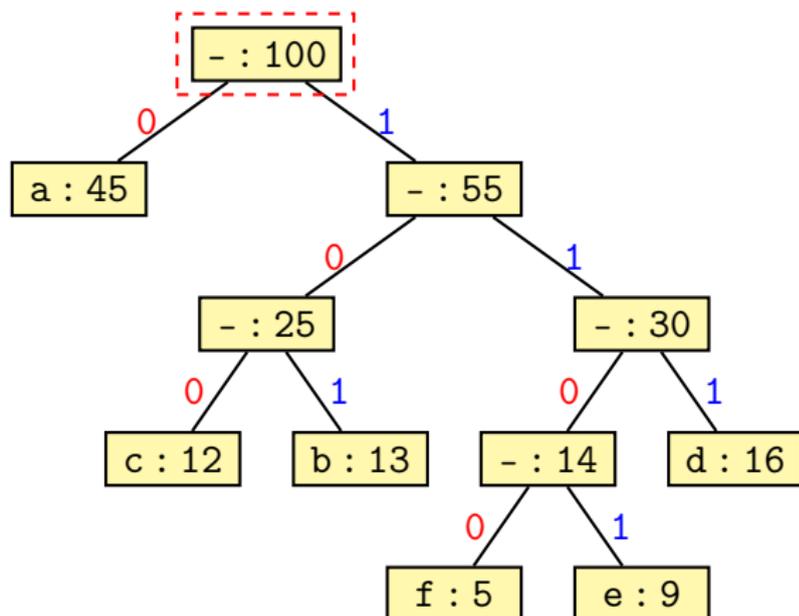
Funzionamento algoritmo

- Si termina quando resta un solo nodo



Funzionamento algoritmo

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



a	0
b	101
c	100
d	111
e	1101
f	1100

Algoritmo

```

TREE huffman(int[] c, int[] f, int n)


---


PRIORITYQUEUE Q = MinPriorityQueue()
for i = 1 to n do
  Q.insert(Tree(c[i], f[i]), f[i])
for i = 1 to n - 1 do
  z1 = Q.deleteMin()
  z2 = Q.deleteMin()
  z = Tree(nil, z1.f + z2.f)
  z.left = z1
  z.right = z2
  Q.insert(z, z.f)
return Q.deleteMin()

```

Input

n : numero caratteri

$c[]$: caratteri alfabeto

$f[]$: frequenze

TREE

c % Carattere

f % Frequenza

$left$ % Figlio sinistro

$right$ % Figlio destro

Complessità: $O(n \log n)$

Correttezza

Teorema

L'output dell'algoritmo Huffman per un dato file è un codice a prefisso ottimo

Proprietà della scelta greedy

Scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale

Sottostruttura ottima

Dato un problema sull'alfabeto Σ , è possibile costruire un sottoproblema con un alfabeto più piccolo

Scelta greedy

Ipotesi

- Siano Σ un alfabeto, f un vettore di frequenze
- Siano x, y i **due caratteri con frequenza più bassa**

Tesi

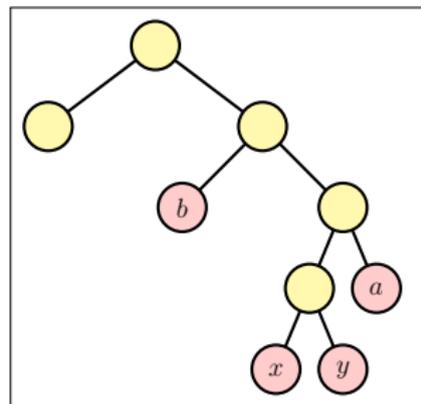
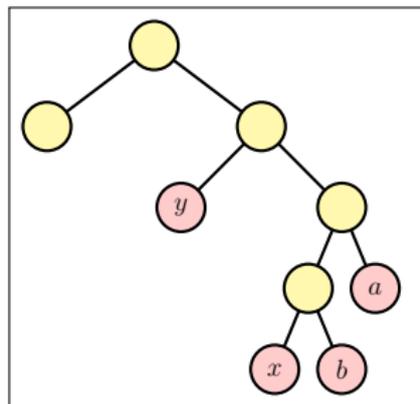
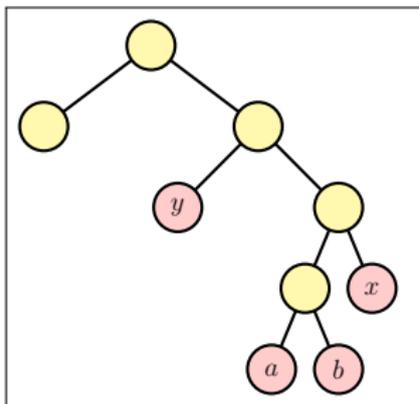
- Esiste un codice prefisso ottimo per Σ in cui x, y hanno la stessa profondità massima e i loro codici differiscono solo per l'ultimo bit (sono foglie sorelle)

Dimostrazione

- Al solito, basata sulla trasformazione di una soluzione ottima
- Supponiamo che esista un codice ottimo T in cui i due caratteri a, b con profondità massima siano diversi da x, y

Scelta greedy

- Assumiamo senza perdere di 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''



Scelta greedy

- Dimostriamo che: $C(f, T'') \leq C(f, T') \leq C(f, T)$

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

$$C(f, T') - C(f, T'') \geq 0 \quad \text{Come sopra}$$

- Ma poiché T è ottimo, sappiamo anche che: $C(f, T) \leq C(f, T'')$
- Quindi T'' è anch'esso ottimo

Albero di copertura di peso minimo

Problema

Dato un grafo pesato, determinare come interconnettere tutti i suoi nodi minimizzando il costo del peso associato ai suoi archi.

- Albero di copertura (di peso) minimo
- Albero di connessione (di peso) minimo
- Minimum spanning tree

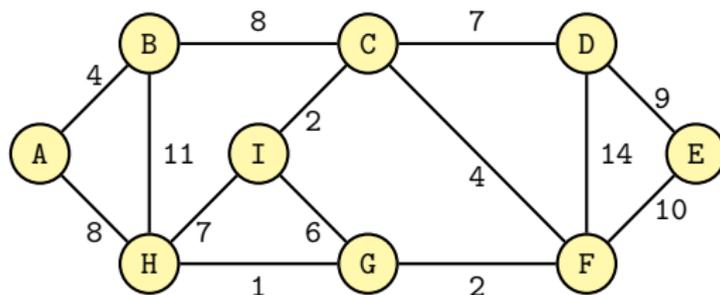
Esempio di applicazione

Una compagnia di telecomunicazioni deve stendere una nuova rete in un quartiere; deve seguire le connessioni esistenti (la rete stradale) e ogni arco ha un costo associato distinto (costi di scavo, etc.)

Definizione del problema

Input

- $G = (V, E)$: un grafo non orientato e connesso
- $w : V \times V \rightarrow \mathbb{R}$: una funzione di peso (costo di connessione)
 - se $(u, v) \in E$, allora $w(u, v)$ è il peso dell'arco (u, v)
 - se $(u, v) \notin E$, allora $w(u, v) = +\infty$
- Poiché G non è orientato, $w(u, v) = w(v, u)$



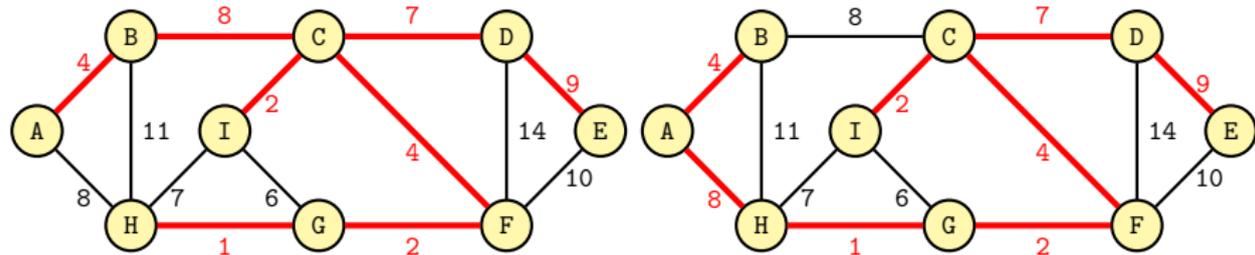
Definizione del problema

Output: albero di copertura di peso minimo

Trovare l'albero di copertura il cui **peso totale** sia minimo rispetto a ogni altro albero di copertura.

$$w(T) = \sum_{(u,v) \in E_T} w(u,v)$$

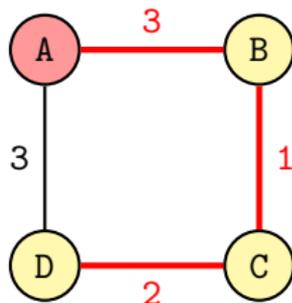
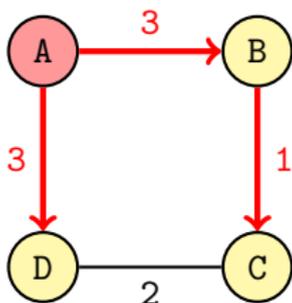
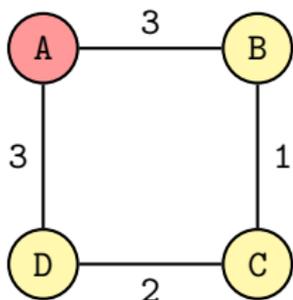
Non è detto che l'albero di copertura minimo sia univoco



Cammini minimi / alberi di copertura di peso minimo

Questi due alberi di copertura sono identici?

- un **albero dei cammini minimi** da sorgente A
- un **albero di copertura di peso minimo**



Algoritmo generico

Schema della lezione

- Progettiamo un algoritmo di tipo "ingordo" generico
- Mostriamo due "istanze" di questo algoritmo: **Kruskal** e **Prim**

Approccio

L'idea è di accrescere un sottoinsieme A di archi in modo tale che venga sempre rispettata la seguente invariante:

- A è un sottoinsieme di qualche albero di connessione minimo

Algoritmo generico

Arco sicuro

Un arco (u, v) è detto **sicuro per A** se $A \cup \{(u, v)\}$ è ancora un sottoinsieme di qualche albero di connessione minimo.

```
SET mst-generico(GRAPH  $G$ , int[]  $w$ )
```

```
SET  $A = \emptyset$ 
```

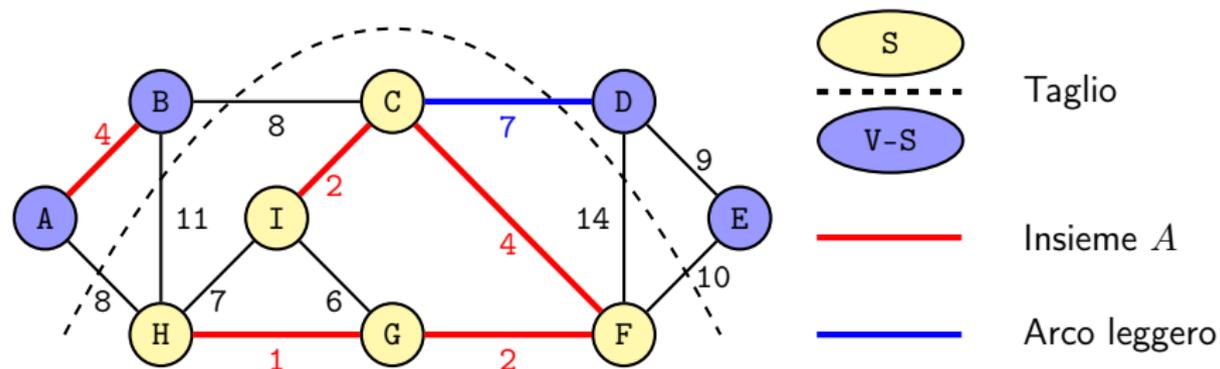
```
while  $A$  non forma un albero di copertura do
```

```
┌   trova un arco sicuro  $(u, v)$   
└    $A = A \cup \{(u, v)\}$ 
```

```
return  $A$ 
```

Definizioni

- Un **taglio** $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V in due sottoinsiemi disgiunti
- Un arco (u, v) **attraversa** il taglio se $u \in S$ e $v \in V - S$
- Un taglio **rispetta** un insieme di archi A se nessun arco di A attraversa il taglio
- Un arco che attraversa un taglio è **leggero** nel taglio se il suo peso è minimo fra i pesi degli archi che attraversano un taglio



Arco sicuro

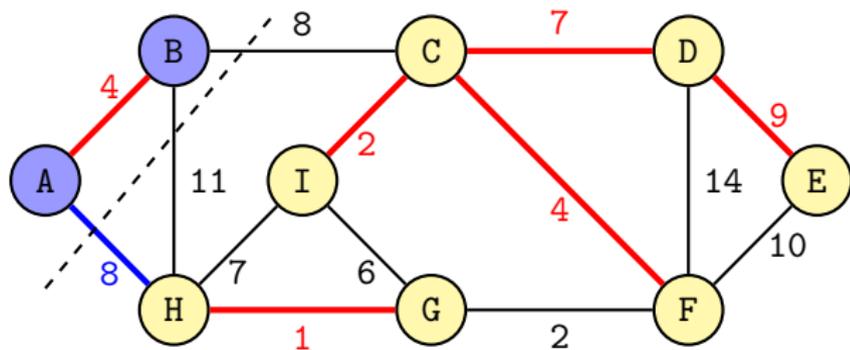
Teorema

- Sia $G = (V, E)$ un grafo non orientato e connesso
- Sia $w : V \times V \rightarrow \mathbb{R}$
- Sia $A \subseteq E$ un sottoinsieme contenuto in un qualche albero di copertura minimo per G
- Sia $(S, V - S)$ un qualunque taglio che rispetta A
- Sia (u, v) un arco leggero che attraversa il taglio

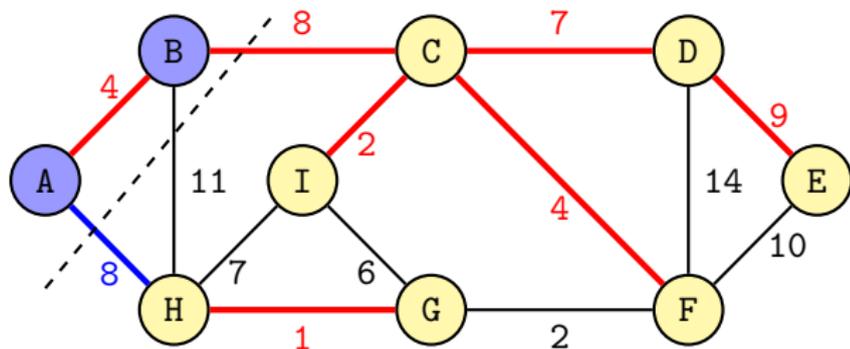
\Rightarrow l'arco (u, v) è sicuro per A

Esempio: arco non sicuro perché il taglio non rispetta A

Arco blu sicuro

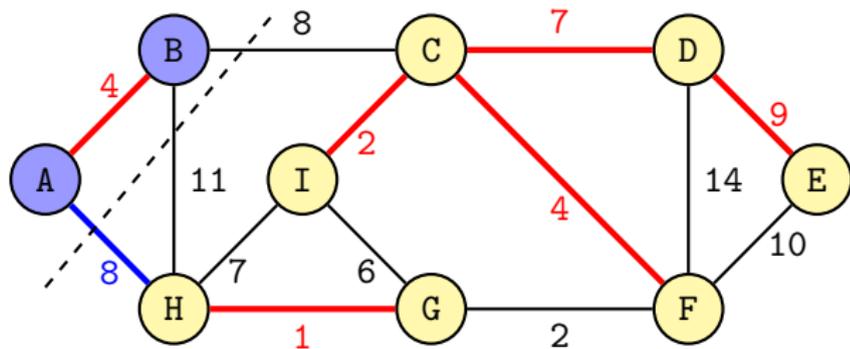


Arco blu non sicuro

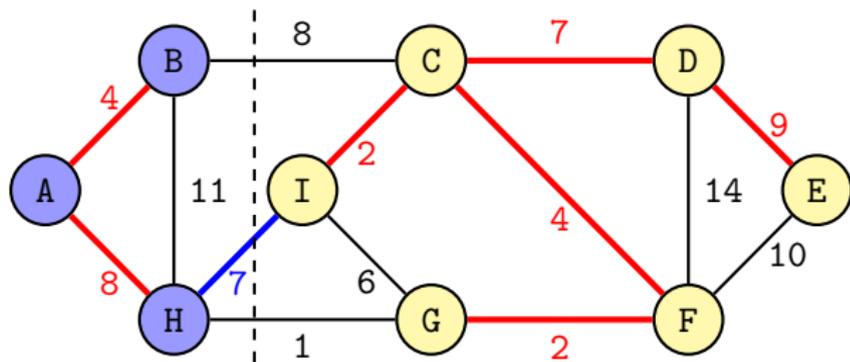


Esempio: arco non sicuro perché non leggero

Arco blu sicuro



Arco blu non sicuro



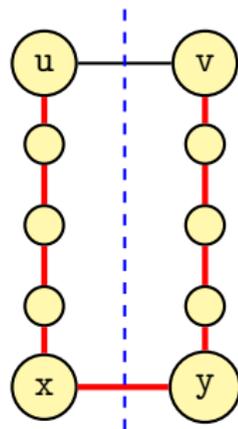
Dimostrazione

Dimostrazione

Sia T un albero di copertura minimo che contiene A . Due casi:

- $(u, v) \in T$: allora (u, v) è sicuro per A
- $(u, v) \notin T$: trasformiamo T in un albero T' contenente (u, v) e dimostriamo che T' è un albero di copertura minimo

- u, v sono connessi da un **cammino** $C \subseteq T$
(per definizione di albero)
- u, v stanno in lati opposti del taglio
((u, v) attraversa il taglio)
- $\exists(x, y) \in C$ che attraversa il taglio



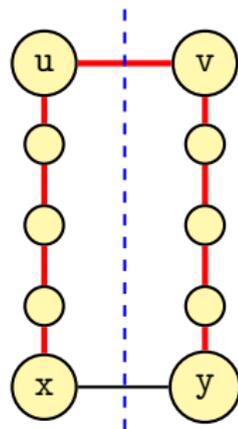
Dimostrazione

Dimostrazione

Sia T un albero di copertura minimo che contiene A . Due casi

- $(u, v) \in T$: allora (u, v) è sicuro per A
- $(u, v) \notin T$: trasformiamo T in un albero T' contenente (u, v) e dimostriamo che T' è un albero di copertura minimo

- $T' = T - \{(x, y)\} \cup \{(u, v)\}$
- T' è un albero di copertura
- $w(T') \leq w(T)$ (perchè $w(u, v) \leq w(x, y)$)
- $w(T) \leq w(T')$ (perchè T minimo)



Archi sicuri

Corollario

- Sia $G = (V, E)$ un grafo non orientato e connesso
- Sia $w : V \times V \rightarrow \mathbb{R}$
- Sia $A \subseteq E$ un sottoinsieme contenuto in un qualche albero di copertura minimo per G
- Sia C una componente connessa (un albero) nella foresta $G_A = (V, A)$
- Sia (u, v) un arco leggero che connette C a qualche altra componente in G_A

\Rightarrow l'arco (u, v) è sicuro per A

Algoritmo di Kruskal

Idea

- Ingrandire sottoinsiemi disgiunti di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo
- Si individua un arco sicuro scegliendo un arco (u, v) di peso minimo tra tutti gli archi che connettono due distinti alberi (componenti connesse) della foresta
- L'algoritmo è greedy perché ad ogni passo si aggiunge alla foresta un arco con il peso minore

Implementazione

- Si utilizza una struttura dati Merge-Find Set

Algoritmo di Kruskal

```
SET kruskal(EDGE[] A, int n, int m)
```

```
SET T = Set()
```

```
MFSET M = Mfset(n)
```

```
{ ordina A[1, ..., m] in modo che A[1].weight ≤ ... ≤ A[m].weight }
```

```
int count = 0
```

```
int i = 1
```

```
% Termina quando l'albero ha n - 1 archi o non ci sono più archi
```

```
while count < n - 1 and i ≤ m do
```

```
    if M.find(A[i].u) ≠ M.find(A[i].v) then
```

```
        M.merge(A[i].u, A[i].v)
```

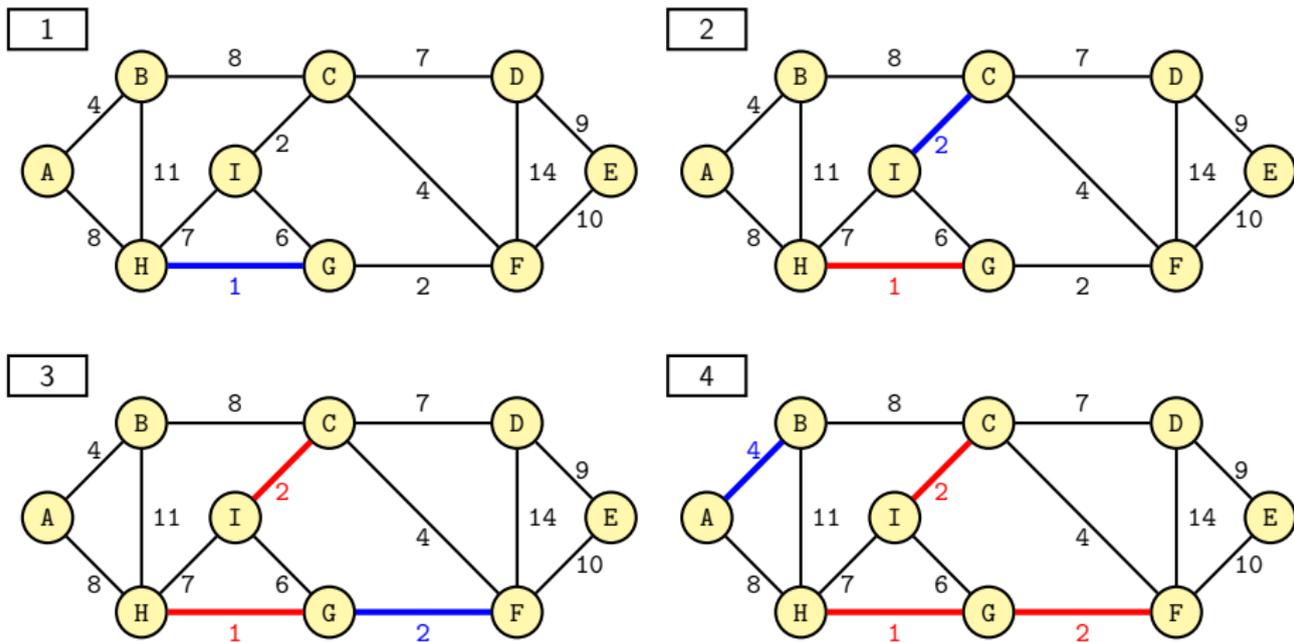
```
        T.insert(A[i])
```

```
        count = count + 1
```

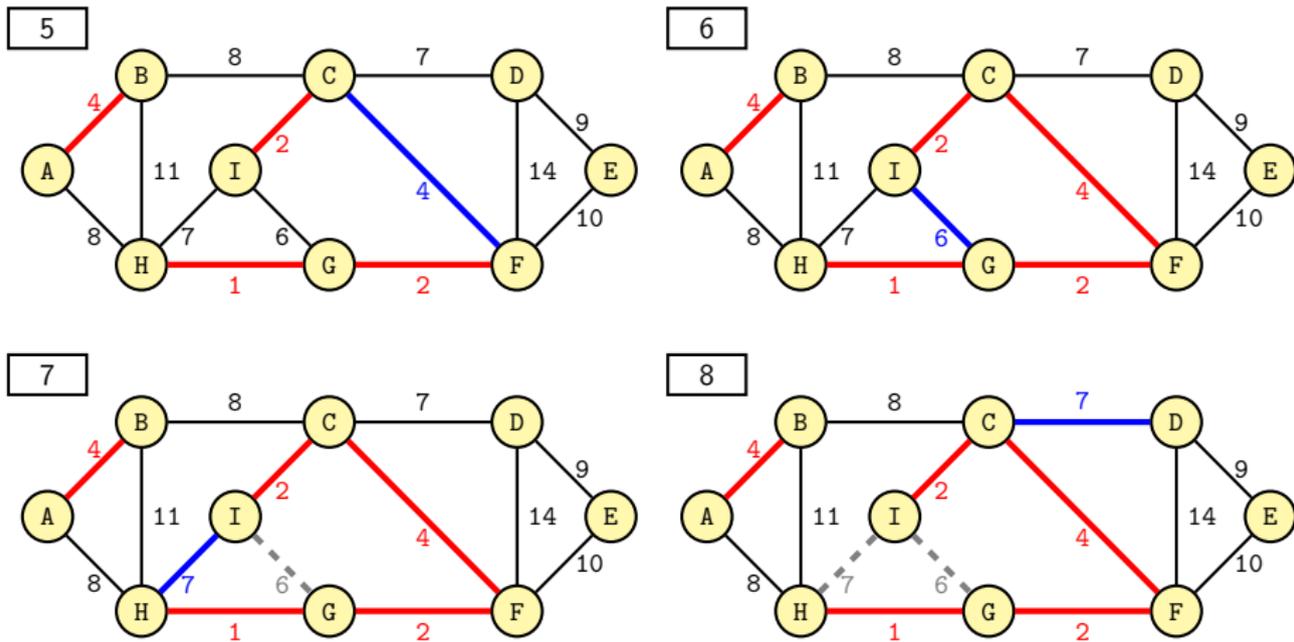
```
    i = i + 1
```

```
return T
```

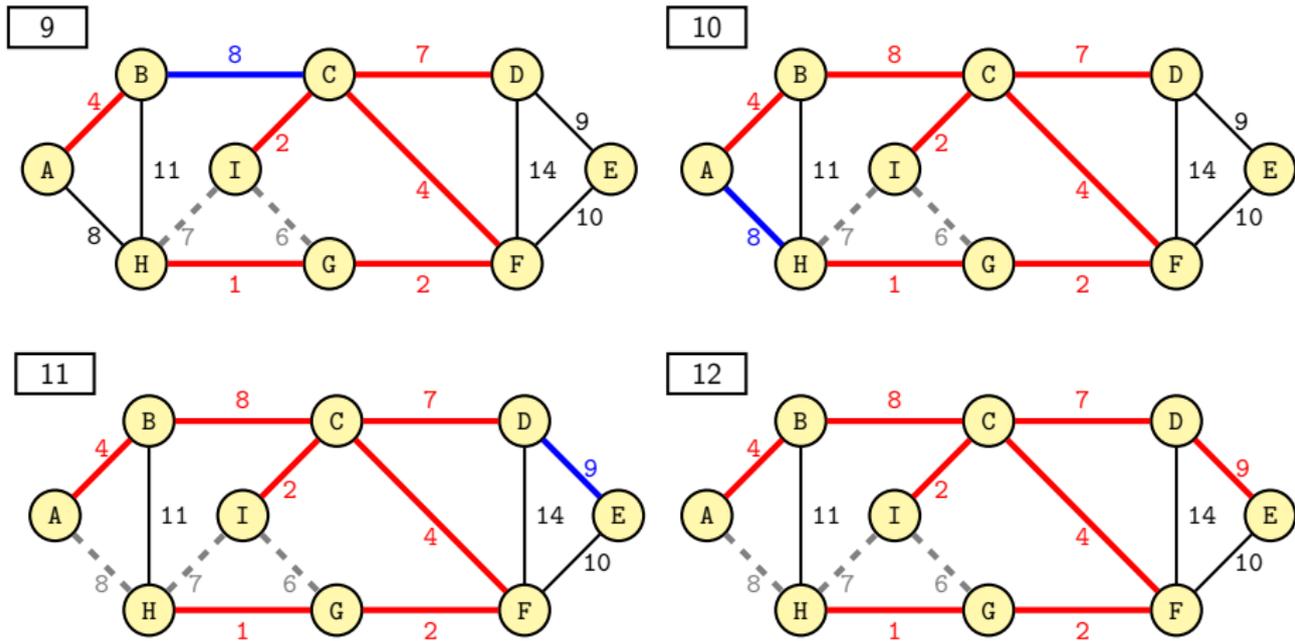
Esempio



Esempio



Esempio



Analisi della complessità algoritmo di Kruskal

- Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati per Merge-Find Set
- Utilizziamo la versione con **euristica sul rango + compressione**, (*) le cui operazioni hanno costo ammortizzato costante

Fase	Volte	Costo
Inizializzazione	1	$O(n)$
Ordinamento	1	$O(m \log m)$
Operazioni <code>find()</code> , <code>merge()</code>	$O(m)$	$O(1)^{(*)}$

- Totale:
 $O(n + m \log m + m) = O(m \log m) = O(m \log n^2) = O(m \log n)$

Algoritmo di Prim

Idea

- L'algoritmo di Prim procede mantenendo in A un singolo albero
- L'albero parte da un vertice arbitrario r (la radice) e cresce fino a quando non ricopre tutti i vertici
- Ad ogni passo viene aggiunto un arco leggero che collega un vertice in V_A con un vertice in $V - V_A$, dove V_A è l'insieme di nodi raggiunti da archi in A

Correttezza

- $(V_A, V - V_A)$ è un taglio che rispetta A (per definizione)
- Per il corollario, gli archi leggeri che attraversano il taglio sono sicuri

Implementazione

Struttura dati per i nodi non ancora nell'albero

- Durante l'esecuzione, i vertici non ancora nell'albero si trovano in una coda con min-priorità Q ordinata in base alla seguente definizione di priorità
- "La priorità del nodo v è il peso minimo di un arco che collega v ad un vertice nell'albero, o $+\infty$ se tale arco non esiste"

Albero registrato come **vettore dei padri**

- Ogni nodo v mantiene un puntatore al padre $p[v]$
- A è mantenuto implicitamente:

$$A = \{(v, p[v]) \mid v \in V - Q - \{r\}\}$$

Algoritmo di Prim

```

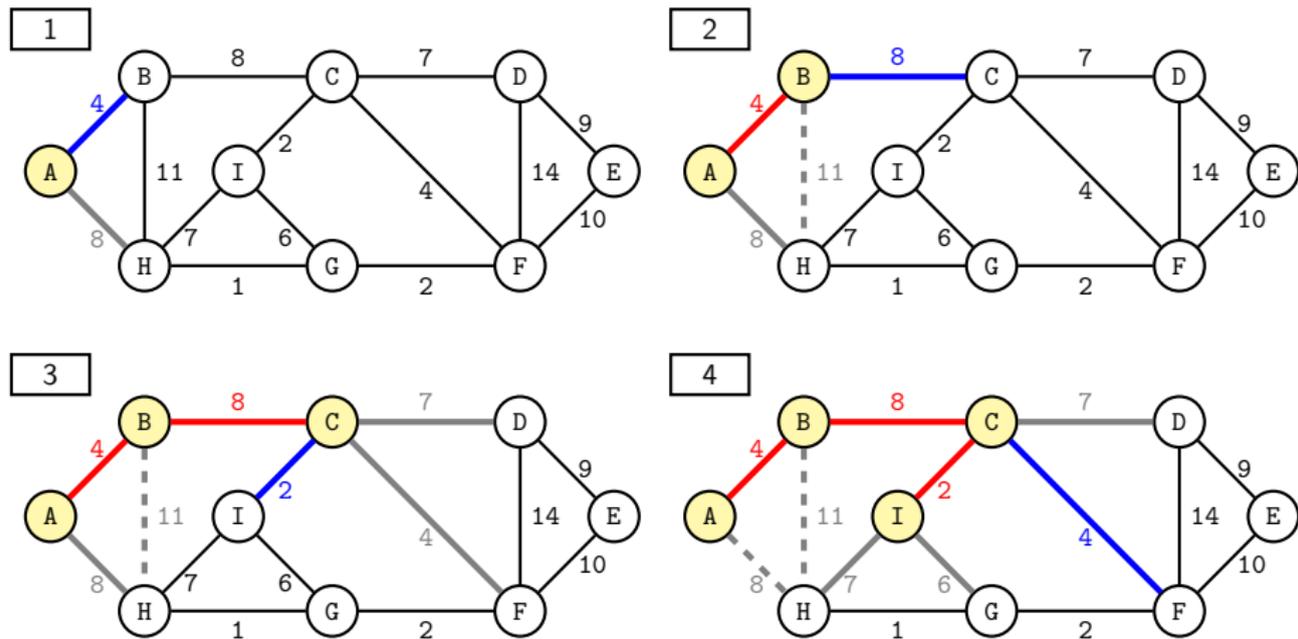
int[] prim(GRAPH  $G$ , NODE  $r$ )


---

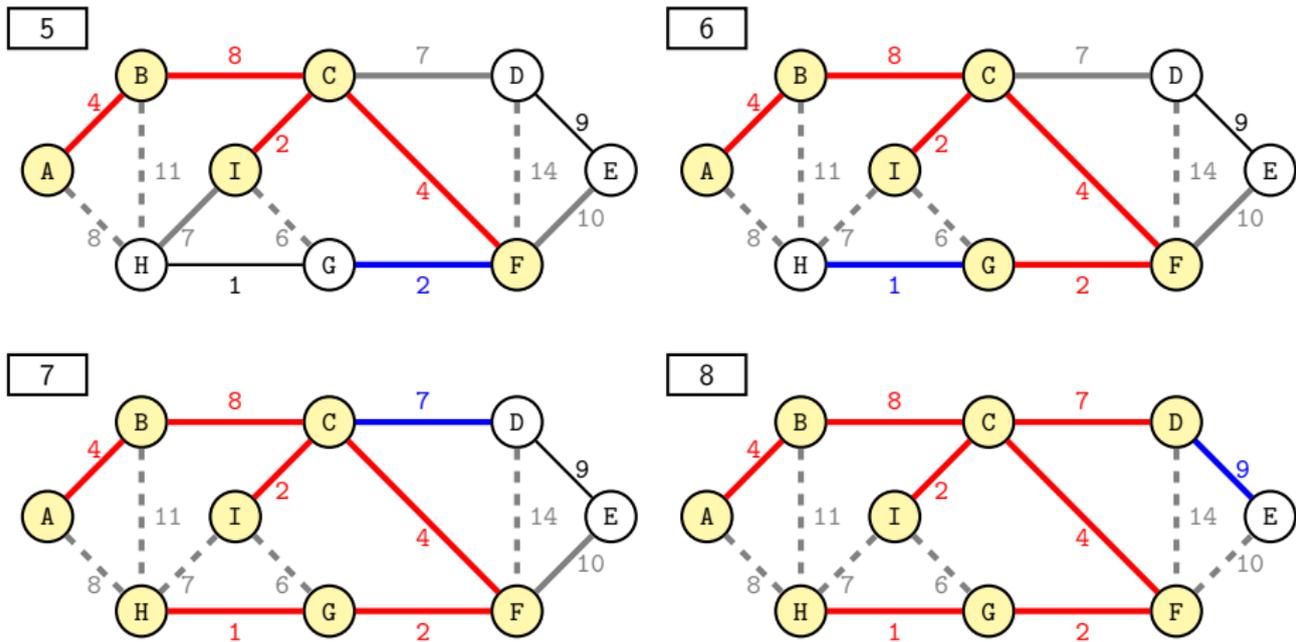

PRIORITYQUEUE  $Q$  = MinPriorityQueue()
PRIORITYITEM[]  $pos$  = new PRIORITYITEM[1 ...  $G.n$ ]
int[]  $p$  = new int[1 ...  $G.n$ ]
foreach  $u \in G.V() - \{r\}$  do
     $pos[u]$  =  $Q.insert(u, +\infty)$ 
 $pos[r]$  =  $Q.insert(r, 0)$ 
 $p[r]$  = 0
while not  $Q.isEmpty()$  do
    NODE  $u$  =  $Q.deleteMin()$ 
     $pos[u]$  = nil
    foreach  $v \in G.adj(u)$  do
        if  $pos[v] \neq \mathbf{nil}$  and  $w(u, v) < pos[v].priority$  then
             $Q.decrease(pos[v], w(u, v))$ 
             $p[v]$  =  $u$ 
return  $p$ 

```

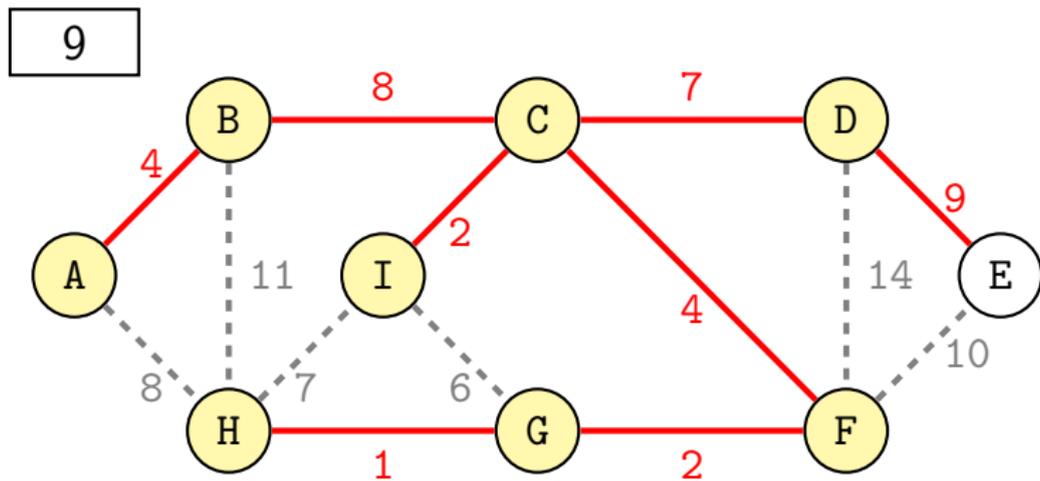
Esempio



Esempio



Esempio



Algoritmo di Prim: Analisi

L'efficienza dell'algoritmo di Prim dipende dalla coda con priorità

- Se si utilizza uno **heap binario**:

Fase	Volte	Costo
Inizializzazione	1	$O(n \log n)$
deleteMin()	$O(n)$	$O(\log n)$
decreasePriority()	$O(m)$	$O(\log n)$

Tempo totale: $O(n + n \log n + m \log n) = O(m \log n)$,
asintoticamente uguale a quello di Kruskal.

- Cosa succede se la coda con priorità è implementata tramite **vettore non ordinato**?

Discussione

Vero o falso

- L'arco con peso minimo è sicuro
- L'arco con il secondo peso minimo è sicuro
- L'arco con il terzo peso minimo è sicuro

Albero di copertura minima in un piano

- Input: n punti nel piano
- Il peso di una coppia di punti è dato dalla distanza euclidea fra di essi
- Trovare un insieme di connessioni di peso minimo
- Da non confondere con gli **Steiner tree**

Applicazioni

Applicazioni dirette per la progettazione

- Reti di telecomunicazione
- Reti idriche
- Reti di trasporto
- Reti elettriche

Alcuni utilizzi particolari

- Segmentazione di immagini
- Riconoscimento scrittura manuale
- Disegno di circuiti elettronici
- Progettazione tassonomie

Prospettiva storica

- $O(m \log n)$:
 - Primo algoritmo: Boruvka (1926)
 - Kruskal (1956)
 - Prim (1957), ma anche Jarnik (1930)
- Fredman-Tarjan (1987): $O(m + n \log n)$:
 - Modifica di Prim che utilizza gli heap di Fibonacci
- Chazelle (2000): $O(m\alpha(m, n) + n \log n)$:
 - Inversa della funzione di Ackermann
- $O(m + n)$:
 - Algoritmo probabilistico di Karger, Klein, Tarjan (1995)
 - Vari algoritmi in tempo lineare per casi particolari
 - Questione aperta: soluzioni lineari in tempo deterministico?

Conclusioni

Vantaggi

- Semplici da programmare
- Molto efficienti
- Quando è possibile dimostrare la proprietà di scelta ingorda, danno la soluzione ottima
- La soluzione sub-ottima può essere accettabile

Svantaggi

- Non sempre applicabili se si vuole la soluzione ottima