**MODULO II ALGORITMI (Greedy)**

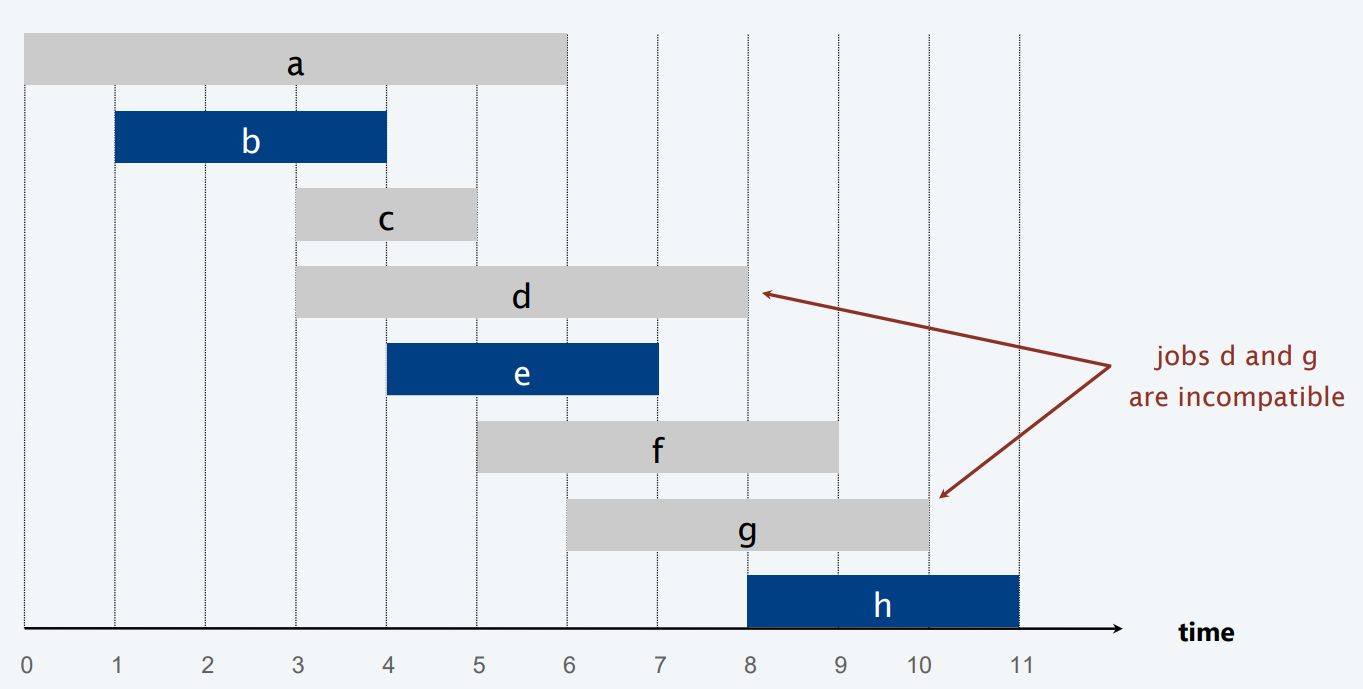
## **Scheduling di intervalli:**

### **Definizione del problema:**

**Lavoro:** Un'attività con un tempo di inizio (sj) e un tempo di fine (fj).

**Compatibilità:** Due lavori sono compatibili se non si sovrappongono nel tempo.

**Obiettivo:** Trovare il massimo sottoinsieme di lavori **reciprocamente compatibili**.



### **Input:**

* Un insieme di n intervalli I1, ..., In.
* Ogni intervallo Ii ha un tempo di inizio (Si) e un tempo di fine (Fi).

### **Soluzione fattibile:**

* Un sottoinsieme S degli intervalli che sono **reciprocamente compatibili**.
* In altre parole, per ogni Ii, Ij in S, Ii non si sovrappone a Ij.

### **Misura da massimizzare:**

* Il numero di intervalli schedulati, ovvero la **cardinalità** di S.

## 

## 

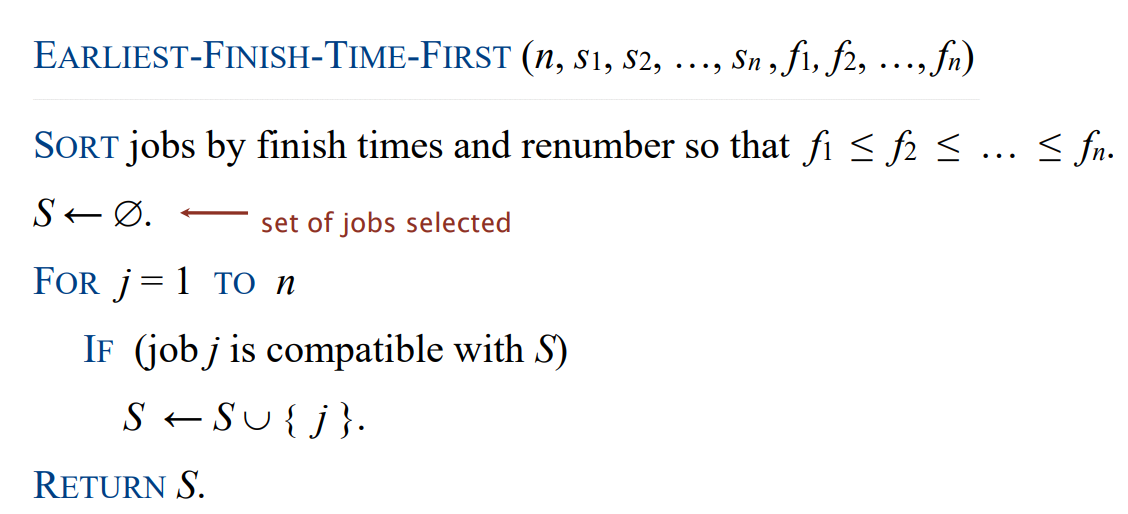
## **Algoritmi Greedy per lo Scheduling di Intervalli**

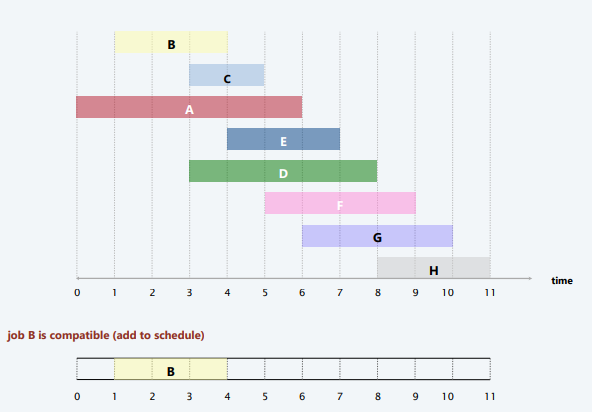
**Schema:**

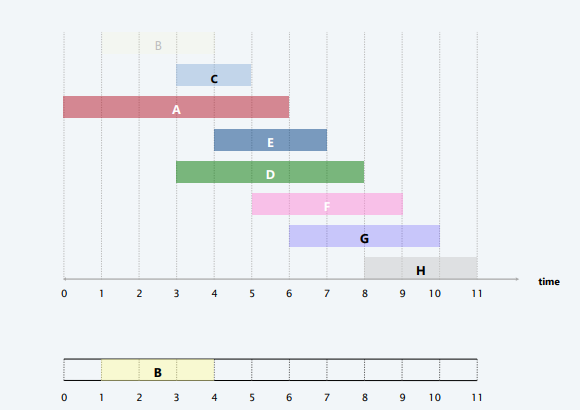
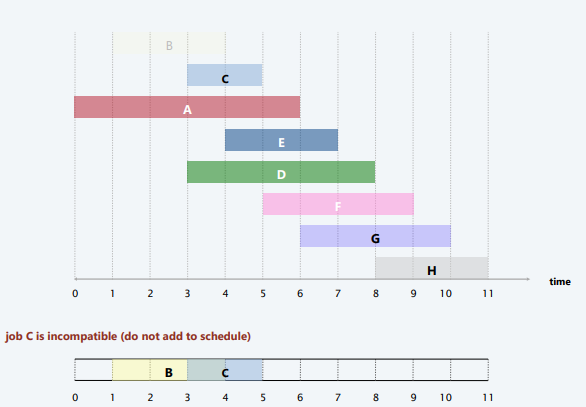
1. **Selezione Ordine Naturale:** Scegli un criterio per ordinare i lavori in modo naturale.
2. **Iterazione e Scelta Lavori Compatibili:** Scorri i lavori nell'ordine stabilito.
   * Aggiungi un lavoro alla soluzione **solo se** è compatibile con tutti i lavori già selezionati.
3. **Soluzione:** Il ciclo termina quando tutti i lavori sono stati esaminati. La collezione di lavori aggiunti è la soluzione.

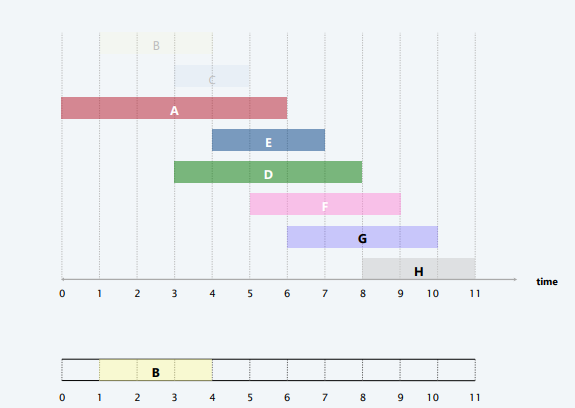
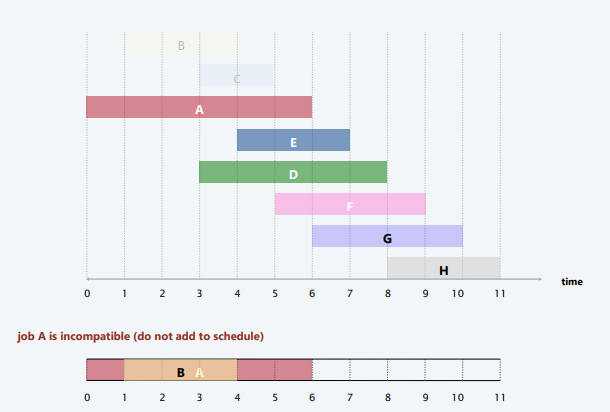
**Strategie di Ordinazione Greedy:**

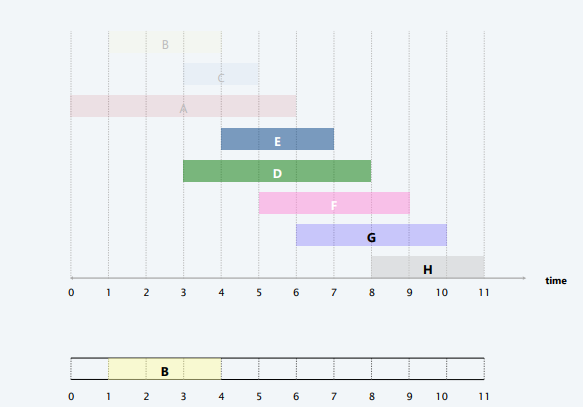
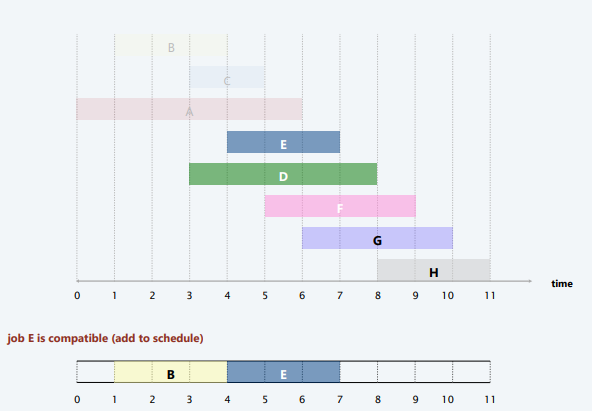
* **Earliest start time [Tempo di inizio più precoce] (sj):** Ordina i lavori in base al tempo di inizio crescente (sj).
* **Earliest finish time [Tempo di fine più precoce] (fj):** Ordina i lavori in base al tempo di fine crescente (fj).
* **Shortest interval [Intervallo più breve] (fj - sj):** Ordina i lavori in base alla differenza tra tempo di fine e inizio (fj - sj), crescente.
* **Fewest conflicts [Minimi conflitti]:** Per ogni lavoro, conta quanti altri lavori con cui è in conflitto. Ordina i lavori in base al numero crescente di conflitti.

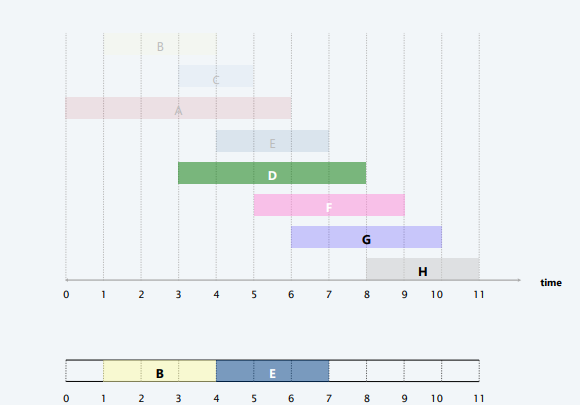
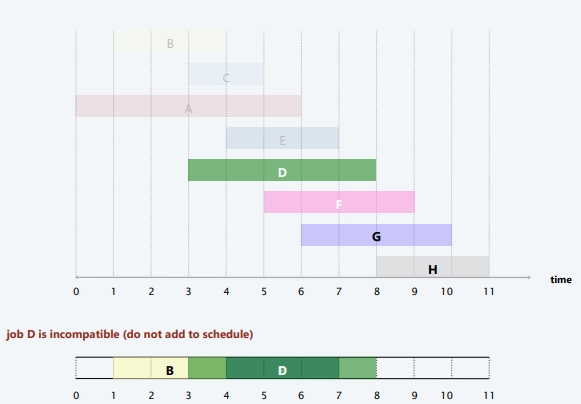
**Interval scheduling: earliest-finish-time-first   
ALGORITMO:  
**

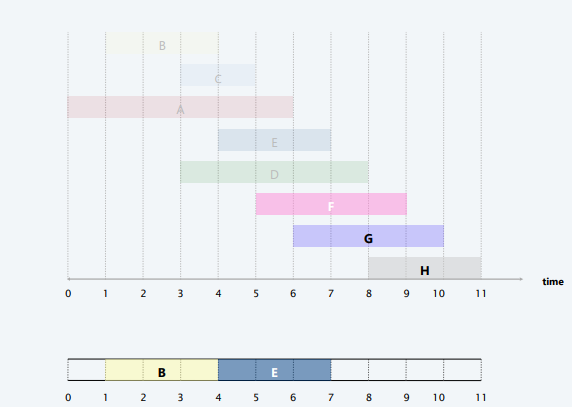
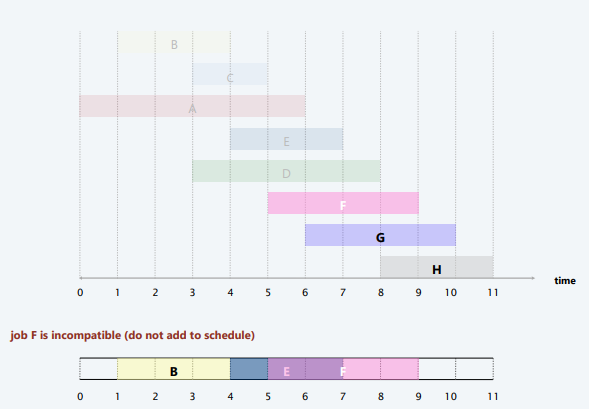
** **

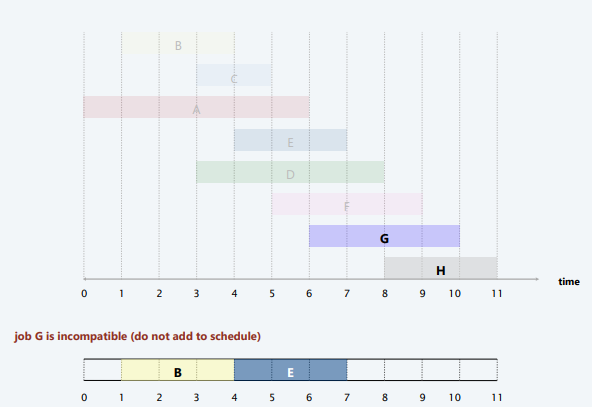
 

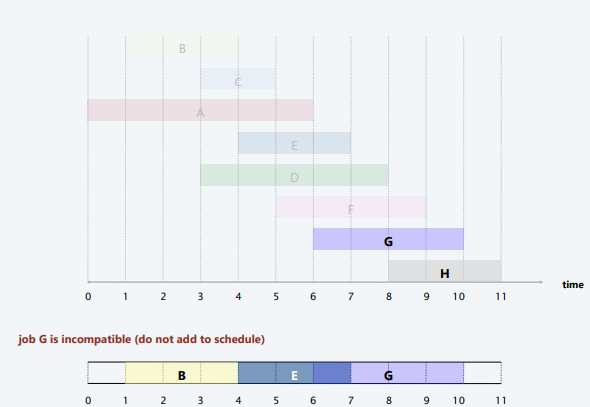
 

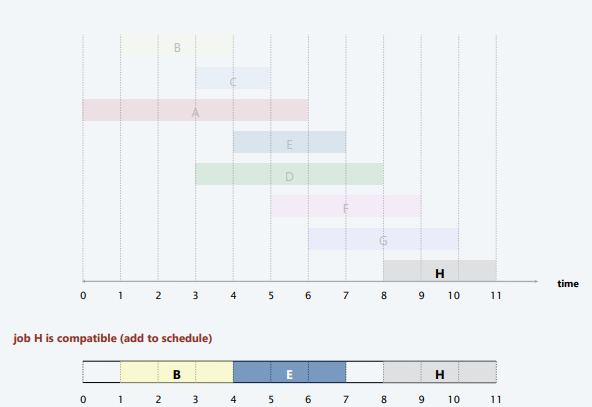
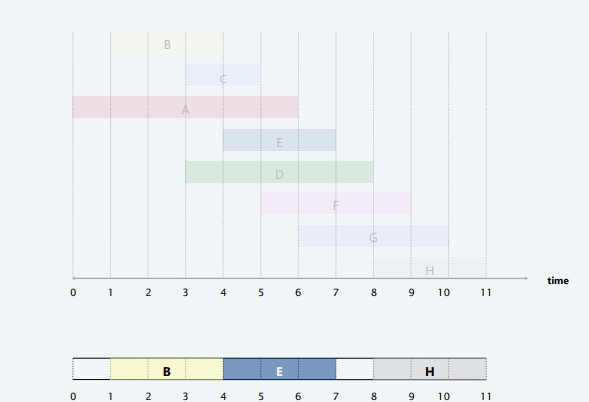
 

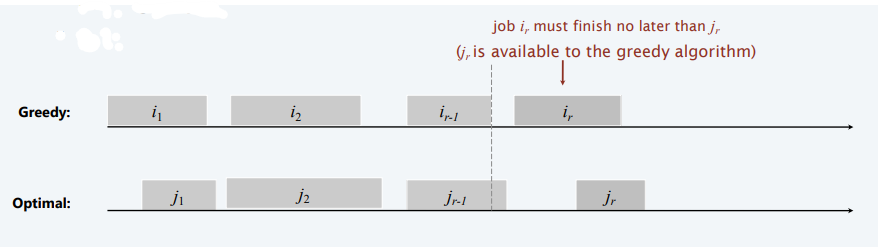
 

**Proposizione: algoritmo "tempo di fine più precoce" (EFT)**

* **L'algoritmo EFT può essere implementato in un tempo O(n log n). Questo viene ottenuto attraverso:**
  + **Tenuta traccia dell'ultimo lavoro j\* aggiunto alla soluzione S.**
  + **Verifica della compatibilità di un nuovo lavoro j con S: sj ≥ fj\* (il tempo di inizio di j deve essere maggiore o uguale al tempo di fine dell'ultimo lavoro in S).**
  + **L'ordinamento dei lavori in base ai tempi di fine richiede un tempo O(n log n) utilizzando algoritmi di ordinamento efficienti come quicksort o mergesort.**

## **Lemma:**

* **Per ogni lavoro ir selezionato dall'algoritmo greedy (ordinato per tempi di fine) in posizione r (dove 1 <= r <= k), il suo tempo di fine f(ir) è minore o uguale al tempo di fine f(jr) del lavoro corrispondente jr in una soluzione ottimale (anche ordinata per tempi di fine) in posizione r.**

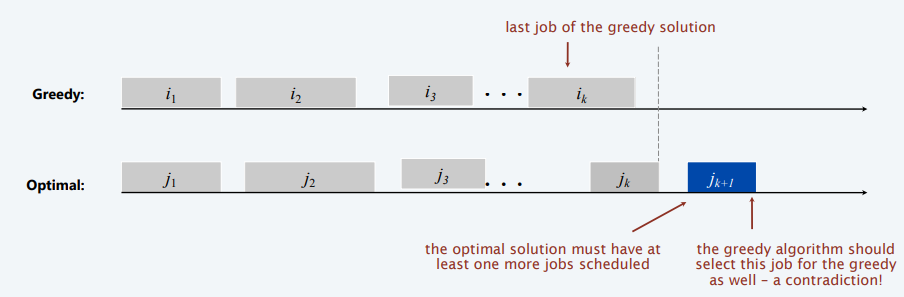
****

## **Teorema:**

* **L'algoritmo EFT è ottimale per la schedulazione di intervalli. In altre parole, trova il numero massimo di lavori non sovrapposti che possono essere schedulati all'interno di un dato insieme di lavori.**

**Dimostrazione (per contraddizione):**

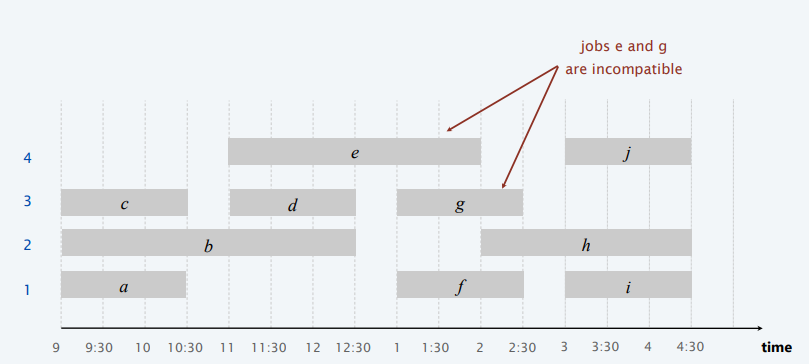
* **Supponiamo per assurdo che l'algoritmo EFT non sia ottimale.**
* **Siano i1, i2, ..., ik i lavori selezionati dall'algoritmo greedy, ordinati in base ai loro tempi di fine.**
* **Siano j1, j2, ..., jm i lavori in una soluzione ottimale, anch'essi ordinati in base ai tempi di fine.**
* **Se l'algoritmo EFT non è ottimale, allora la soluzione ottimale deve contenere più lavori della soluzione greedy (cioè m > k).**

****

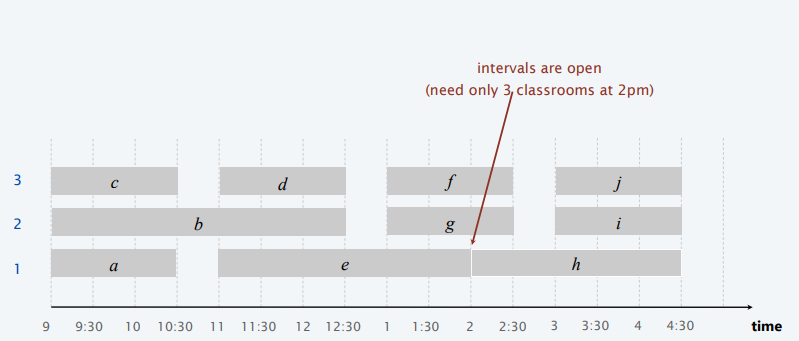
**Interval Partitioning**  
・La lezione j inizia a sj e termina a fj

. Obiettivo: trovare il numero minimo di aule per programmare tutte le lezioni in modo che non ci siano due lezioni alla stessa ora nella stessa aula.

Questo programma utilizza 4 aule per programmare 10 lezioni.



Questo programma utilizza 3 aule per programmare 10 lezioni.



* **Input**:

- Un insieme di n intervalli I1,...,In

- l'intervallo Li ha un tempo di inizio Si e un tempo di fine Fi

* **Soluzione fattibile**:

- Una partizione degli intervalli in sottoinsiemi (chiamati classi) C1,...,Cd tali che ogni Ci contenga intervalli reciprocamente compatibili

* **Misura (da minimizzare)**:

- numero di aule, i.e. d

**Interval Partitioning : algoritmi greedy**

Modello greedy. Considerare le lezioni in un ordine naturale. Assegnare ogni

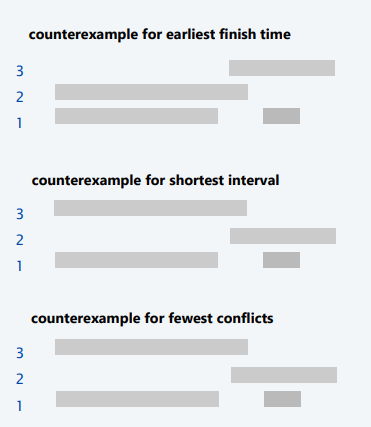
lezione a un'aula disponibile (quale?); assegnare una nuova aula se non ce ne sono di disponibili.

・**Earliest start time** [Orario di inizio più precoce] Considerare le lezioni in ordine crescente di sj.

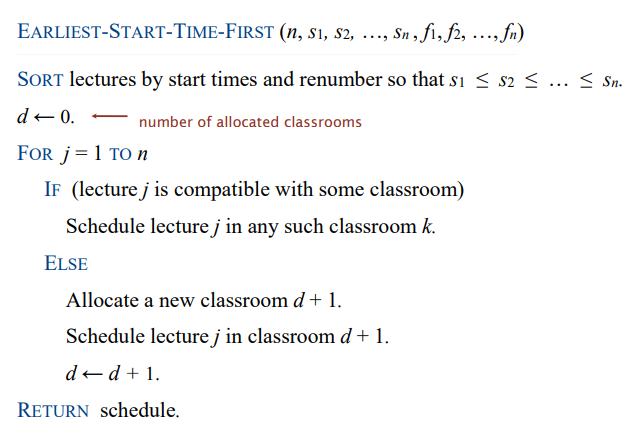
・**Earliest finish time** [Orario di fine più precoce] Considerare le lezioni in ordine crescente di fj.

・**Shortest interval** [Intervallo più breve] Considerare le lezioni in ordine crescente di fj - sj.

・**Fewest conflicts** [Meno conflitti] Per ogni lezione j, contare il numero di lezioni in conflitto cj. Programmare in ordine crescente di cj.



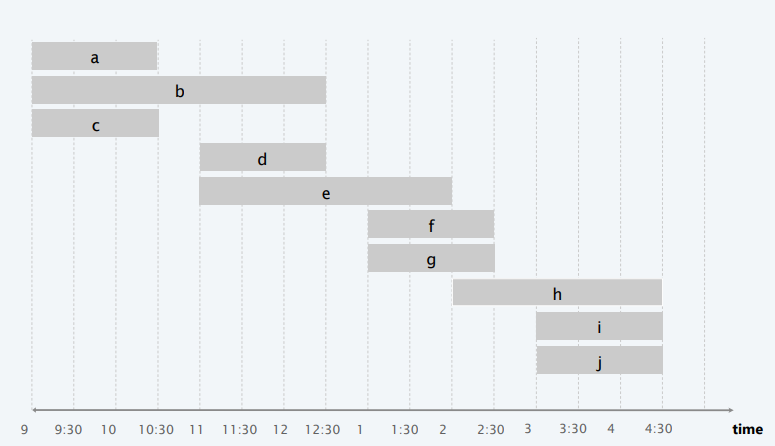
**Interval partitioning: earliest-start-time-first algorithm**

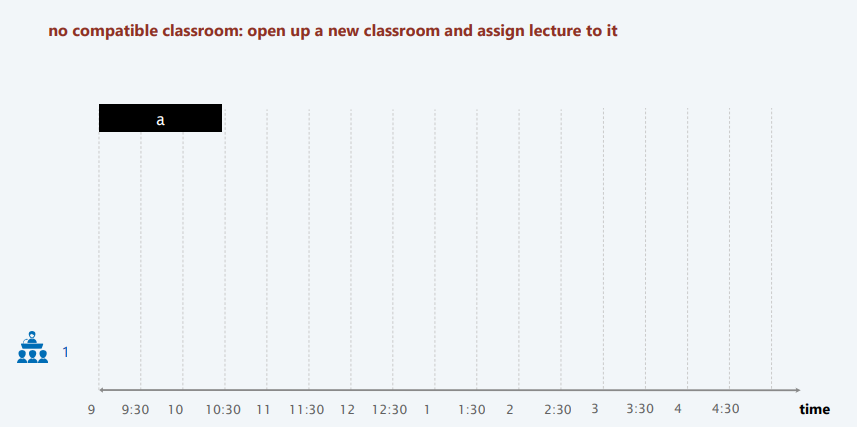
****

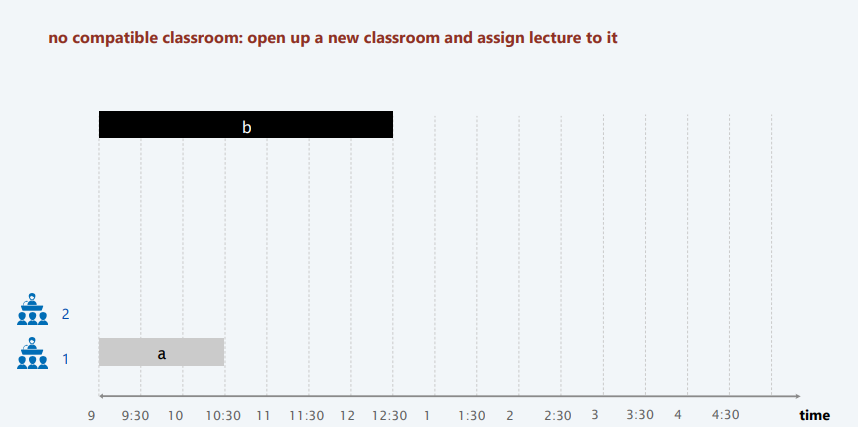
**Considerare le lezioni in ordine di orario:**

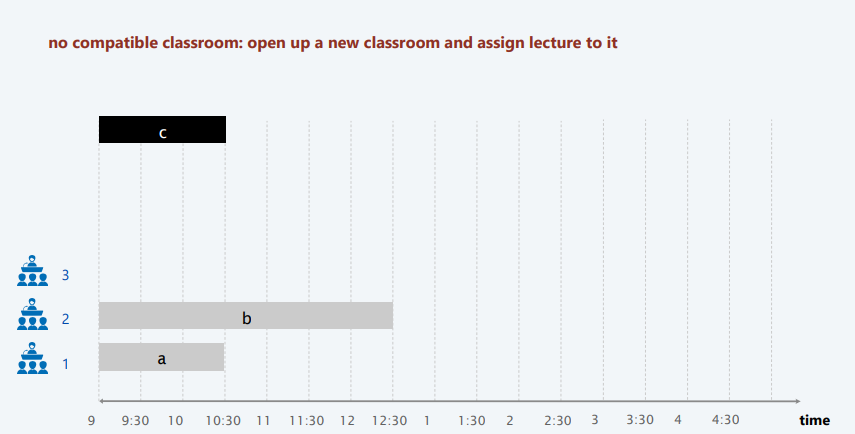
**Assegnare la lezione successiva a un'aula compatibile (se esiste).**

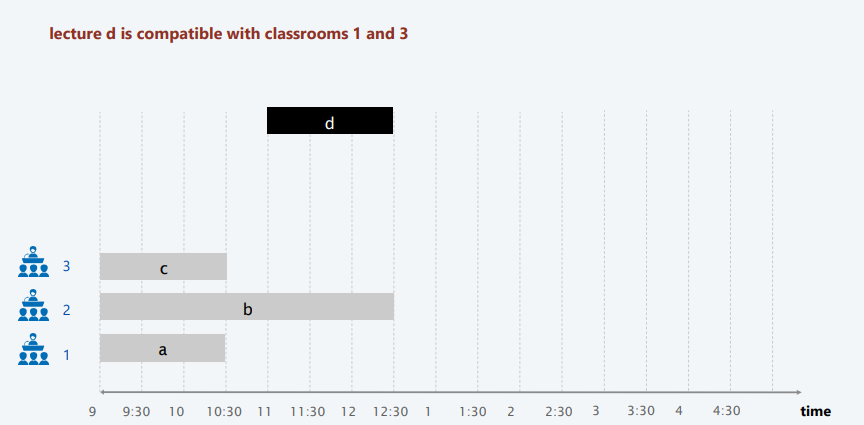
**Altrimenti, aprire una nuova aula.**

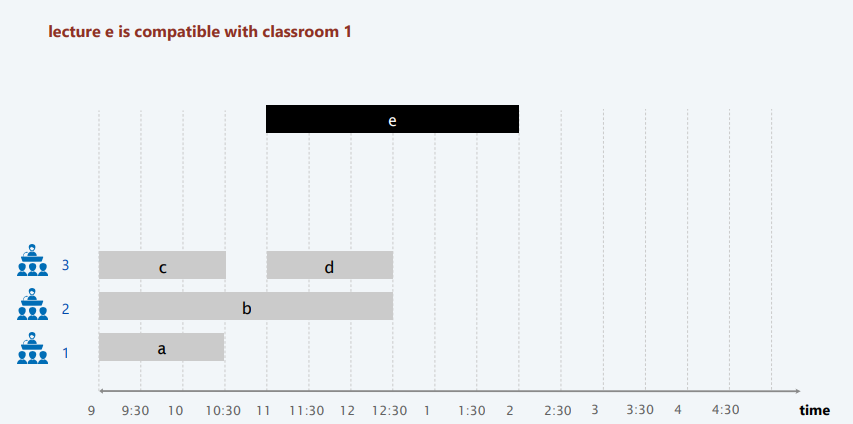
****

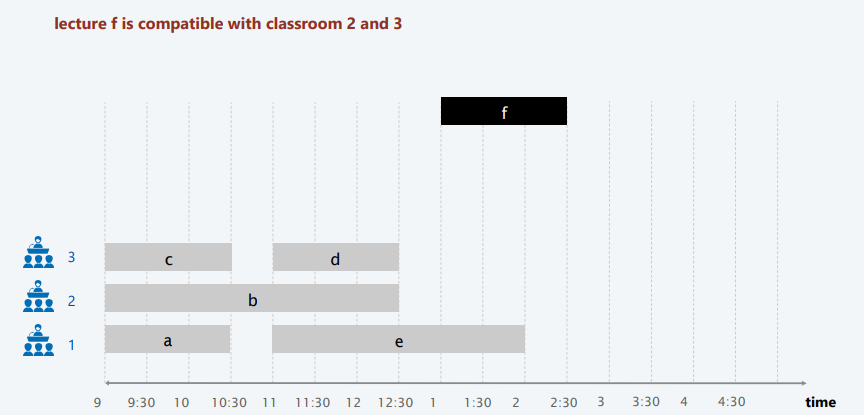
****

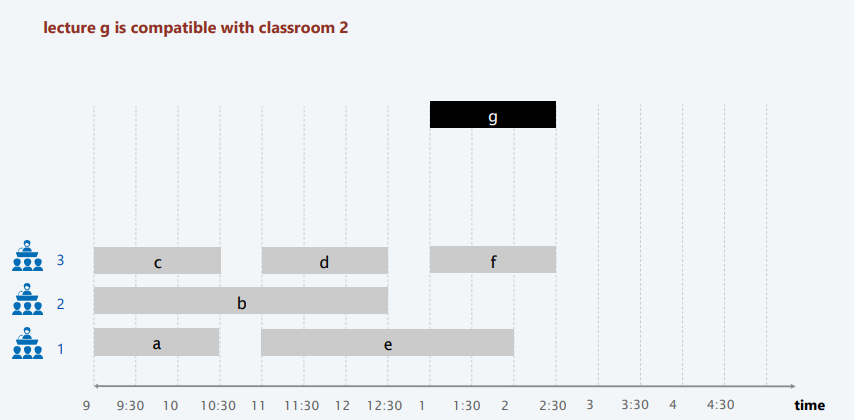


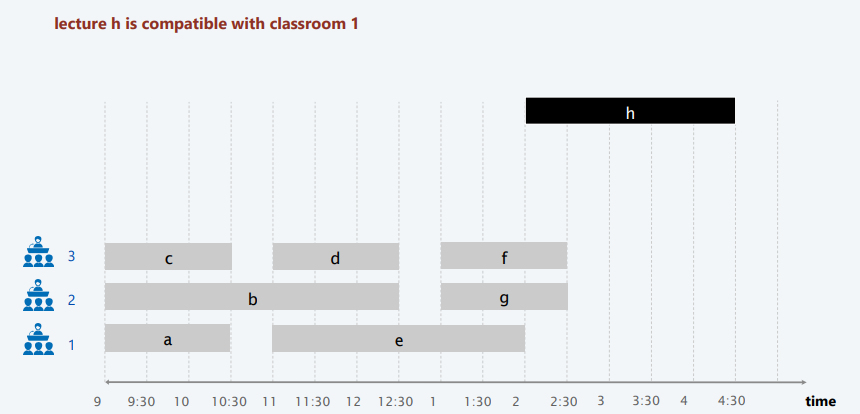


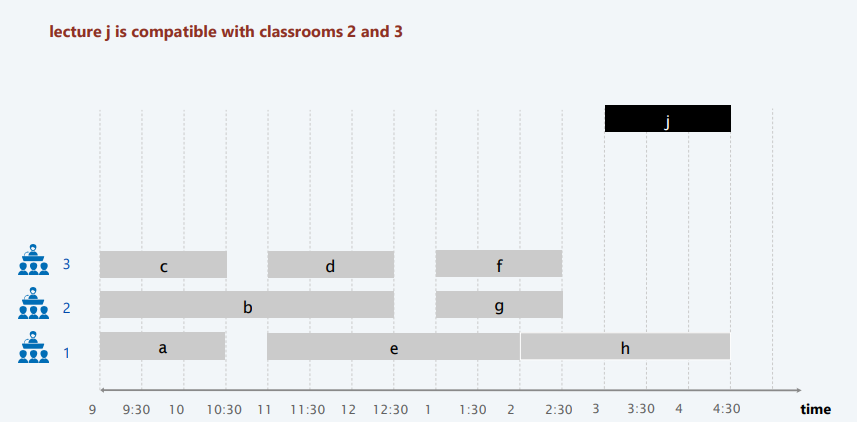


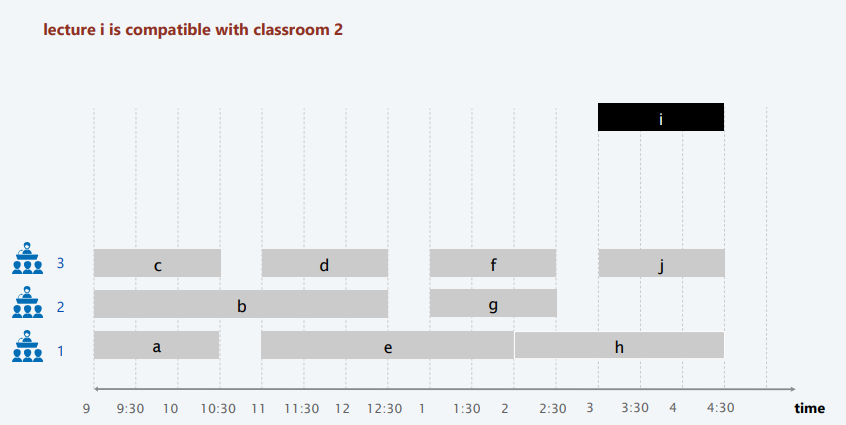


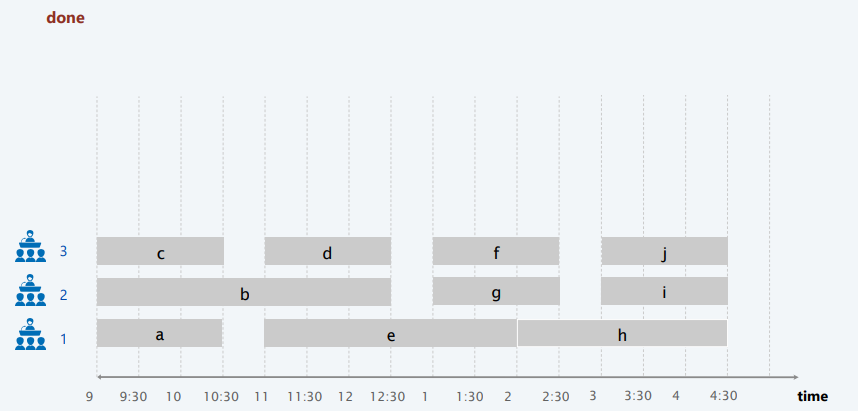












**Proposizione.**

* L'algoritmo earliest-start-time-first può essere implementato in O(n log n) tempo.

**Pf**. L'ordinamento per tempi di inizio richiede un tempo O(n log n).

* Salvaguardare le aule in una coda di priorità (chiave = orario di fine dell'ultima lezione).

- per assegnare una nuova aula, INSERT l'aula nella coda di priorità.

- per programmare la lezione j nell'aula k, aumentare la chiave dell'aula k a fj.

- Per determinare se la lezione j è compatibile con un'aula, confrontare sj con FIND-MIN

* Il numero totale di operazioni sulla coda di priorità è O(n); ciascuna richiede un tempo O(log n). ▪

*Osservazione*. Questa implementazione sceglie un'aula k il cui tempo di fine

L'ultima lezione è il più prossimo.

**Interval partitioning: lower bound on optimal solution**

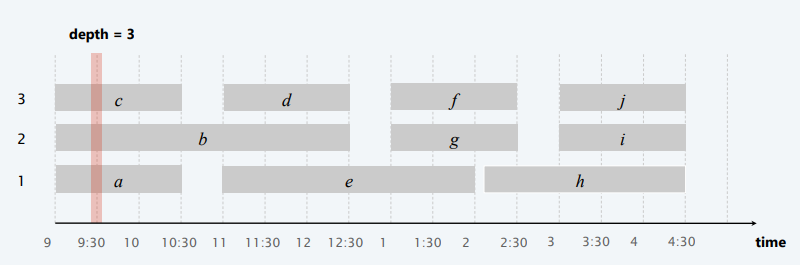
***Definizione***. La profondità di un insieme di intervalli aperti è il numero massimo di intervalli che contengono un determinato punto.

**Osservazione chiave**. Numero di intervalli necessari ≥ profondità.

Q. Il numero minimo di aule necessarie è sempre uguale alla profondità?

A. Sì! Inoltre, l'algoritmo earliest-start-time-first trova una programmazione

il cui numero di aule è uguale alla profondità.



**Osservazione.** L'algoritmo "earliest-start-time first" non programma mai due lezioni incompatibili nella stessa aula.

**Teorema**. L'algoritmo Earliest-start-time-first è ottimale.

**Pf**.

・Lasciare d = numero di aule che l'algoritmo assegna.

L'aula d è stata aperta perché era necessario programmare una lezione, ad esempio j, che è incompatibile con una lezione in ognuna delle altre d - 1 aule.

Pertanto, queste d lezioni terminano ciascuna dopo sj.

・Siccome abbiamo ordinato per orario di inizio, ognuna di queste lezioni incompatibili comincia non più tardi di sj.

・Quindi, abbiamo d lezioni che si sovrappongono all'ora sj + ε.

・**Osservazione chiave** tutti gli orari utilizzano d aule. ▪

## **Il problema Union-Find**

### **Definizione e operazioni:**

Il problema Union-Find si occupa della gestione di una collezione di insiemi disgiunti, ovvero insiemi che non hanno elementi in comune. Le operazioni principali supportate sono:

**1. makeSet(x):**

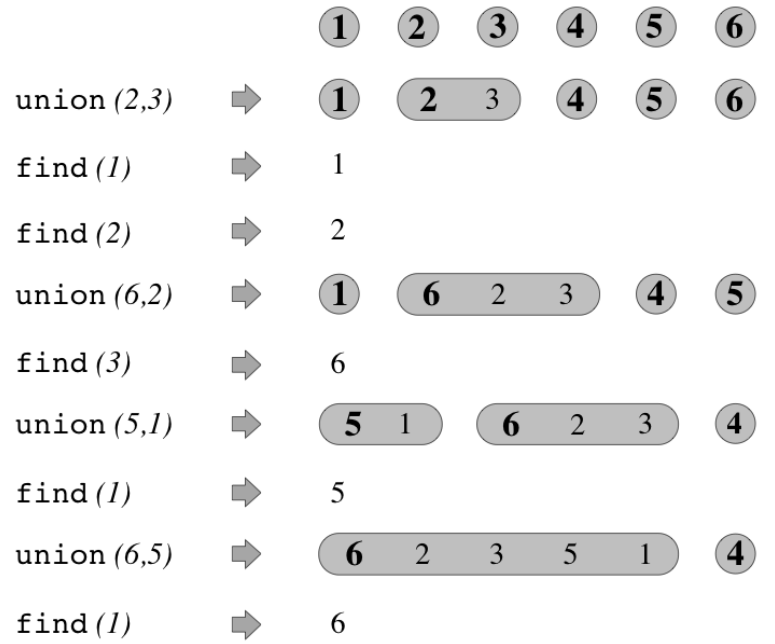
* Crea un nuovo insieme contenente un singolo elemento x.
* L'insieme viene identificato con il nome x.

**2. union(A, B):**

* Unisce gli insiemi A e B in un unico insieme, di nome A.
* I vecchi insiemi A e B vengono distrutti.
* Si presume un accesso diretto agli insiemi A e B.

**3. find(x):**

* Restituisce il nome dell'insieme che contiene l'elemento x.
* Si presume un accesso diretto all'elemento x.



## **Rappresentazione con una foresta di alberi**

### **Idea generale:**

Per rappresentare gli insiemi disgiunti, possiamo utilizzare una **foresta di alberi**. Ogni insieme è rappresentato da un albero radicato, dove:

* La **radice** contiene il nome dell'insieme (elemento rappresentativo).
* Le **foglie** contengono gli elementi dell'insieme, incluso l'elemento rappresentativo che si trova nella radice e ne determina il nome.

### 

### **Approcci elementari (basati su alberi):**

Esistono due strategie principali per implementare il problema Union-Find utilizzando una foresta di alberi:

**1. Alberi QuickFind:**

* In questo approccio, tutti gli alberi hanno **altezza 1**, ovvero sono costituiti da un singolo nodo (la radice).
* Ogni radice rappresenta il nome dell'insieme corrispondente.
* Le foglie (elementi dell'insieme) coincidono con la radice stessa.

**2. Alberi QuickUnion:**

* In questo approccio, gli alberi possono avere qualsiasi altezza.
* La radice di ogni albero rappresenta il nome dell'insieme corrispondente.
* Le foglie non sono necessariamente presenti, ma solo l'elemento rappresentativo si trova nella radice.

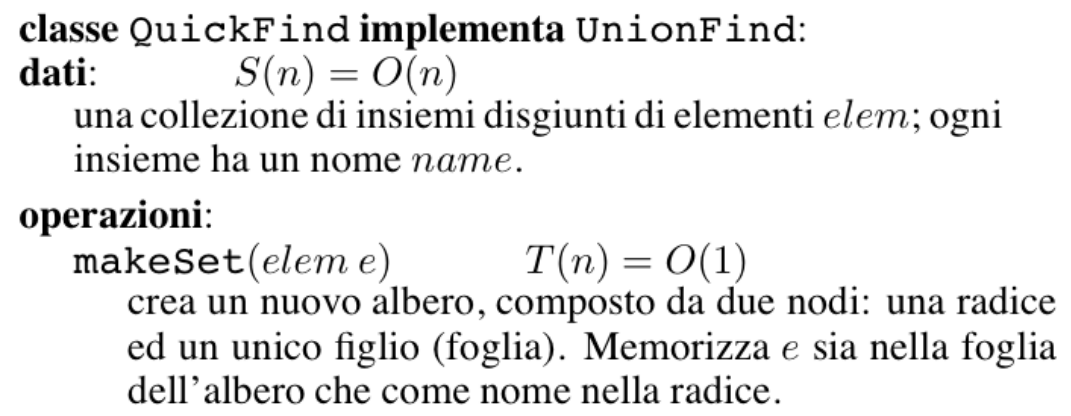
## **Caratteristiche degli alberi QuickFind:**

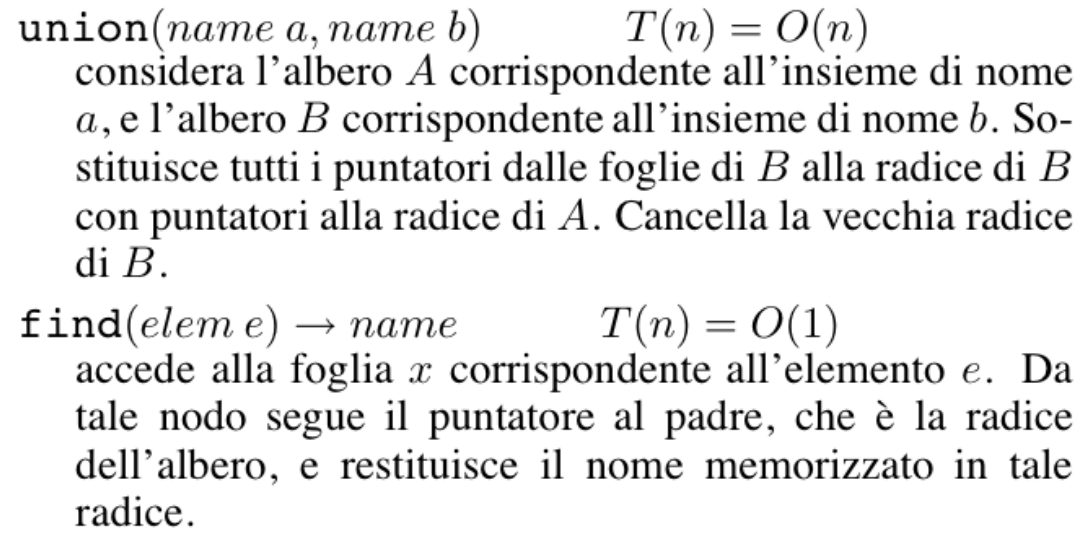
**Vantaggi:**

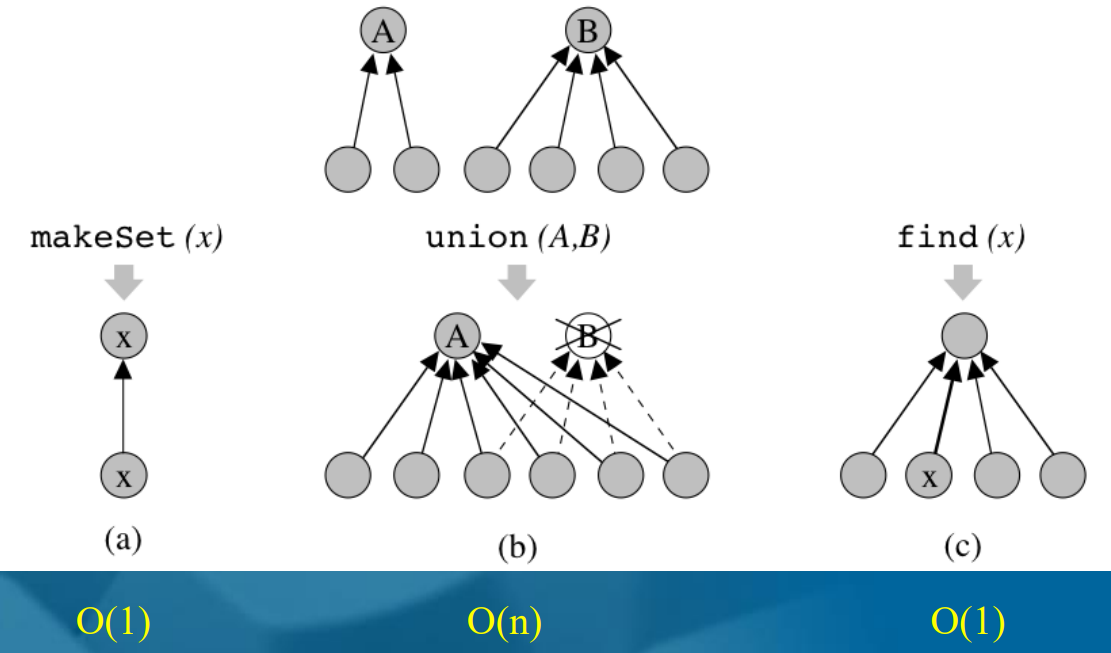
* Semplicità di implementazione.
* Operazione find(x) molto efficiente: basta risalire l'albero fino alla radice.

**Svantaggi:**

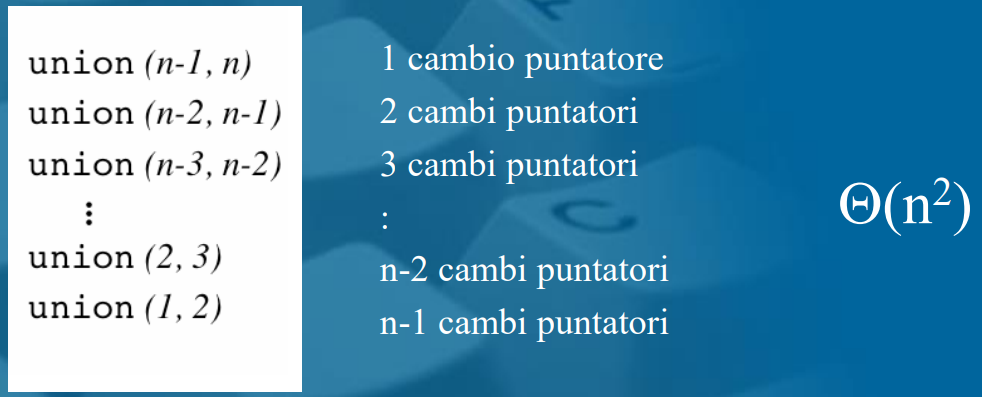
* Operazione union(A, B) meno efficiente: può portare alla creazione di alberi molto alti, con conseguente aumento del tempo di esecuzione.





ESEMPIO:  
 

**Union di costo lineare:  
find e makeSet richiedono solo tempo O(1), ma particolari sequenze di union possono essere molto inefficienti:**

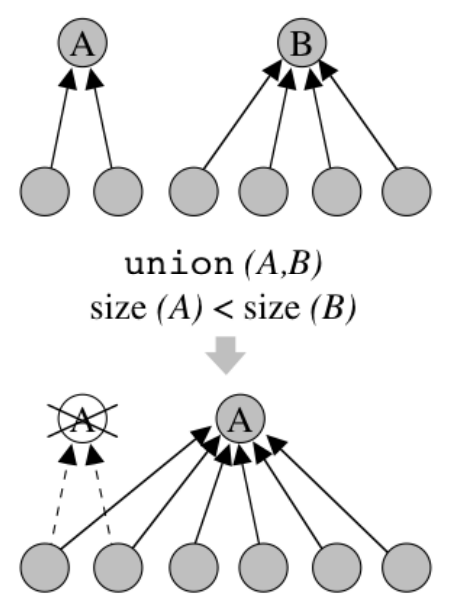
****

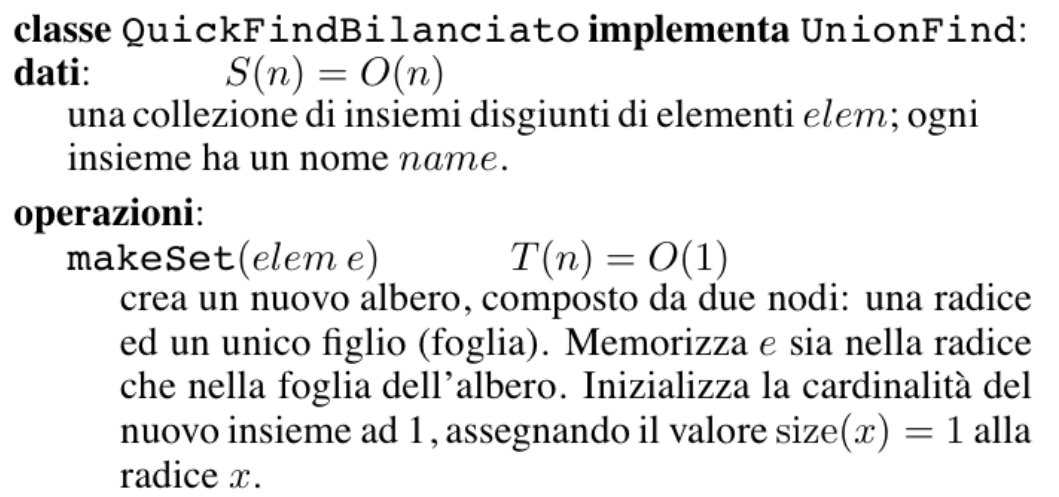
## **Unione per dimensione (Union by Size)**

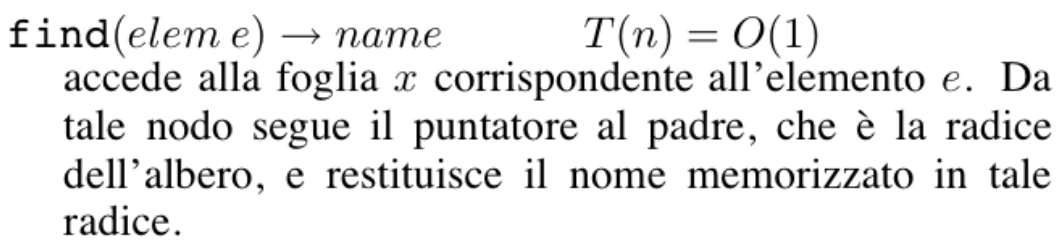
**L'Unione per dimensione è una strategia per implementare l'operazione di unione (union(A, B)) nel problema Union-Find che si basa sulla dimensione (numero di elementi) degli insiemi da unire.**

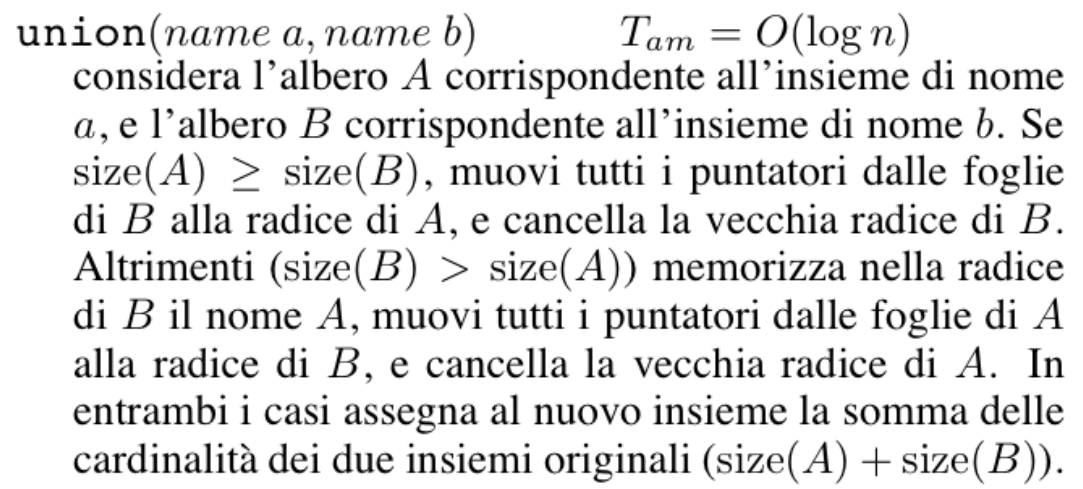
### **Funzionamento:**

1. **Confronto delle dimensioni:**
   * **Si confrontano le dimensioni (size) degli insiemi A e B.**
2. **Attacco dell'insieme più piccolo:**
   * **L'insieme con la dimensione più piccola viene attaccato come sottoalbero all'insieme con la dimensione maggiore.**
3. **Aggiornamento della radice (se necessario):**
   * **Se la radice dell'insieme a cui viene attaccato l'altro ha una dimensione maggiore della nuova radice, la radice dell'albero combinato diventa la radice con la dimensione maggiore.**
4. **Aggiornamento della dimensione:**
   * **La dimensione dell'albero combinato viene aggiornata sommando le dimensioni dei due insiemi originali.**

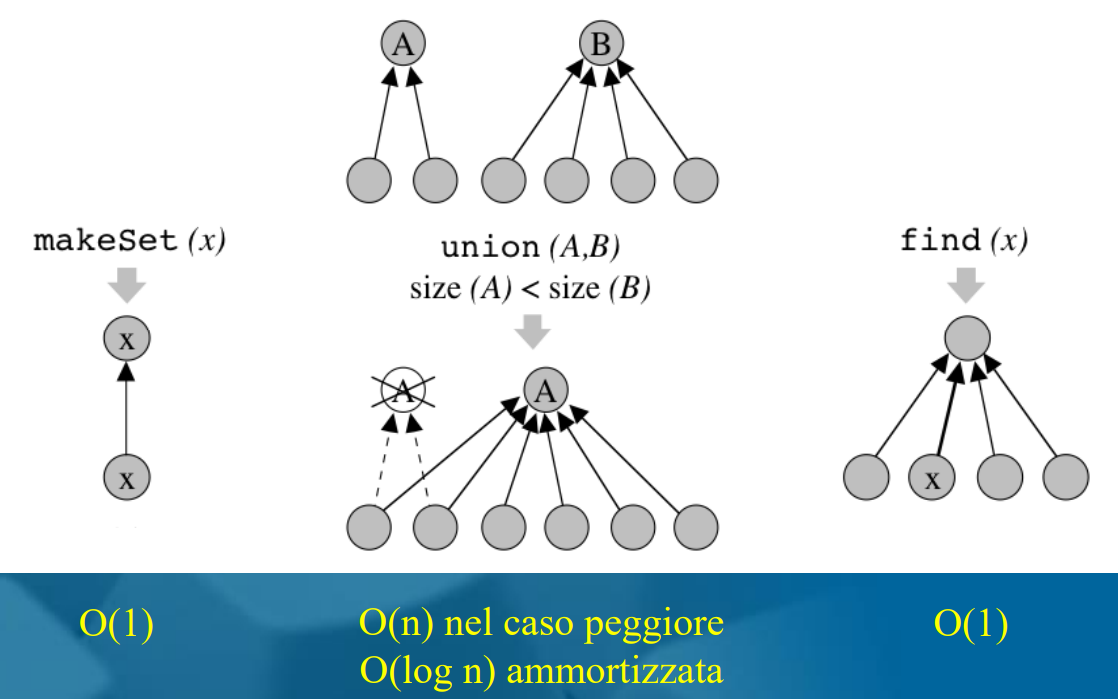
****

****

****

****

**Tam = tempo per operazione ammortizzato sull’intera sequenza di unioni vedremo che una singola union può costare (n), ma l’intera sequenza di n-1 union costa O(n log n)**

****

## **Dimostrazione del tempo di esecuzione per** mfind**,** nmakeSet **e** n-1union

### **Idea generale:**

**L'obiettivo è dimostrare che il tempo totale per eseguire una sequenza di m operazioni find, n makeSet e al massimo n-1 union è in O(m + n log n).**

### **Analisi delle operazioni:**

**1.** find **e** makeSet**:**

* **È facile dimostrare che entrambe le operazioni richiedono tempo Θ(m + n).**
* **L'operazione find richiede tempo proporzionale all'altezza dell'albero in cui si trova l'elemento da cercare, che è al massimo n (altezza massima di un albero con n nodi).**
* **L'operazione makeSet richiede tempo costante per creare un nuovo albero con un singolo nodo.**

**2.** union**:**

* **Per analizzare le operazioni di union, ci concentriamo su un singolo nodo/elemento.**
* **Dimostriamo che il tempo speso per tale nodo è O(log n), il che implica che il tempo totale speso per tutte le union è O(n log n).**

### **Analisi del tempo per un singolo nodo in** union**:**

* **Quando eseguiamo un'operazione union, per ogni nodo che cambia padre pagheremo un tempo costante.**
* **Tuttavia, ogni nodo può cambiare al massimo O(log n) padri.**

**Motivazione:**

* **Ogni volta che un nodo cambia padre, la cardinalità dell'insieme a cui apparterrà è almeno doppia rispetto a quella dell'insieme a cui apparteneva!**
* **All'inizio, un nodo è in un insieme di dimensione 1.**
* **Se cambia padre, si troverà in un insieme di dimensione almeno 2.**
* **All'i-esimo cambio, si troverà in un insieme di dimensione almeno 2^i.**
* **Quindi, il numero di cambi di padre per un nodo è limitato a log\_2(n), che è equivalente a log n.**

**Conclusione:**

* **Il tempo speso per un singolo nodo sull'intera sequenza di n union è O(log n).**
* **Di conseguenza, il tempo totale per tutte le union è O(n log n).**

**Tempo totale:**

* **Considerando il tempo per find, makeSet e union, il tempo totale per l'intera sequenza di operazioni è:**

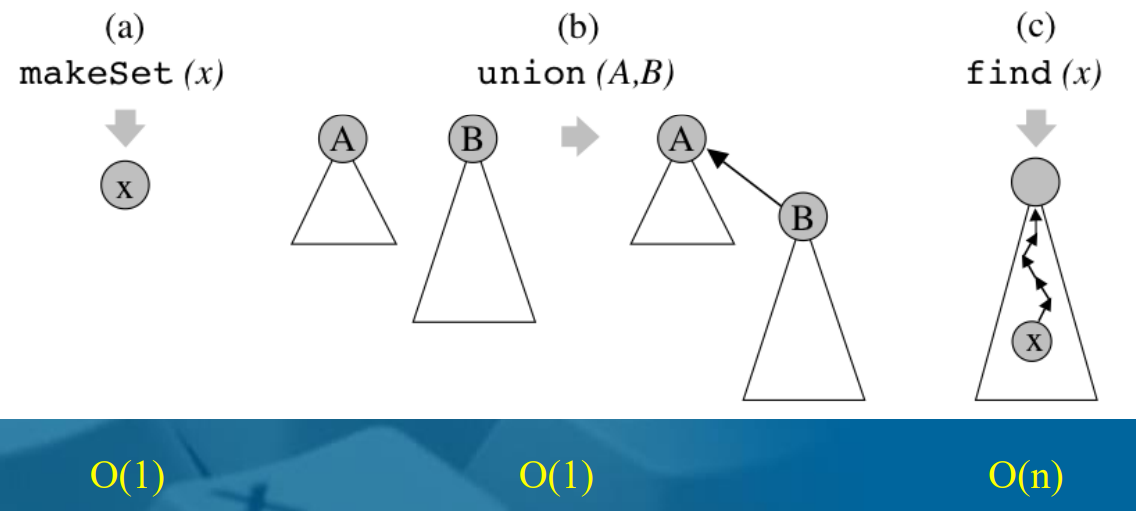
**O(m + n) + O(n log n) = O(m + n log n)**

## **Alberi QuickUnion**

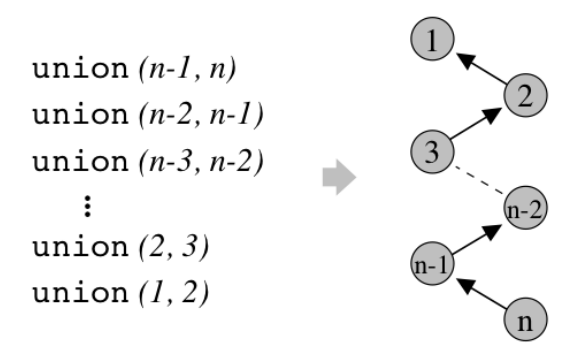
### **Idea generale:**

**Gli alberi QuickUnion rappresentano gli insiemi disgiunti utilizzando una foresta di alberi, dove:**

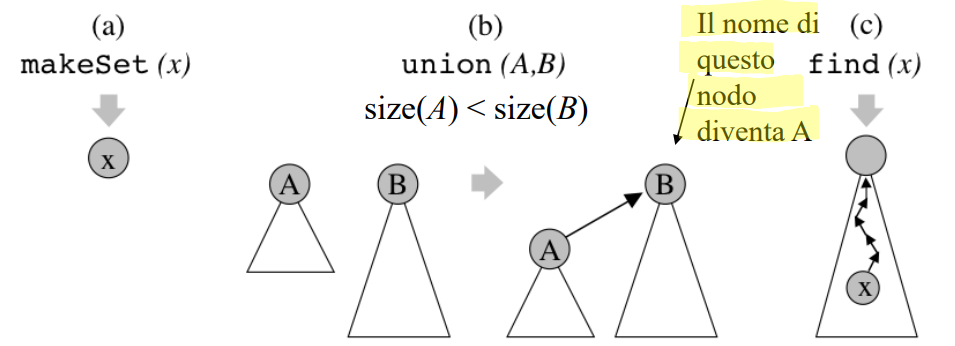
* **Ogni albero può avere un'altezza qualsiasi, non solo 1 come negli alberi QuickFind.**
* **La radice di ogni albero contiene l'elemento rappresentativo dell'insieme.**
* **I nodi rimanenti (escluso la radice) contengono gli altri elementi dell'insieme.**

****

**Find di costo lineare:  
  
particolari sequenze di union possono generare un albero di altezza lineare, e quindi la find è molto inefficiente (costa n-1 nel caso peggiore)**

****

**—> Se eseguiamo n makeSet, n-1 union come sopra, seguite da m find, il tempo richiesto dall’intera sequenza di operazioni è O(n+n-1+mn)=O(mn).**

**Bilanciamento in alberi QuickUnion Union by size:   
nell’unione degli insiemi A e B, rendiamo la radice dell’albero con meno nodi figlia della radice dell’albero con più nodi  
**

## **Lemma: Dimostrazione**

### **Enunciato:**

**Con l'utilizzo di "union by size", dato un albero QuickUnion con size (numero di nodi) s e altezza h, vale che s >= 2^h.**

### **Dimostrazione:**

**1. Caso base:**

* **Per un albero con un singolo nodo (h = 0), s = 1 e 2^h = 2^0 = 1. La disuguaglianza è verificata.**

**2. Induzione matematica:**

* **Assumiamo che la disuguaglianza sia vera per un albero con altezza h-1, ovvero s >= 2^(h-1).**
* **Consideriamo un albero con altezza h.**

**Caso 2.1: Foglia:**

* **Se l'albero è una foglia (nessun sottoalbero), allora s = 1 e h = 1. La disuguaglianza s >= 2^h = 2^1 = 2 è verificata.**

**Caso 2.2: Albero con sottoalberi:**

* **Se l'albero ha uno o più sottoalberi, sia s\_1 e s\_2 le size dei due sottoalberi con la massima size.**
* **Per ipotesi induttiva, s\_1 >= 2^(h-1) e s\_2 >= 2^(h-1).**
* **Poiché l'albero viene creato con "union by size", la size dell'albero risultante è s = s\_1 + s\_2.**
* **Applicando la proprietà distributiva della disuguaglianza:**

**s = s\_1 + s\_2 >= 2^(h-1) + 2^(h-1) = 2 \* 2^(h-1) = 2^h**

* **La disuguaglianza s >= 2^h è verificata anche per l'albero con altezza h.**

**Conclusione:**

**Per induzione matematica, abbiamo dimostrato che per un qualsiasi albero QuickUnion con "union by size", la size è sempre maggiore o uguale a 2^h.**

## **Tempo di esecuzione**

### **Operazione** find**:**

**L'operazione find in un albero QuickUnion con "union by size" richiede tempo O(log n), dove n è il numero di elementi.**

* **Per trovare l'elemento rappresentativo, si risale l'albero fino alla radice.**
* **La "union by size" garantisce che l'altezza degli alberi sia logaritmica in n, portando ad un tempo di esecuzione di O(log n).**

### **Tempo totale:**

**Considerando una sequenza di n makeSet, n-1 union e m find, il tempo totale di esecuzione con "union by size" e compressione del percorso è:**

**O(n) + O(n-1) + O(m log n) = O(n + m log n)**

* **Il tempo per n makeSet è O(n).**
* **Il tempo per n-1 union è O(n-1), assumendo un'altezza media costante degli alberi.**
* **Il tempo per m find è O(m log n), con un costo di log n per ogni operazione.**

## 

## 

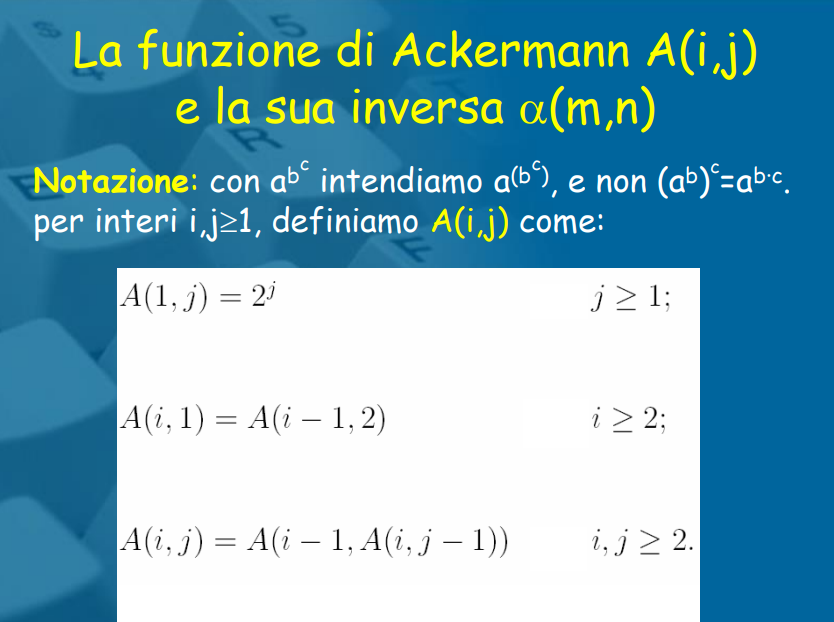
## **Teorema di Tarjan e van Leeuwen**

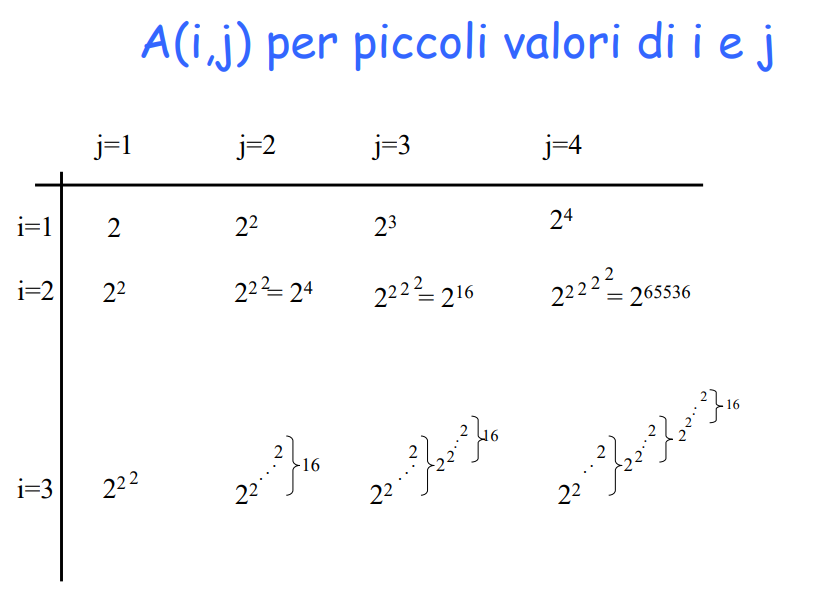
**Il teorema di Tarjan e van Leeuwen stabilisce che utilizzando in QuickUnion le euristiche di "union by rank" (o "union by size") e compressione del percorso, una qualsiasi sequenza di n makeSet, n-1 union e m find ha un costo di:**

**O(n + m alpha(n+m,n))**

**dove alpha(x,y) è la funzione inversa della funzione di Ackermann.**

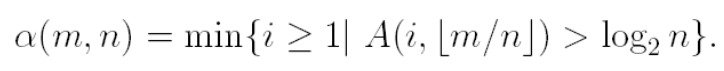
* **La funzione di Ackermann è una funzione a due variabili che cresce molto lentamente.**
* **La funzione inversa 𝛼(x,y) è ancora più lenta della funzione di Ackermann.**

****

****

**In pratica, questo significa che il costo di esecuzione dell'algoritmo è quasi lineare (O(n)), con un piccolo fattore aggiuntivo che dipende dalla funzione di Ackermann.**

**La funzione 𝛼(m,n)   
Per interi m>=n>=0, definiamo (m,n) come:**

****

****

## **Minimum Spanning Tree (MST)**

### **Definizione:**

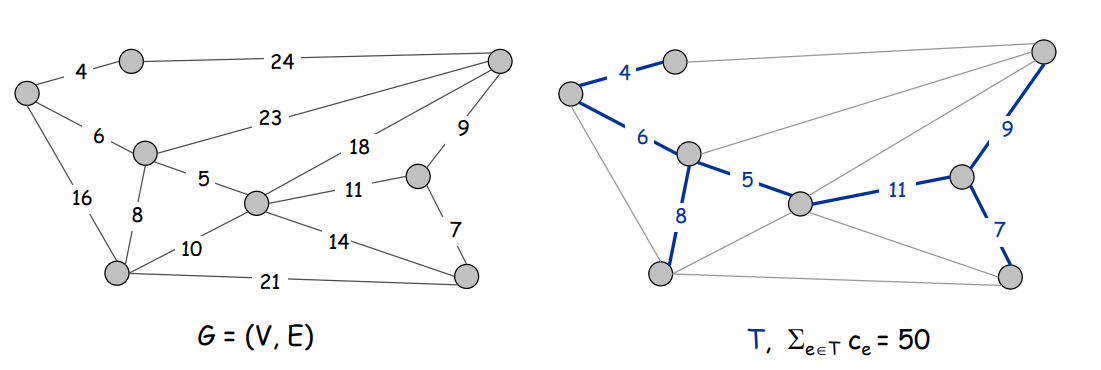
**Un Minimum Spanning Tree (MST), o albero minimo ricoprente, di un grafo connesso e pesato G = (V, E) è un sottoinsieme T degli spigoli E che soddisfa le seguenti condizioni:**

1. **Spanning Tree: T è un albero ricoprente, ovvero collega tutti i vertici di V esattamente una volta, formando un ciclo privo.**
2. **Costo minimo: La somma dei pesi degli spigoli in T è minima rispetto a qualsiasi altro albero ricoprente del grafo.**

### **Teorema di Cayley**

**Il Teorema di Cayley fornisce il numero di alberi ricoprenti completi di un grafo completo K\_n, dove n è il numero di vertici.**

**Il teorema afferma che esistono n^(n-2) alberi ricoprenti completi di K\_n.**

****

## **Problema dell'Albero Minimum Spanning Tree (MST)**

### **Definizione formale:**

**Il problema dell'Albero Minimum Spanning Tree (MST) può essere formalizzato come segue:**

**Ingresso:**

* **Un grafo connesso e non orientato G = (V, E).**
  + **V: Insieme di vertici del grafo.**
  + **E: Insieme di spigoli che collegano i vertici in V.**
* **Ogni spigolo e ∈ E ha un peso a valore reale c\_e che rappresenta il suo costo.**

**Soluzione fattibile:**

* **Un albero ricoprente T del grafo G.**
  + **Un albero ricoprente è un sottoinsieme di spigoli F ⊆ E che soddisfa due condizioni:**
    1. **Connettività: T = (V, F) collega esattamente una volta tutti i vertici in V. Non ci sono cicli nell'albero.**
    2. **Fattibilità: Tutti gli spigoli in F appartengono al grafo originale E (cioè, F ⊆ E).**

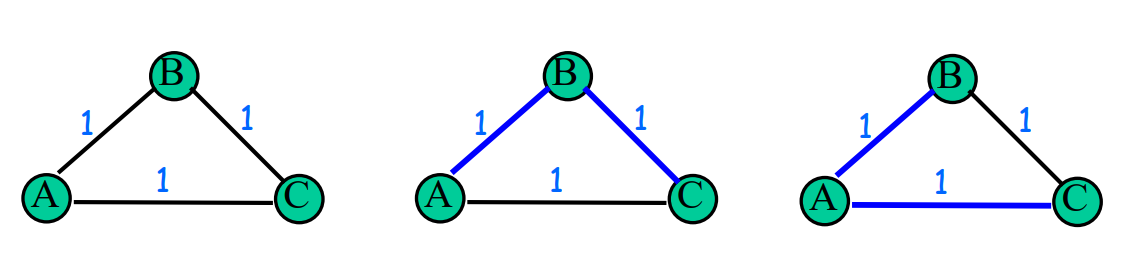
**Misura da minimizzare:**

* **Il costo totale dell'albero ricoprente T.**
* **Viene calcolato come la somma dei pesi di tutti gli spigoli in T:**

**c(T) = Σ(e ∈ T) c\_e**

## **Unicità dell'Albero Minimum Spanning Tree (MST)**

**L'MST non è sempre unico in generale. Tuttavia, esiste una proprietà importante:**

****

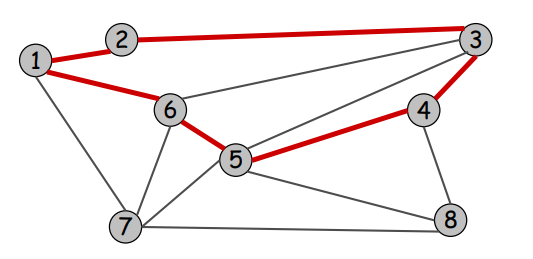
**Proprietà: Se tutti i pesi degli spigoli in G sono distinti (nessun peso è uguale a un altro), allora l'MST è unico.**

**Cycles and Cuts**

**Cycles:**

**Un ciclo in un grafo è un insieme di spigoli che formano un percorso chiuso. Si inizia da un vertice a, si passa a un vertice b, poi a c, e così via, fino a tornare al vertice iniziale a. In altre parole, il percorso si "chiude" su se stesso.**

**Esempio di ciclo: a-b, b-c, c-d, ..., y-z, z-a**

****

### **Cut :**

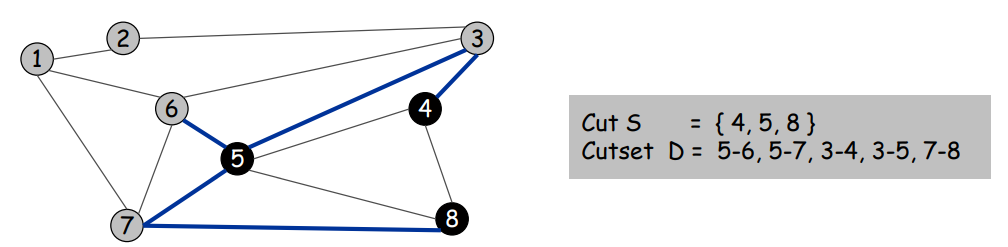
**Un taglio (cut) in un grafo è un sottoinsieme di vertici S. A volte viene definito come una partizione dell'insieme dei vertici V in due sottoinsiemi: S e V\S (vertici in S e vertici *non* in S).**

**Esempio di taglio: S = {v1, v3, v5}**

### **Cutset :**

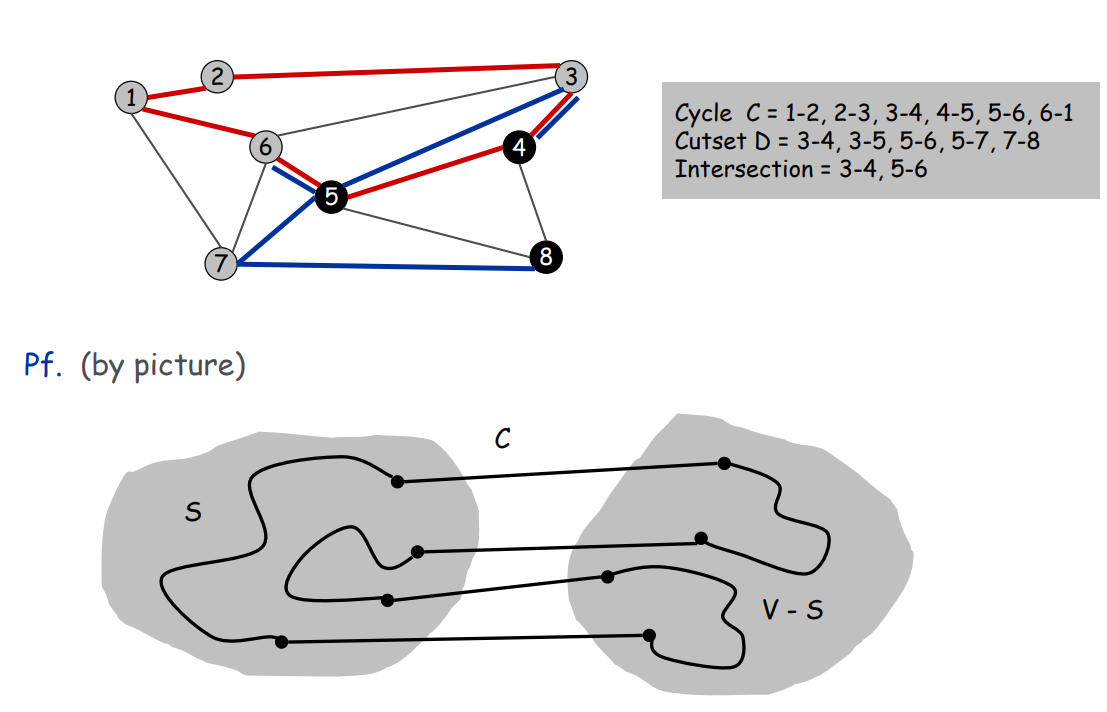
**L'insieme di taglio (cutset) corrispondente a un taglio S è il sottoinsieme di spigoli che ha esattamente un endpoint in S. In altre parole, gli spigoli dell'insieme di taglio collegano vertici in S con vertici *non* in S.**

**Esempio di insieme di taglio: L'insieme di taglio per S = {v1, v3, v5} conterrebbe tutti gli spigoli che collegano v1, v3, o v5 con qualsiasi altro vertice nel grafo.**

****

### **Intersezione Cycle-Cut**

**Affermazione: Un ciclo e un insieme di taglio si intersecano in un numero pari di spigoli.**

****

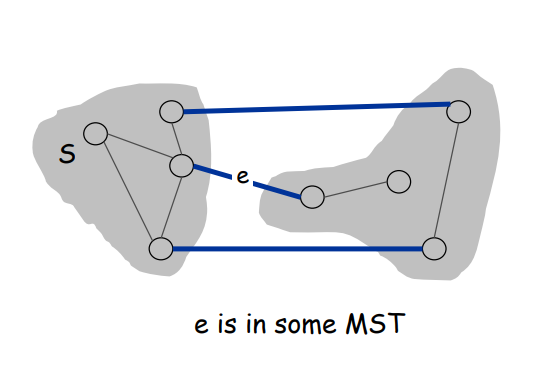
## **Algoritmi Greedy per MST**

**Gli algoritmi greedy per trovare l'MST (Minimum Spanning Tree) si basano su due proprietà fondamentali: la proprietà di taglio e la proprietà di ciclo.**

### **Proprietà di Taglio (Cut Property)**

**La proprietà di taglio afferma:**

**Dato un sottoinsieme di vertici** S **e uno spigolo** e **con il costo minimo tra tutti gli spigoli che collegano un vertice in** S **a un vertice *non* in** S **(spigolo attraversante il taglio), allora esiste un MST che contiene lo spigolo** e**.**

****

### **Dimostrazione della proprietà di taglio:**

1. **Ipotesi: Supponiamo di avere un MST T che *non* contiene lo spigolo e con il costo minimo che attraversa il taglio S.**
2. **Costruzione di un nuovo albero: Consideriamo l'albero T' ottenuto aggiungendo lo spigolo e a T.**
3. **Ciclo in T': Poiché T era un albero ricoprente e e non era presente, l'aggiunta di e deve creare un ciclo in T'.**
4. **Sostituzione con un costo maggiore: In questo ciclo, c'è almeno uno spigolo con un costo maggiore di e.**
5. **Sostituzione per costo minore: Possiamo rimuovere quello spigolo dal ciclo e mantenere la connessione sostituendolo con e.**
6. **Nuovo albero con costo inferiore: Il nuovo albero risultante, ottenuto da T con e al posto dello spigolo rimosso dal ciclo, è ancora un albero ricoprente e ha un costo totale inferiore a T.**
7. **Contraddizione: Ciò contraddice l'ipotesi iniziale che T fosse un MST.**

**Pertanto, l'MST deve contenere lo spigolo e con il costo minimo che attraversa il taglio.**

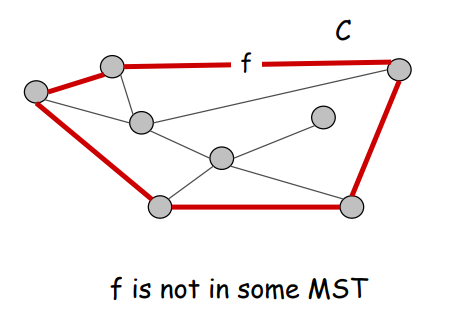
### 

### **Proprietà di Ciclo (Cycle Property)**

**La proprietà di ciclo afferma:**

**Dato un ciclo** C **e lo spigolo** f **con il costo massimo all'interno del ciclo, allora esiste un MST che *non* contiene lo spigolo** f**.**

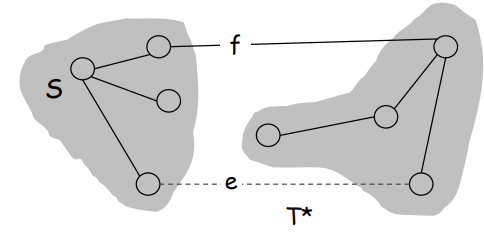
**In altre parole, se abbiamo un ciclo e lo spigolo più costoso al suo interno, possiamo rimuoverlo e comunque trovare un MST valido.**

****

### **Dimostrazione della proprietà di ciclo:**

1. **Ipotesi: Supponiamo che ogni MST debba contenere lo spigolo f con il costo massimo in un ciclo C.**
2. **Ridondanza in qualsiasi MST: Ciò significherebbe che *tutti* gli MST conterrebbero lo spigolo f.**
3. **Taglio attraverso il ciclo: Possiamo definire un taglio S che separa i vertici del ciclo C in due insiemi.**
4. **Spigolo con costo minimo attraverso il taglio: Poiché f è lo spigolo con il costo massimo nel ciclo, ci sarà sempre un altro spigolo nel ciclo con un costo inferiore a f che attraversa il taglio S.**
5. **Applicazione della proprietà di taglio: La proprietà di taglio ci dice che un MST deve contenere lo spigolo con il costo minimo che attraversa il taglio S.**
6. **Contraddizione: Ciò contraddice l'ipotesi iniziale che tutti gli MST debbano contenere f.**

**Pertanto, possiamo rimuovere lo spigolo f dal ciclo e comunque trovare un MST valido.**

****

**Kruskal's algorithm**

**Iniziare con T = ɸ . Considerare i bordi in ordine crescente ordine crescente di costo. Inserire l’arco e in T a meno che non si crei un ciclo.**

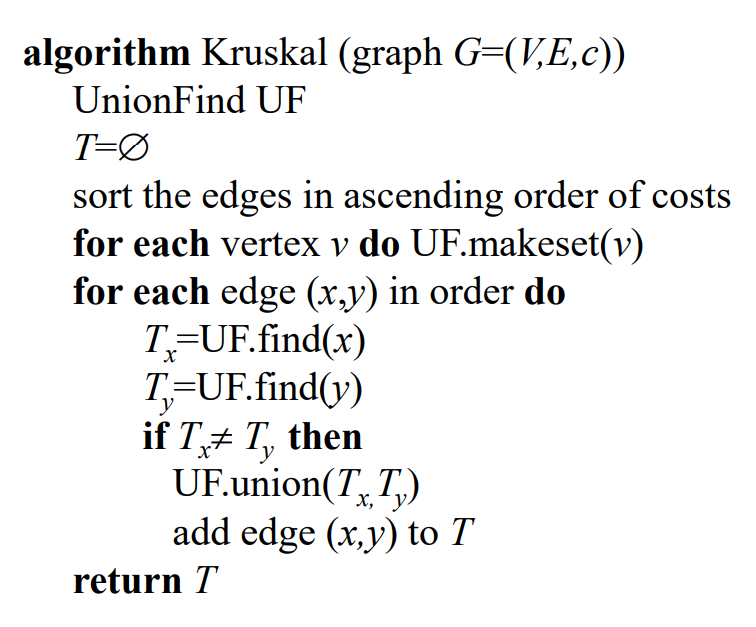
**Osservazione.**

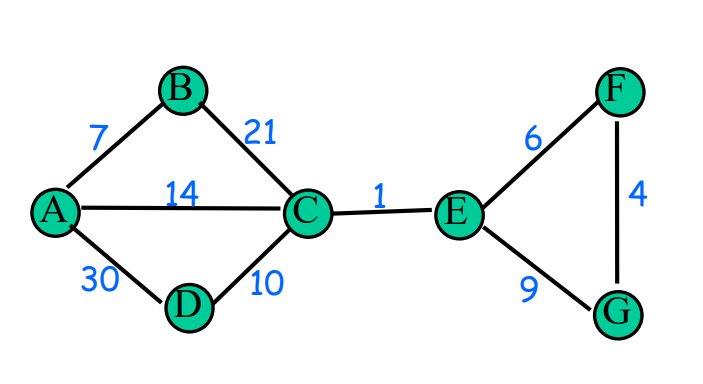
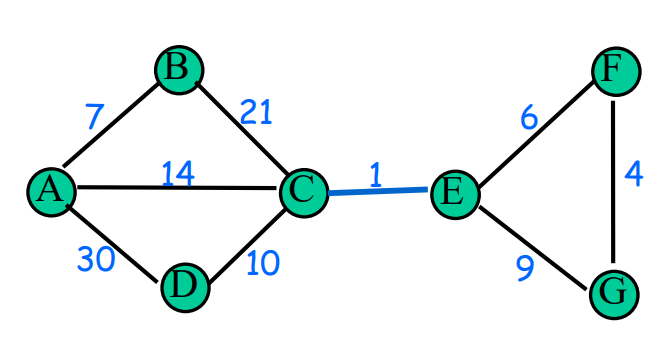
**Un'implementazione efficiente dell'algoritmo di Kruskal utilizza una struttura dati Union-Find**

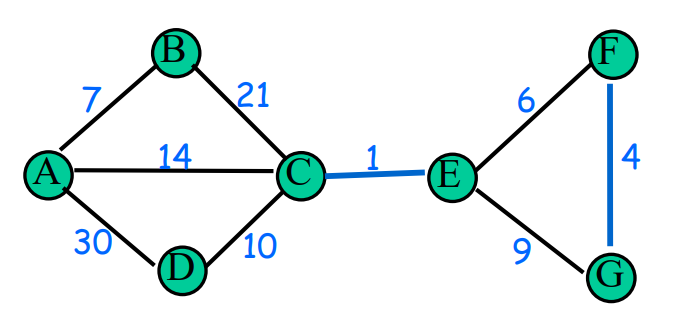
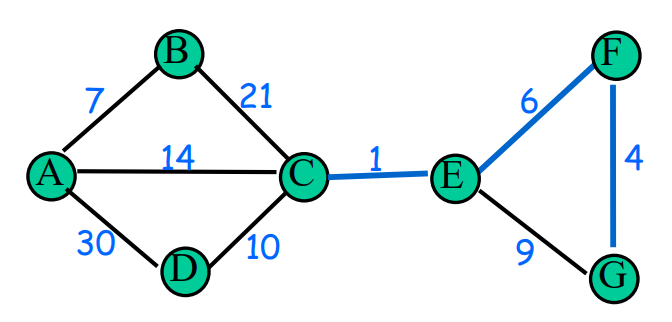
**struttura di dati:**

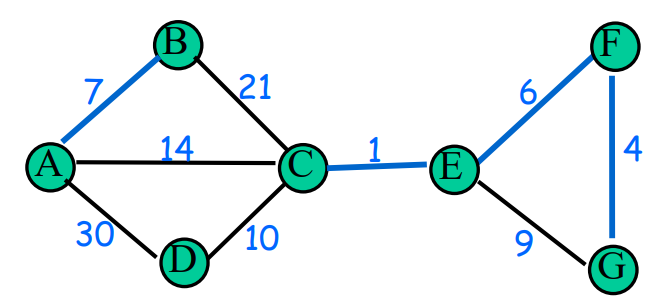
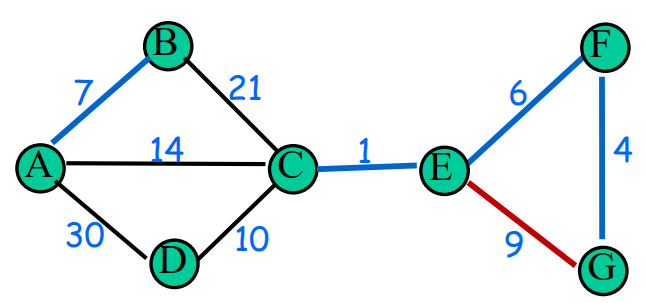
**- per mantenere le componenti connesse delle soluzioni correnti**

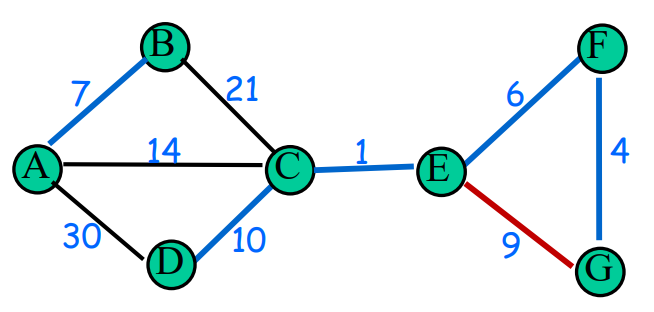
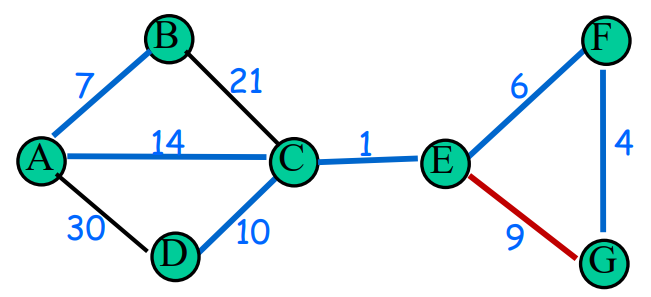
**- per verificare se il bordo corrente forma un ciclo (con la soluzione corrente soluzione)**

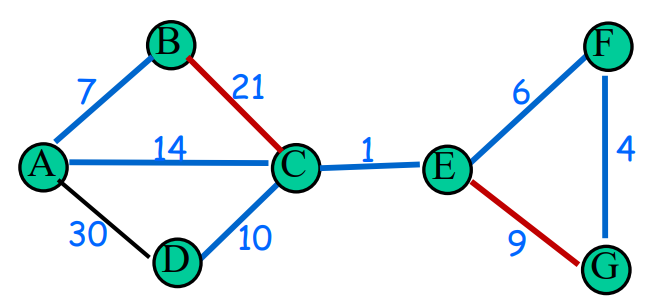
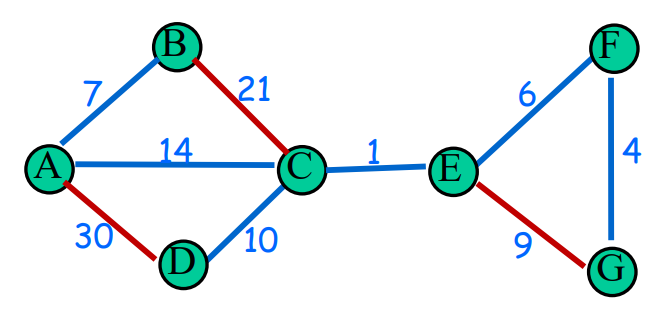
**PSEUDOCODICE:  
**

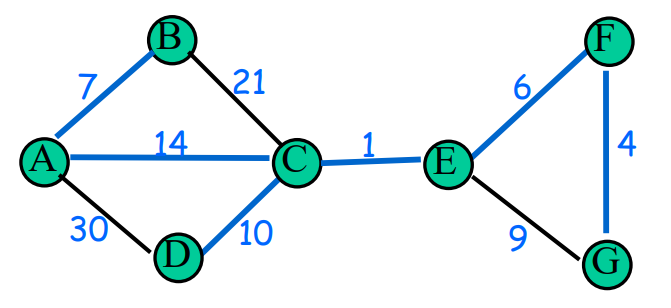
** **

** **

** **

** **

** **

****

## **Correttezza dell'algoritmo di Kruskal**

## **Quando Kruskal aggiunge l'arco (x, y) alla soluzione**

**L'algoritmo di Kruskal aggiunge l'arco (x, y) alla soluzione solo se soddisfa due condizioni:**

1. **Ordine crescente di costo: Gli archi vengono considerati in ordine crescente di costo. Pertanto, (x, y) deve avere un costo minore o uguale a tutti gli archi esaminati in precedenza.**
2. **Nessun ciclo: Aggiungere (x, y) non deve creare un ciclo nell'albero che stiamo costruendo.**

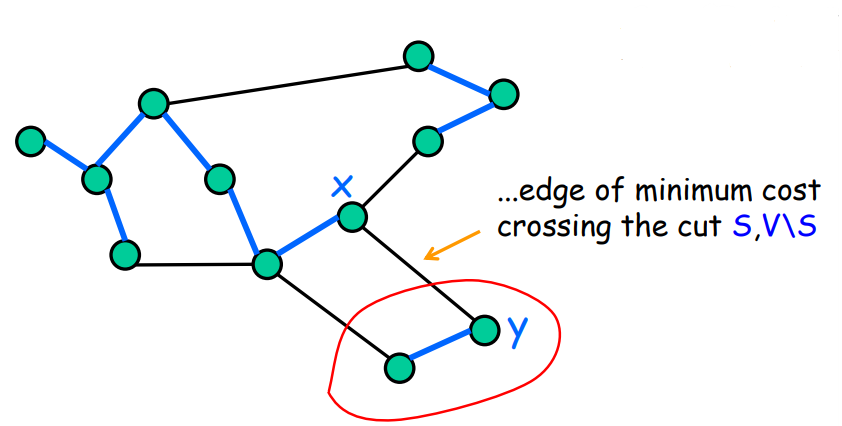
**Per capire meglio la seconda condizione, possiamo analizzarla in base al tuo punto di partenza:**

**Insieme S dei vertici appartenenti alla stessa componente connessa di y:**

**Considera l'insieme S dei vertici che appartengono alla stessa componente connessa di y. In altre parole, S include y e tutti i vertici a cui y è già collegato nell'albero parziale che stiamo costruendo finora.**

**L'algoritmo di Kruskal non aggiungerà mai un arco con entrambi gli endpoint in S. Ciò violerebbe la proprietà di taglio, poiché un MST non deve contenere archi ridondanti che collegano vertici già all'interno della stessa componente connessa.**

**Tuttavia, l'algoritmo considererà l'arco (x, y) se x è *al di fuori* dell'insieme S. In questo caso, (x, y) diventa un candidato per collegare due componenti connesse separate.**

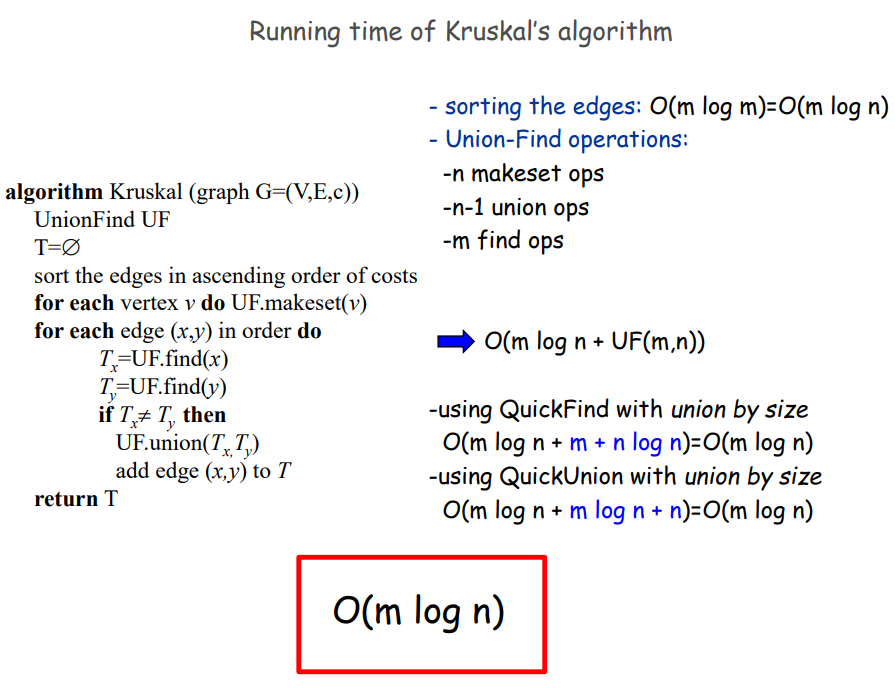
****

**Condizione per aggiungere (x, y):**

**L'arco (x, y) verrà aggiunto all'albero di Kruskal solo se soddisfa entrambe le condizioni:**

* **Il costo di (x, y) è minore o uguale al costo di tutti gli archi esaminati finora.**
* **Aggiungendo (x, y), non si crea alcun ciclo nell'albero parziale. In altre parole, x deve appartenere a una componente connessa diversa da S.**

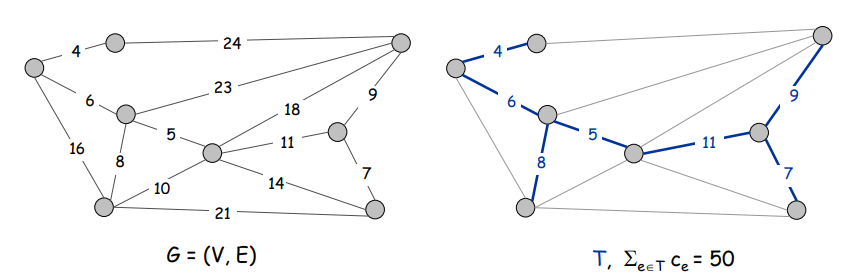
**Se entrambe le condizioni sono vere, (x, y) diventa l'arco di costo minimo che collega due componenti connesse diverse, e verrà aggiunto all'albero di Kruskal per espandere la soluzione MST.**

****

**Prim's algorithm**

**The Minimum Spanning Tree (MST) problem**

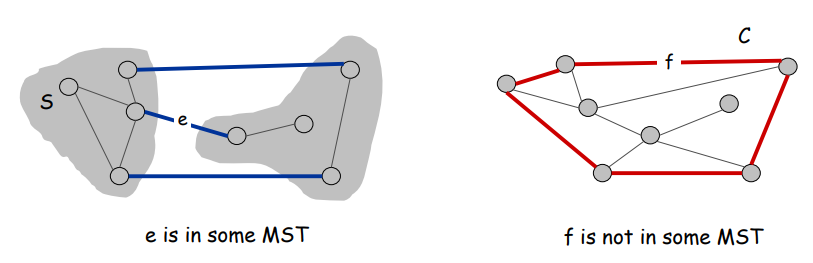
* **Input: Un grafo connesso e non orientato G = (V, E) con pesi a valore reale c\_e per ogni arco e.**
* **Soluzione fattibile: Un albero ricoprente T del grafo G. Un albero ricoprente collega esattamente una volta tutti i vertici in V ed è un sottoinsieme degli archi originali F ⊆ E.**
* **Misura da minimizzare: Il costo totale dell'albero ricoprente T, che è la somma dei pesi di tutti gli archi in T: c(T) = Σ(e ∈ T) c\_e.**

****

**Greedy Algorithms**

**Cut property: Sia S un qualsiasi sottoinsieme di nodi e sia e un bordo di costo minimo con un solo estremo in S. Esiste allora un MST che contiene e.**

**Cycle property: Sia C un ciclo qualsiasi e sia f un bordo di costo massimo appartenente a C. Allora esiste un MST che non contiene f.**

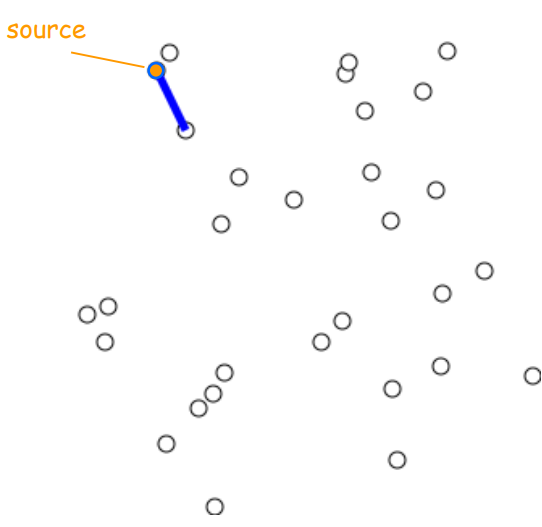
****

**Prim's algorithm [Jarník 1930, Dijkstra 1957, Prim 1959].  
  
Iniziare con un nodo radice s e far crescere avidamente un albero T da s verso l'esterno.**

**A ogni passo, si aggiunge a T il bordo più economico e che ha esattamente un punto finale in T.**

**Correttezza.**

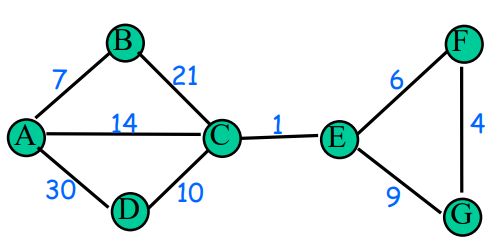
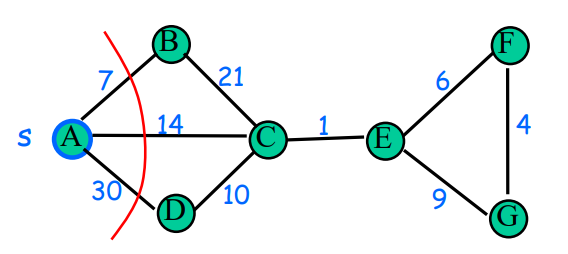
**Conseguenza immediata della proprietà di taglio, utilizzata esattamente n-1 volte.**

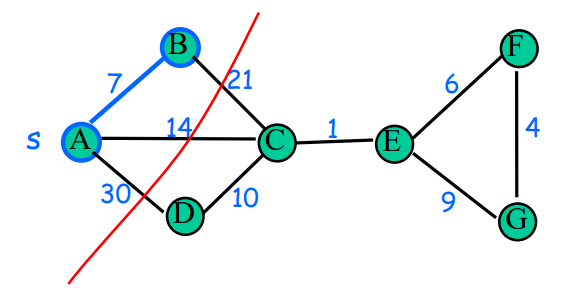
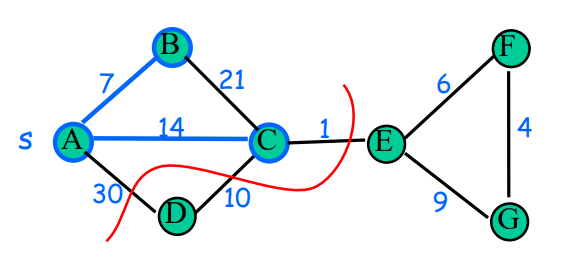


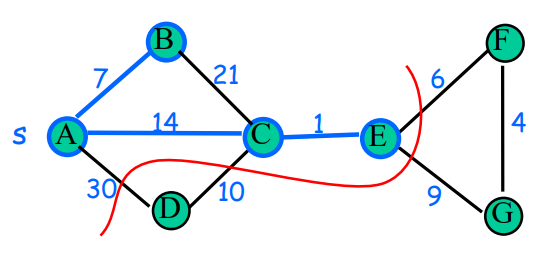
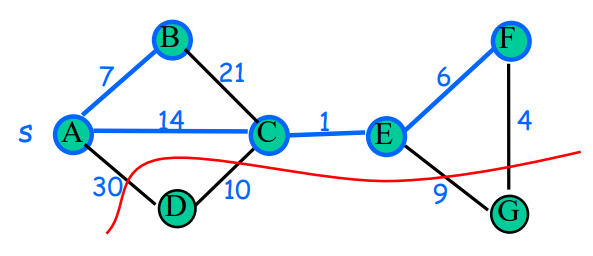
**Grafo completo euclideo:**

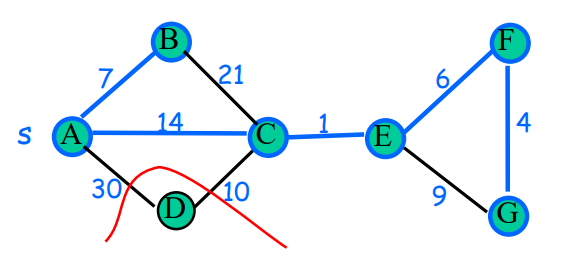
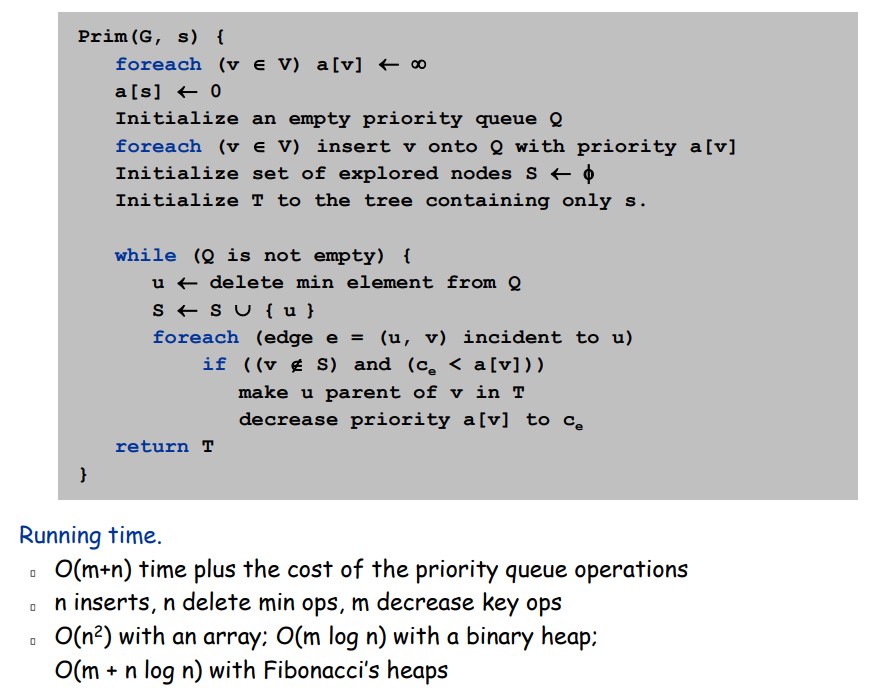
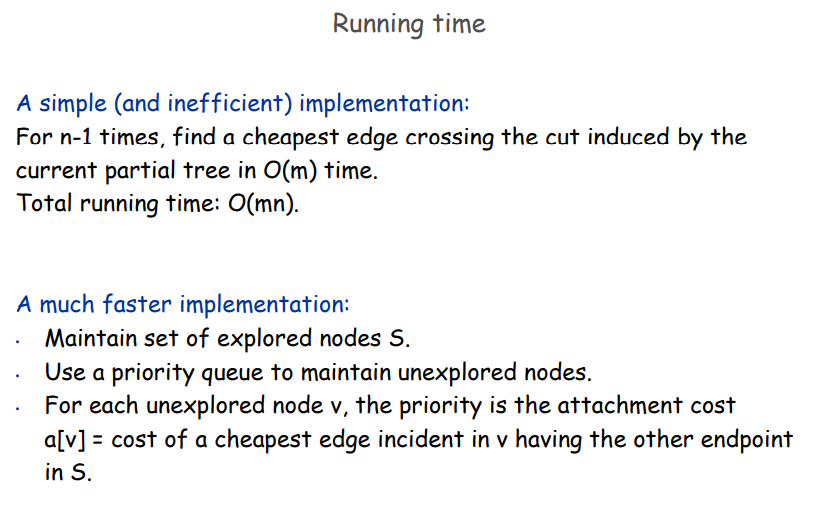
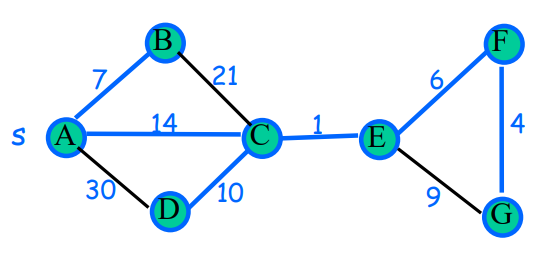
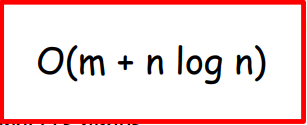
**- vertici disposti sul piano**

**- Per ogni coppia di vertici u e v, il costo del bordo (u,v) è la distanza euclidea tra u e v.**

** **

** **

** **

**  **

**Clustering**

**Clustering (Raggruppamento)**

* **È una tecnica utilizzata per raggruppare un insieme di oggetti (U) in base alle loro similarità.**
* **Ogni oggetto (p1, p2, ..., pn) ha attributi o caratteristiche che lo definiscono.**
* **L'obiettivo è classificare questi oggetti in gruppi (cluster) in modo tale che gli oggetti all'interno di un cluster siano più simili tra loro rispetto agli oggetti in cluster diversi.**

**Funzione di Distanza**

* **Questa funzione gioca un ruolo cruciale nel determinare la similarità.**
* **Assegna un valore numerico che indica quanto siano "vicini" due oggetti nello spazio dei dati.**

**Problema Fondamentale**

* **La sfida principale nel clustering è suddividere i punti dati in cluster in modo tale che:**
  + **I punti all'interno di un cluster siano vicini tra loro secondo la funzione di distanza.**
  + **I punti appartenenti a cluster diversi siano distanti tra loro.**

**k-Clustering**

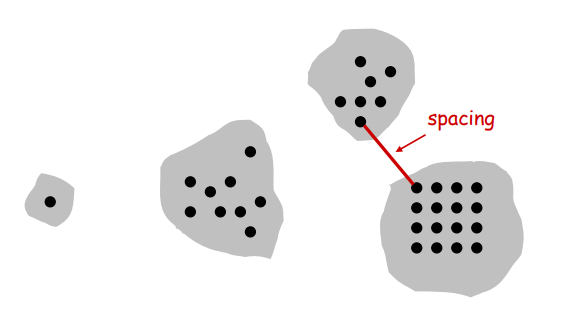
* **Questo è un tipo specifico di clustering in cui si mira a dividere i punti dati (oggetti) in un numero predefinito (k) di gruppi non vuoti (cluster).**

**Funzione di Distanza**

* **Come accennato in precedenza, la funzione di distanza gioca un ruolo cruciale nel determinare quanto siano "vicini" due punti dati.**
* **In questo caso, ipotizziamo che la funzione di distanza rispetti alcune proprietà fondamentali:**
  + **Identità degli Indiscernibili: Se due punti (pi e pj) sono identici, la loro distanza deve essere zero (d(pi, pj) = 0).**
  + **Nonnegatività: La distanza tra due punti qualsiasi non può essere negativa (d(pi, pj) >= 0).**
  + **Simmetria: La distanza tra due punti è la stessa in entrambe le direzioni (d(pi, pj) = d(pj, pi)).**

**Spacing**

* **Nell'ambito del clustering di massima separazione (spacing) , lo spacing si riferisce alla distanza minima tra due punti dati appartenenti a cluster diversi.**

****

## **Algoritmo Greedy per il Clustering (Single-Linkage)**

**Questo algoritmo di clustering si basa sull'idea di trovare gruppi di oggetti "vicini" in modo iterativo e raggruppandoli fino a raggiungere un numero predefinito di cluster (k).**

**Procedimento:**

1. **Costruisci un grafo:**
   * **Considera ogni oggetto (n oggetti in totale) come un vertice nel grafo. Inizialmente, ogni oggetto rappresenta un cluster separato (n cluster totali).**
2. **Trova gli oggetti più vicini in cluster diversi:**
   * **Identifica la coppia di oggetti più vicini tra loro, a condizione che appartengano a cluster diversi.**
   * **Collega questi due oggetti con un arco nel grafo.**
3. **Ripeti finché non ottieni k cluster:**
   * **Ripeti il passaggio 2 (trovare la coppia più vicina in cluster diversi e collegarli) n-k volte.**
   * **Questo processo iterativo raggruppa oggetti vicini, formando via via meno cluster.**
   * **Fermati quando ottieni esattamente k cluster (componenti connesse) nel grafo.**

**Osservazione chiave:**

* **Questa procedura è equivalente all'algoritmo di Kruskal per trovare l'Minimum Spanning Tree (MST) in un grafo, con la differenza che ci si ferma quando si ottengono k componenti connesse (cluster) invece di una singola componente connessa (MST).**

**Clustering Gerarchico:**

* **Eseguire l'algoritmo di Kruskal fino alla fine (cioè fermarsi quando si ottiene una singola componente connessa) produce implicitamente un clustering gerarchico.**
* **Un clustering gerarchico fornisce raggruppamenti per ogni valore di k da n (n cluster singoli) a 1 (tutti gli oggetti in un unico cluster).**

**Analisi dell'Algoritmo Greedy di Clustering**

**Teorema:**

**Sia C\* il clustering C*1, ..., C*k formato eliminando i k-1 archi più costosi di un MST. C\* è un k-clustering con separazione massima.**

**Dimostrazione:**

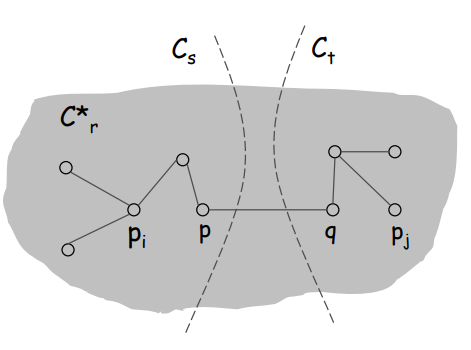
* **Indichiamo con C un altro qualsiasi k-clustering, ovvero C1, ..., Ck.**
* **La separazione di C\* è la lunghezza d\* del (k-1)esimo arco più costoso dell'MST.**
* **Siano pi e pj due oggetti appartenenti allo stesso cluster in C\*, diciamo C*r, ma a cluster diversi in C, diciamo Cs e Ct. Allora, esiste un qualche arco (p, q) sul cammino tra pi e pj in C*r che collega due cluster distinti in C.**

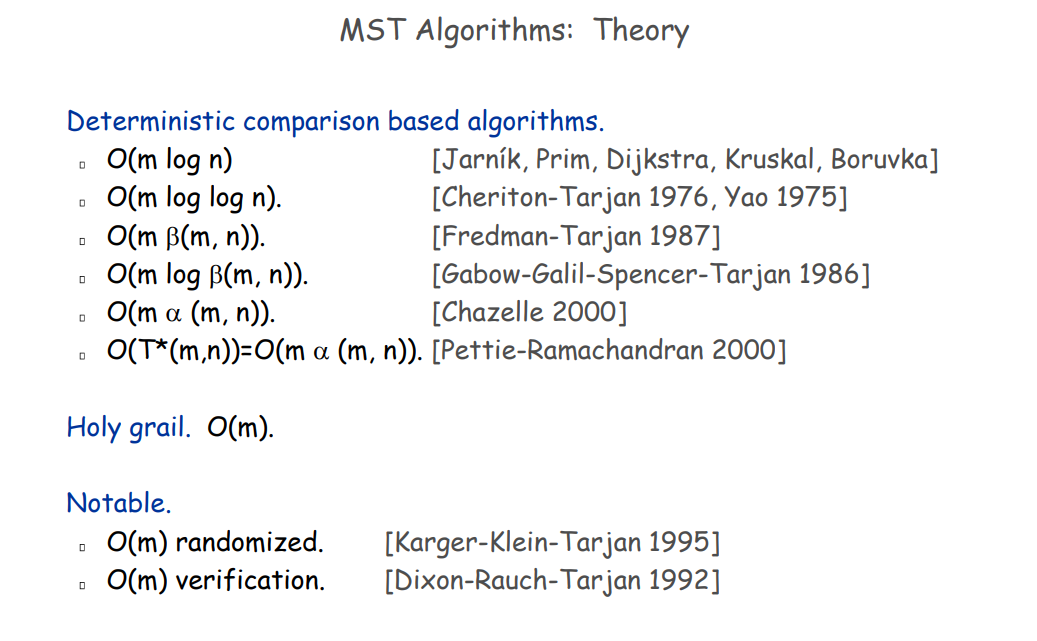
**Analisi:**

* **Tutti gli archi sul cammino pi-pj hanno lunghezza minore o uguale a d\*, poiché l'algoritmo di Kruskal li ha selezionati in questo modo.**
* **Di conseguenza, la separazione di C è al massimo d\*, dato che p e q appartengono a cluster diversi in C. (quadratino nero che indica la fine della dimostrazione) ▪**

**Spiegazione:**

1. **Definizione del Teorema: Il teorema afferma che il clustering ottenuto eliminando i k-1 archi più costosi dall'MST (C\*) ha una separazione massima rispetto ad altri k-clustering (C).**
2. **Dimostrazione:**
   * **Si introduce la variabile d\* che rappresenta la lunghezza del (k-1)esimo arco più costoso nell'MST. Questo valore rappresenta la separazione minima garantita per C\*.**
   * **Si considerano due oggetti (pi e pj) appartenenti allo stesso cluster in C\* ma a cluster diversi in un altro k-clustering (C).**
   * **Si dimostra che esiste un arco sul cammino tra pi e pj in C\* che collega due cluster distinti in C.**
   * **Poiché l'MST è costruito con archi in ordine crescente di costo, tutti gli archi sul cammino pi-pj in C\* hanno lunghezza minore o uguale a d\*.**
   * **Di conseguenza, qualsiasi coppia di oggetti appartenenti a cluster diversi in C non può essere più vicina di d\*, dimostrando che la separazione di C è al massimo d\*.**
3. **Conclusione: Il teorema stabilisce che l'algoritmo greedy di clustering single-linkage, eliminando i k-1 archi più costosi dall'MST, produce un clustering con una separazione massima rispetto ad altri k-clustering con lo stesso numero di cluster (k).**

****

****