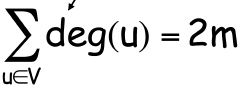
***8Hell Aimer. A te lettore, che con eterno timore, leggi queste righe, sappi che della tua sofferenza questo è solo l’inizio, ma sappi anche tutti gli inferni hanno un inizio e una fine, quindi affronta queste pagine con coraggio e siederai alla destra di Odino. HELL AIMER!!!***

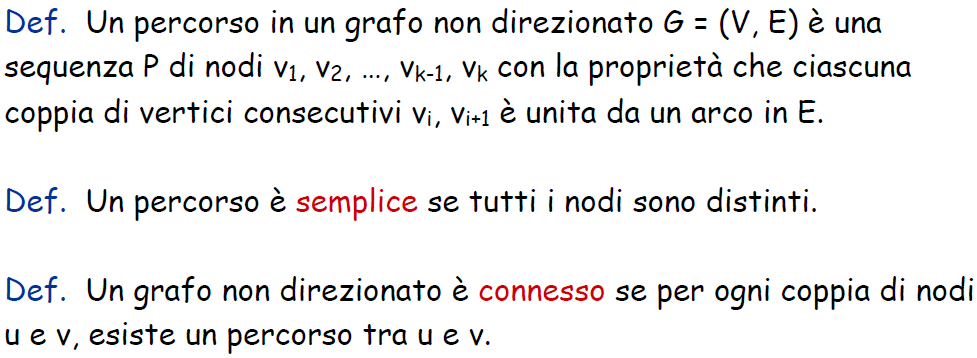
**Grafi:** insieme di **nodi** (**V**) collegati da un insieme di **archi** (**E**), definito come **G=(V,E).** Con la dicitura **deg(u)** dove **u** è un nodo del grafo, si intende il numero di archi che incidono il nodo.

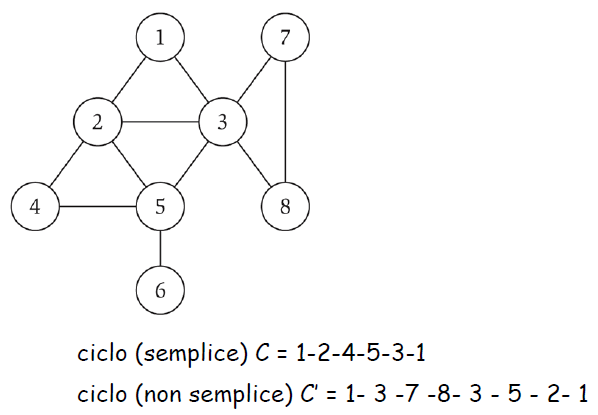
**Proprietà: m**:numero degli archi, **n**:numero dei nodi:

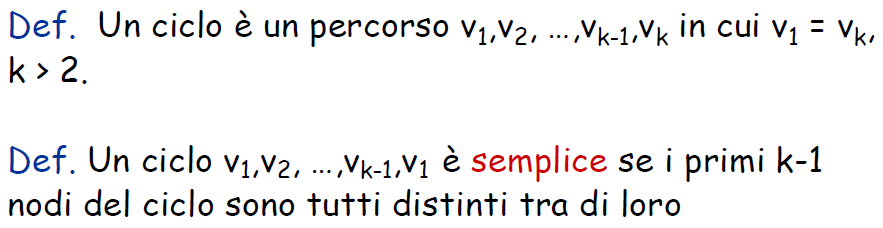
* La somma di tutti i gradi dei nodi di un grafo **G** è **2m (Dim:** banalmente ogni arco tocca due nodi quindi viene contato due volte all’interno della sommatoria**).**
* Il numero di archi in un grado **G** non direzionato è al più **n(n-1)/2 (Dim:** il primo nodo lo posso scegliere in **n** modi, il secondo invece in **n-1**, dimezzo il tutto in quanto un arco **(u,v)** è uguale a un arco **(v,u)).**
* Il numero **m** di archi in **G** è al più  **(Dim:** posso scegliere il primo nodo in **n** modi e il secondo in altrettanti **n** modi, se considero anche nodi aventi estremi uguali**).**

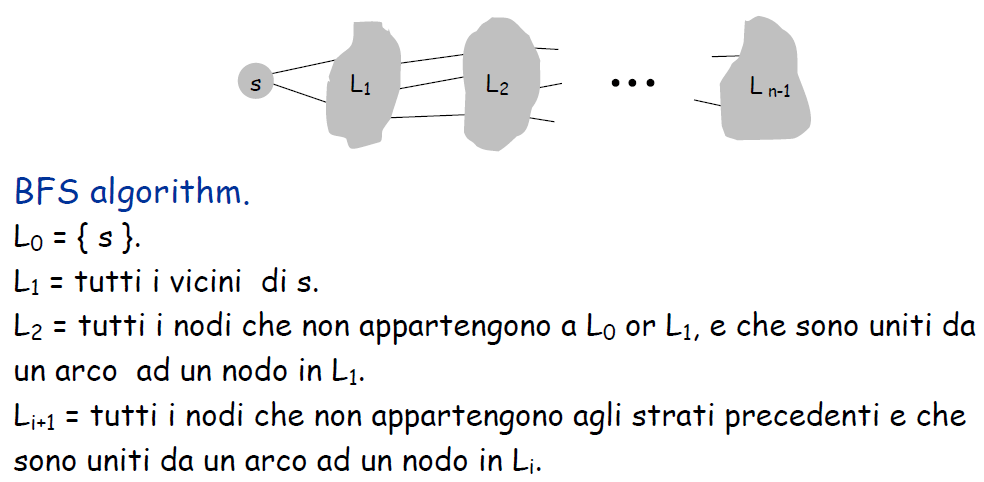
**Rappresentazione di grafi:**

| **Grafo** | **Matrice di Adiacenza** | **Lista di Adiacenza** |
| --- | --- | --- |
|  |  |  |

**Definizioni:** 

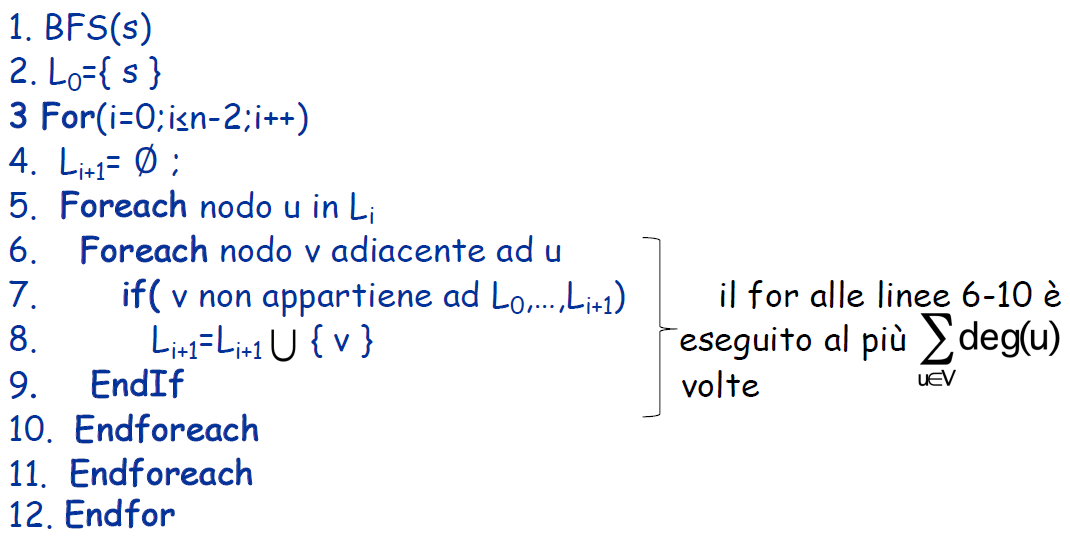


****

**Alberi:** un grafo non direzionato è un **albero** se è **connesso** e **non contiene cicli**. La **radice** **r** rappresenta il nodo in cui gli archi sono orientati a partire da **r**. Un **nodo genitore** è un nodo che precede un **nodo figlio** nel percorso, mentre un **nodo antenato** è qualsiasi nodo antecedente un altro. Una **foglia** è un nodo che non ha discendenti.

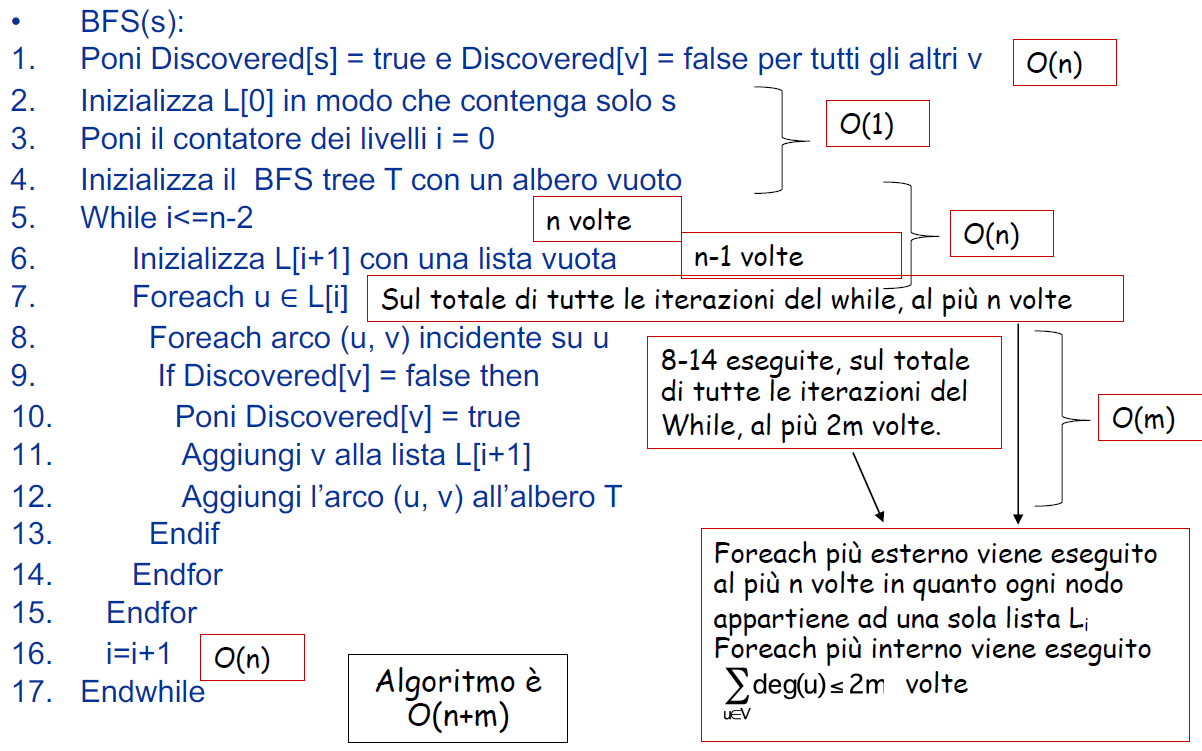
**Visite di grafi:**

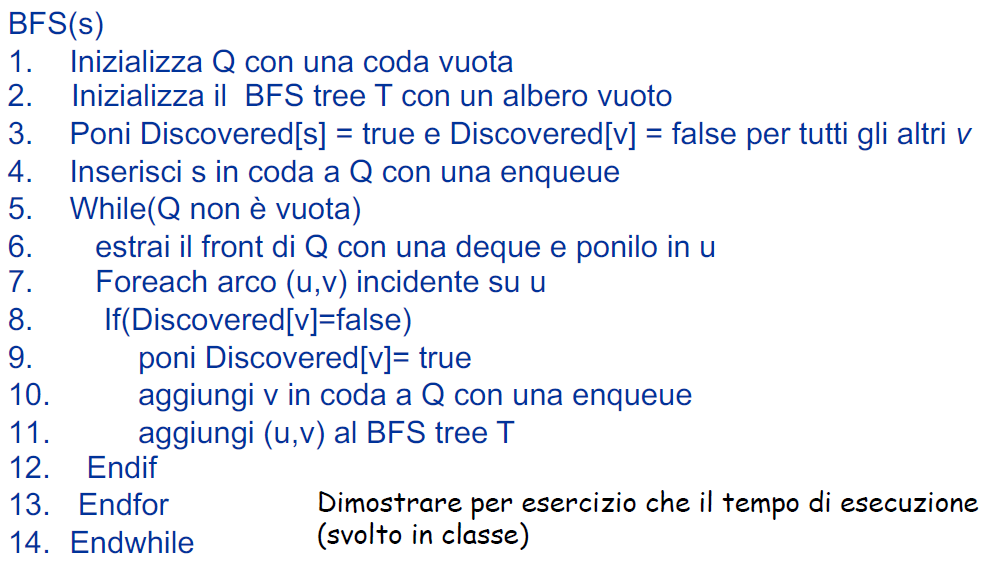
* **Algoritmo Breadth First Search (BFS):** Esplora il grafo a partire da una sorgente **s**, muovendosi in tutte le possibili direzioni e visitando i nodi livello per livello.

**Pseudocodice:**

La **profondità di un nodo u** è 0 se **u** è la radice di **T**, mentre è 1 + profondità del padre di **u**, altrimenti.

I nodi inseriti nel **layer** hanno profondità **i.**

**Implementazione con lista di adiacenza:**

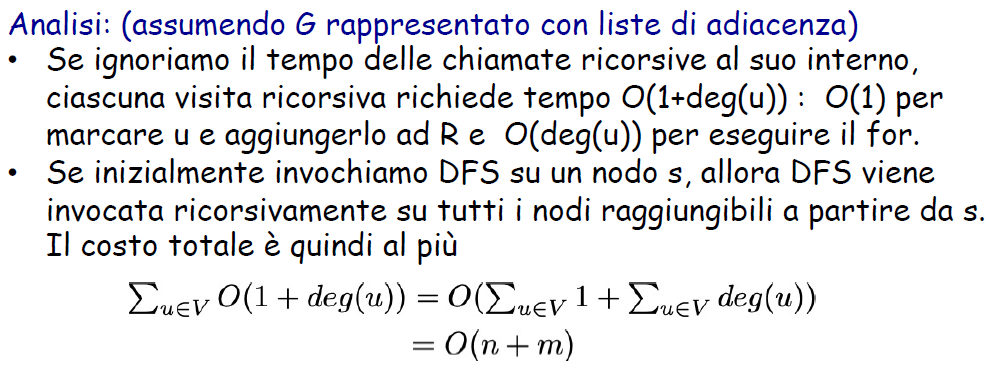
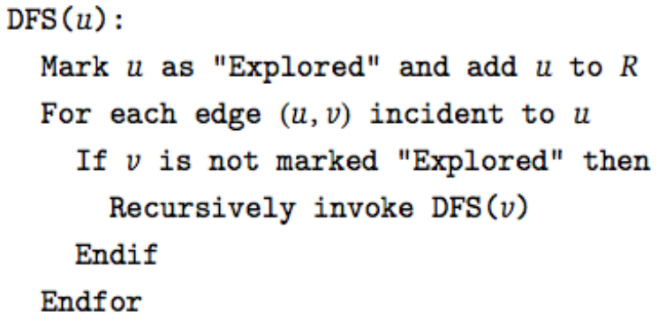
**Implementazione con coda FIFO:**

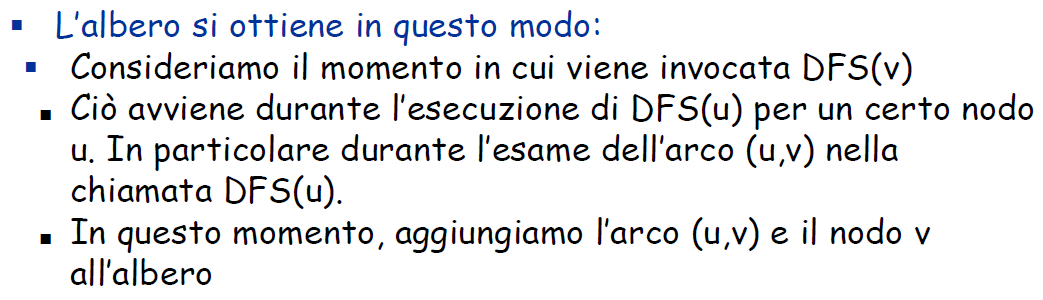
Viene estratta la sorgente e vengono **inseriti i nodi del livello 1**.

Poi man mano vengono **estratti i nodi del livello 1 ed inseriti quelli del livello 2**.

Quando **non ci sono più elementi** del livello 1, cominciano ad essere **estratti i nodi del livello 2** e via via **inseriti quelli del livello 3 e così via**.

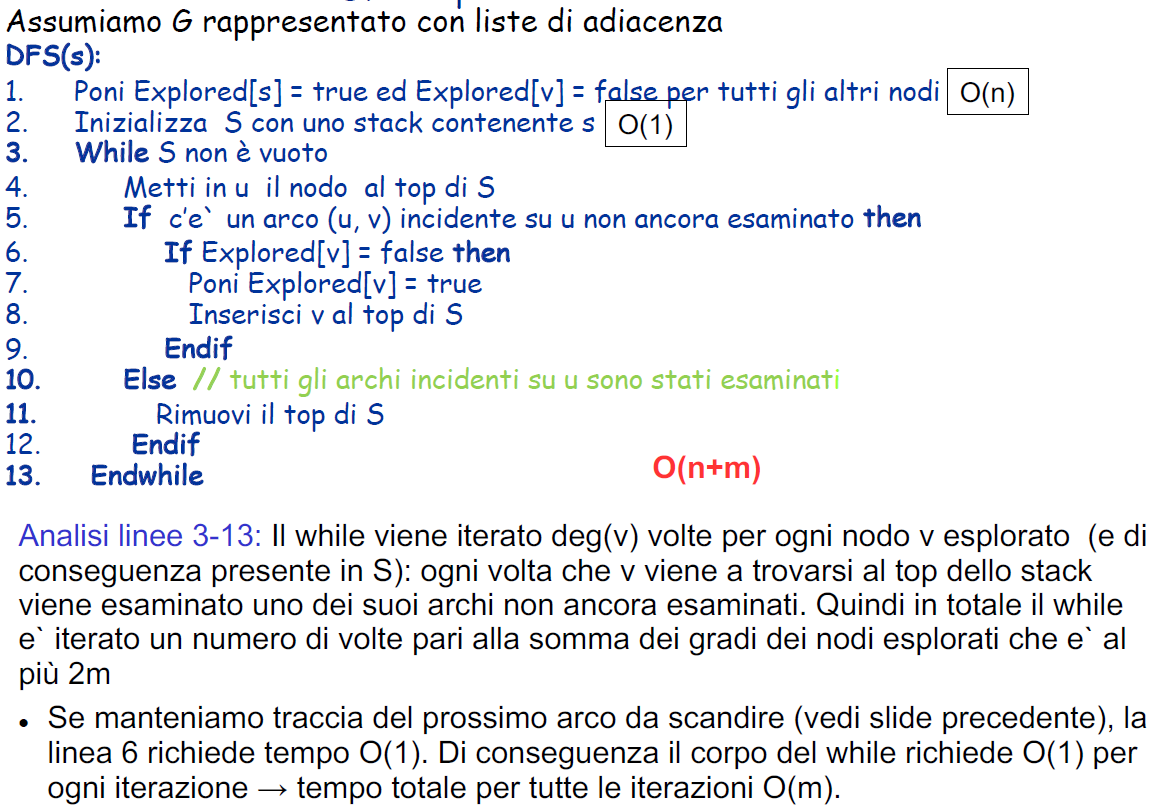
* **Algoritmo Depth First Search (DFS):** Riproduce il comportamento di una persona che esplora un labirinto. L’algoritmo si sposta da un nodo all’altro fin quando non arriva a un vicolo cieco, a questo punto torna indietro (**Backtrack**) e riinizia l’esplorazione fino a quando non ci sono più nodi da visitare.

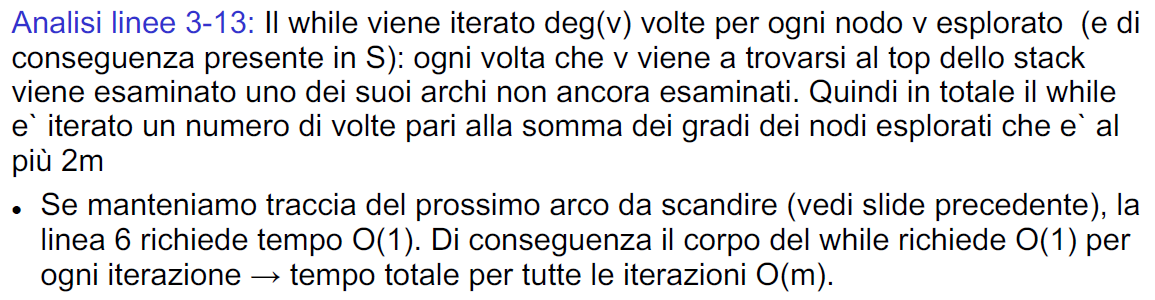
**Pseudocodice:**

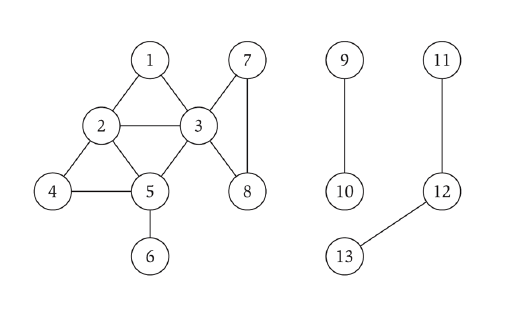
**R** rappresenta l’insieme dei vertici raggiunti.

Come il **BFS** il **DFS** produce un albero che ha come radice la sorgente **s** e come nodi tutti i nodi del grafo raggiungibili da **s**.

**Implementazione tramite stack:**

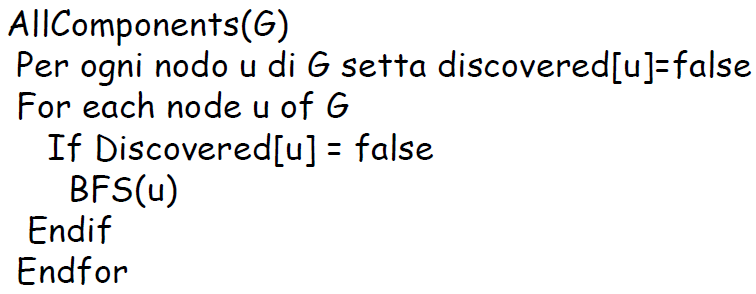
****

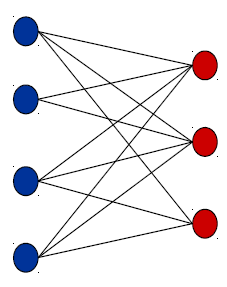


**Componente connessa:** Sottoinsieme di vertici tale che per ciascuna coppia di vertici **u** e **v** esiste un percorso tra **u** e **v.** 

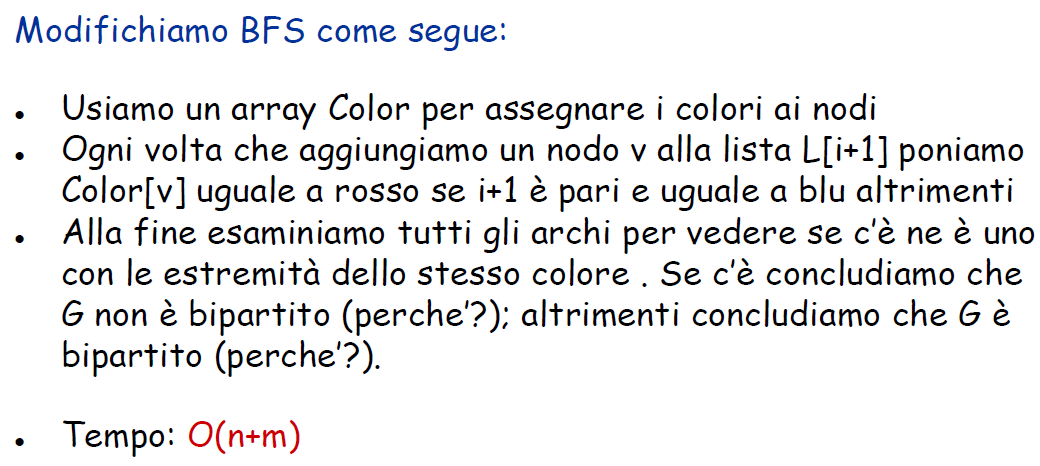
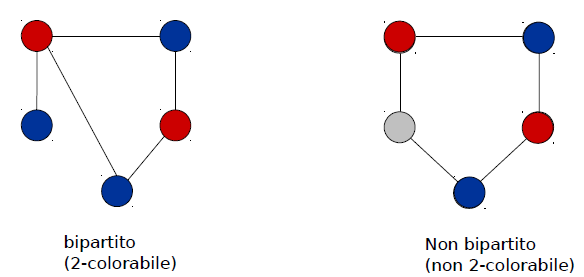
Nel grafo a destra per esempio, la componente connessa del **nodo 1** è: **{1, 2, 3, 4, 5, 6, 7, 8}.**

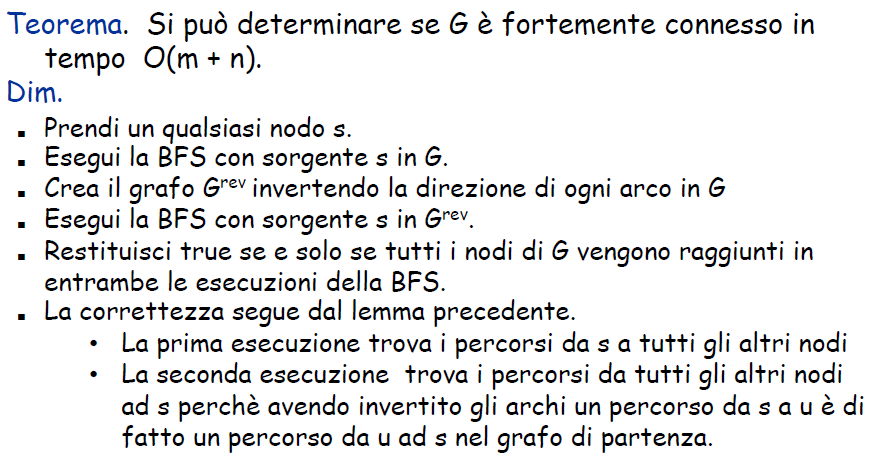
Per trovare la componente connessa di un determinato nodo **s** in un grafo si esegue l’algoritmo **BFS** o **DFS** a partire dal nodo **s**.

**Teorema:** Per ogni due nodi **s** e **t** di un grado, le loro componenti connesse o sono uguali o disgiunte. **(Dim:** Banalmente se esiste un cammino tra **s** e **t** le loro componenti connesse saranno uguali. Altrimenti se non esiste un cammino tra **s** e **t**, le componenti connesse saranno sicuramente diverse**).**

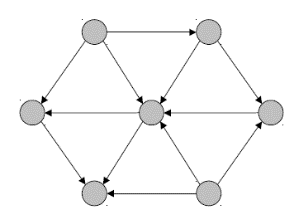
**Grafi bipartiti:** un grafo non direzionato è **bipartito** se l’insieme di nodi può essere partizionato in due sottoinsiemi **X** e **Y** tali che ciascun arco del grafo ha una delle due estremità in **X**  e l’altra in **Y.**

**Lemma:** se un grafo **G** è bipartito, non può contenere un ciclo dispari. **(Dim:** Non è possibile colorare di rosso e blu i nodi su un ciclo dispari in modo che ogni arco abbia le estremità di diverso colore**)**

Questo lemma ci permette di modificare il **BFS** per verificare se un grafo è bipartito.

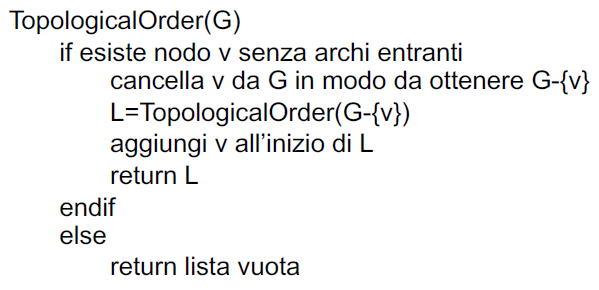
**Mutualmente raggiungibili:** due nodi **u** e **v** si dicono **mutualmente raggiungibili** se esiste un percorso tra **u** a **v** e un percorso da **v** a **u.**

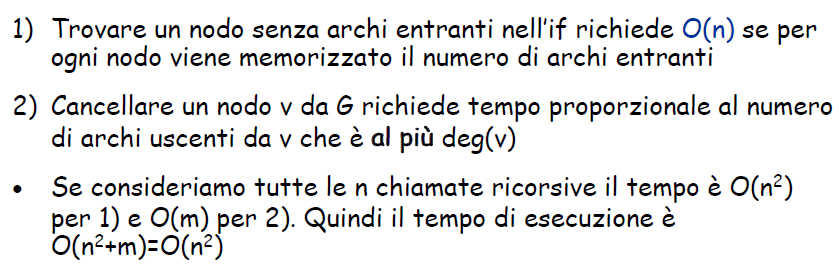
**Fortemente connesso:** un grafo **G** si dice **fortemente connesso** se ogni coppia di nodi è mutualmente raggiungibile.

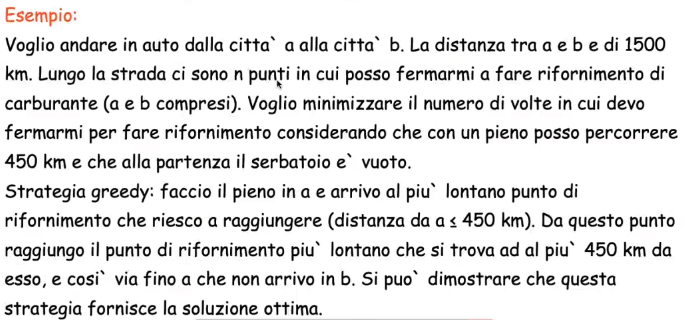
**DAG:** è un grafo direzionato che non contiene cicli direzionati. All’interno di un DAG viene detto **ordinamento topologico**, un etichettature dei suoi nodi **v1,v2,…,Vn**, tale che se **G** contiene un arco **(Vi, Vj)** si ha che **i<j.** 

**Alcuni Lemmi (dim sulle slide):**

* Se un grafo direzionato **G** ha un ordinamento topologico allora **G** è un **DAG**.
* Se **G** è un DAG allora **G** ha un nodo senza archi entranti.
* Se **G** è un DAG, **G** ha un ordinamento topologico.

**Algoritmo per l’ordinamento topologico:**

****

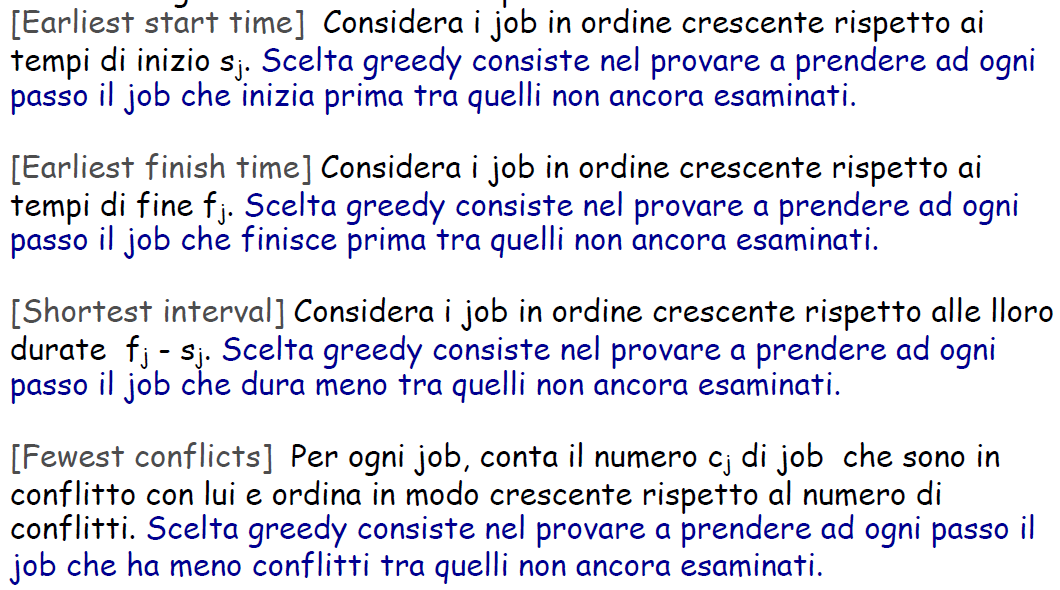
**Algoritmi Greedy:** è un paradigma di progettazione di algoritmi che viene utilizzato principalmente per problemi di ottimizzazione, si basa sul effettuare ad ogni passo la scelta che in quel momento sembra la migliore (**localmente ottima**) nella speranza di ottenere alla fine una soluzione che sia **globalmente ottima**. Questo approccio non funziona sempre ma per molte classi di problemi sì. 

Per ogni problema esistono vari approcci Greedy, il difficile sta nel trovare quello migliore in base al problema che si sta andando ad affrontare.

**Problemi svolti con algoritmi Greedy:**

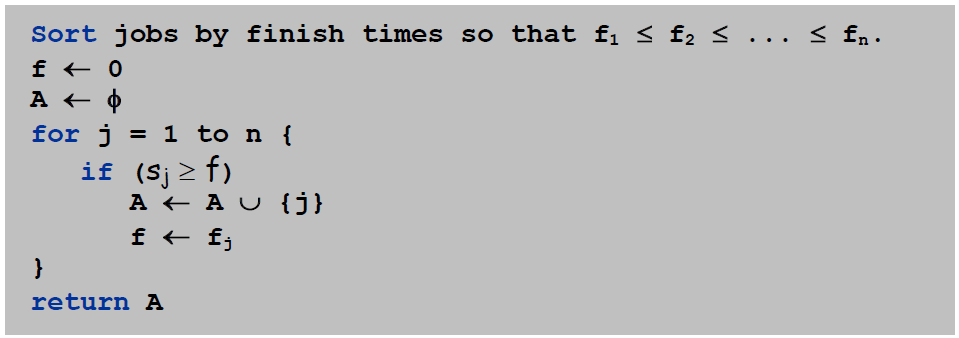
* **Interval Scheduling:** Abbiamo come input un insieme di attività ognuna che inizia in un instante e finisce in un instante , sapendo che può essere eseguita una singola attività alla volta, il nostro obiettivo è fare in modo che vengano svolte più attività possibili.

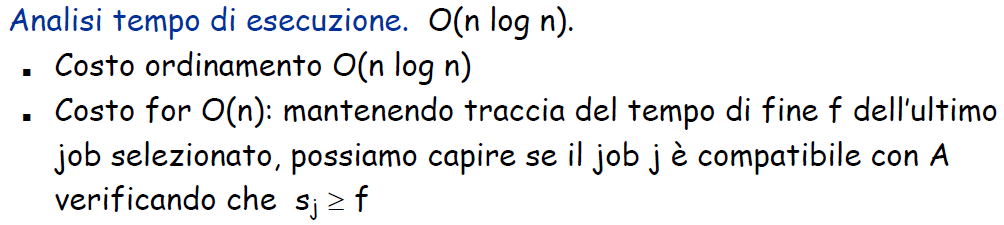
**Job Compatibili:** due job si dicono compatibili se oppure **,** in sostanza quando iniziano uno dopo l’altro e non si sovrappongono.

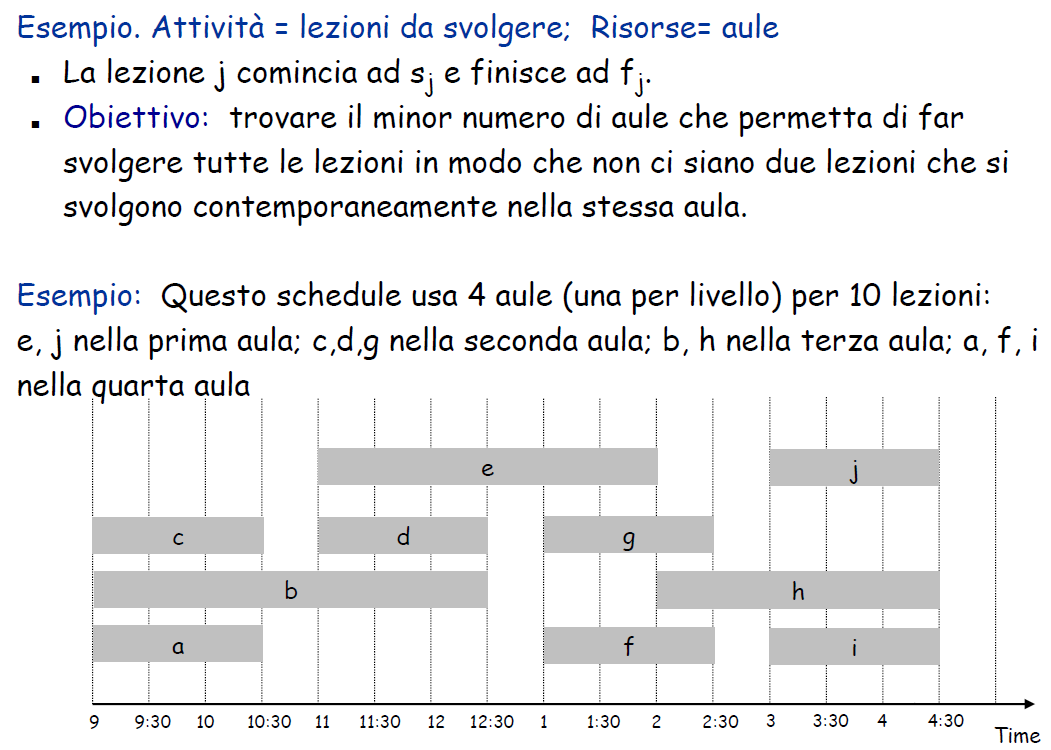
**Diverse strategie:** per risolvere il problema possiamo ricercare diverse strategie Greedy:

La strategia migliore è la **Earliest Finish Time**, ovvero selezionare prima i job che terminano prima. In modo da evitare i problemi legati alla altre, che risultato efficienti solo con alcuni input ben specifici e non nel caso medio.

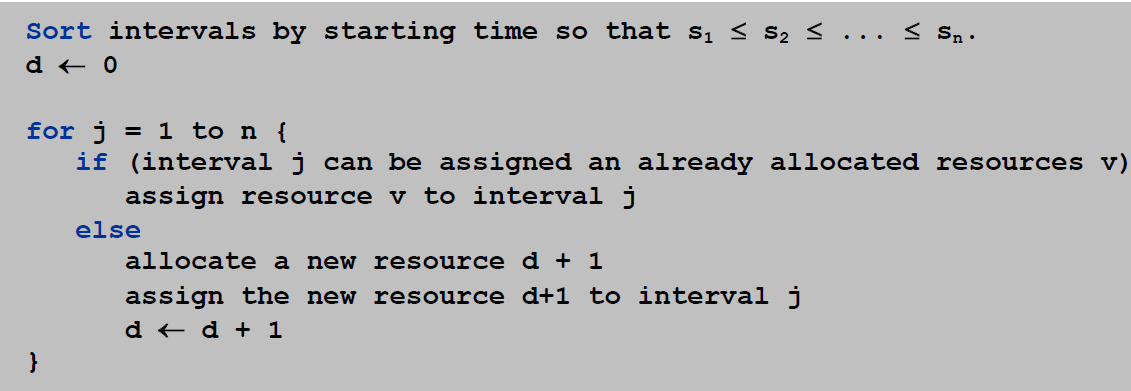
**Pseudocodice:**

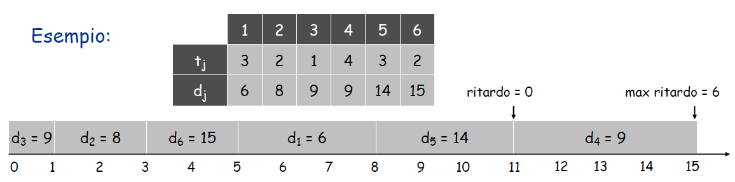
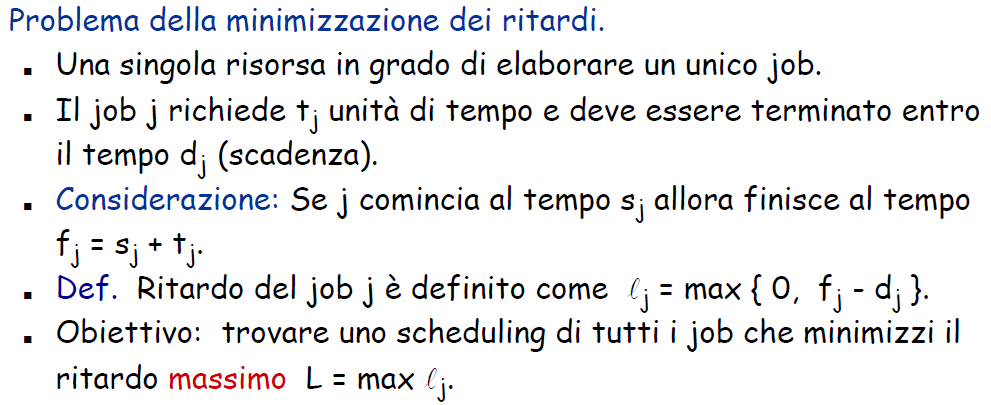
**Teorema:** L’algoritmo Greedy basato sulla strategia “**Earliest Finish Time**” è ottimo (**Dim:** slide Greedy 1)

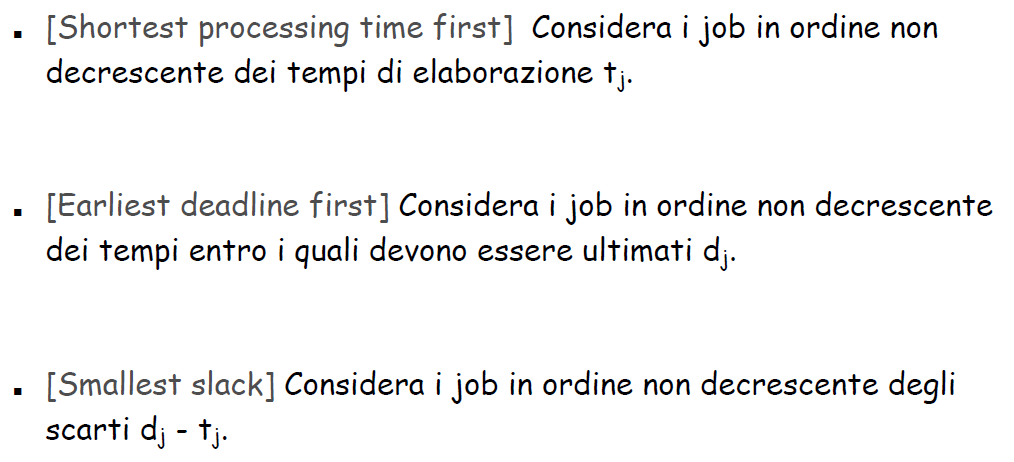


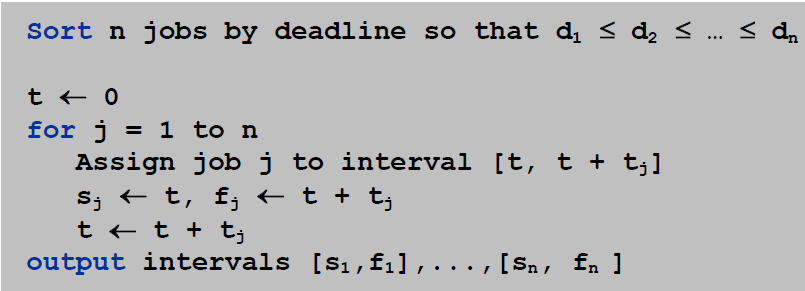
* **Partizionamento di intervalli:** In questo caso disponiamo di **più risorse identiche** tra di loro e **vogliamo che vengano svolte tutte le attività** in modo tale da usare il **minor numero di risorse** e tenendo contro del fatto che **due attività non possono usufruire della stessa risorsa** allo stesso tempo.

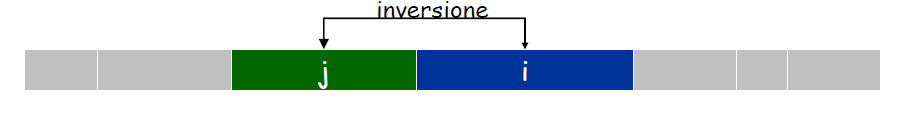
**Profondità:** rappresenta il numero massimo di intervalli intersecabili con una singola linea verticale che si muove lungo l’asse delle ascisse. (**Numero di risorse >= profondità**).

**Pseudo codice:**

* **Minimizzare i Ritardi:** il problema consiste nel minimizzare i ritardi nell’esecuzione dei vari task da una singola risorsa.

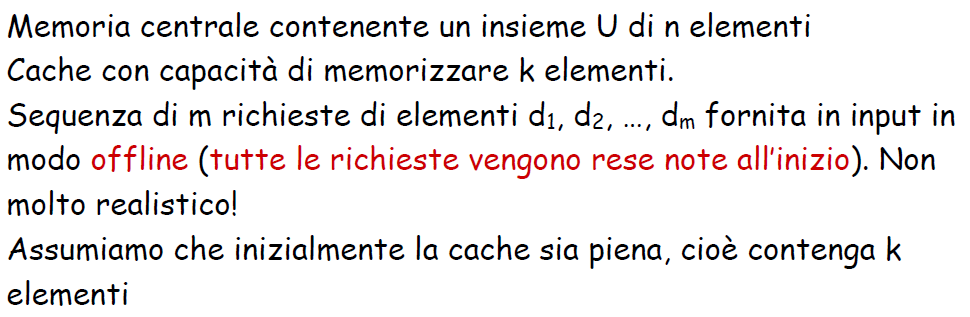
**Strategie risolutive Greedy:** anche in questo caso abbiamo diverse strategie Greedy, che considerano i job in un ordine diversi.

**Pseudocodice della soluzione ottima:** 

L’algoritmo si basa sul concetto di **inversione** che altro non è che data una coppia di job **i j** in questo ordine, viene eseguita prima **j** e poi **i**. 

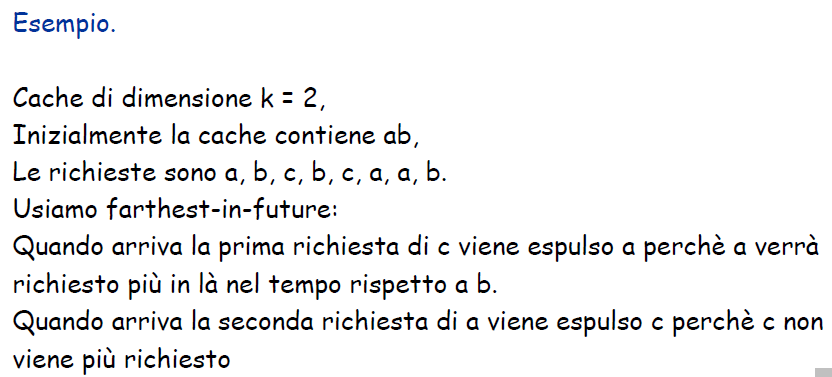
* **Caching offline ottimale:** è un problema che risolve il problema di gestire la memoria cache di un calcolatore:
  + **Cache hit:** elemento già presente nella cache quando richiesto.
  + **Cache miss:** elemento non presente nella cache quando richiesto: occorre portare l’elemento richiesto nella cache e se la cache è piena occorre espellere dalla cache alcuni elementi per fare posto a quelli richiesti.

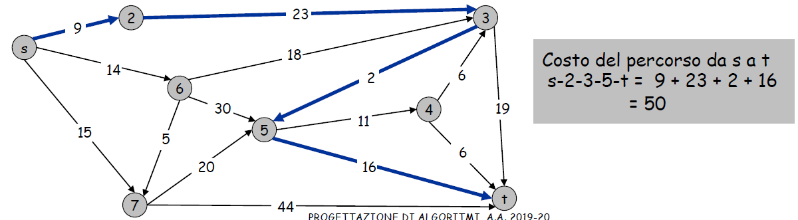
**Formalizzazione del problema:**

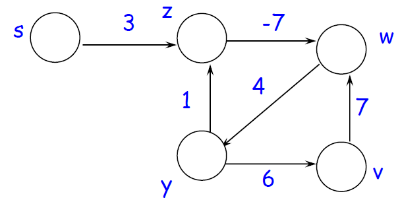
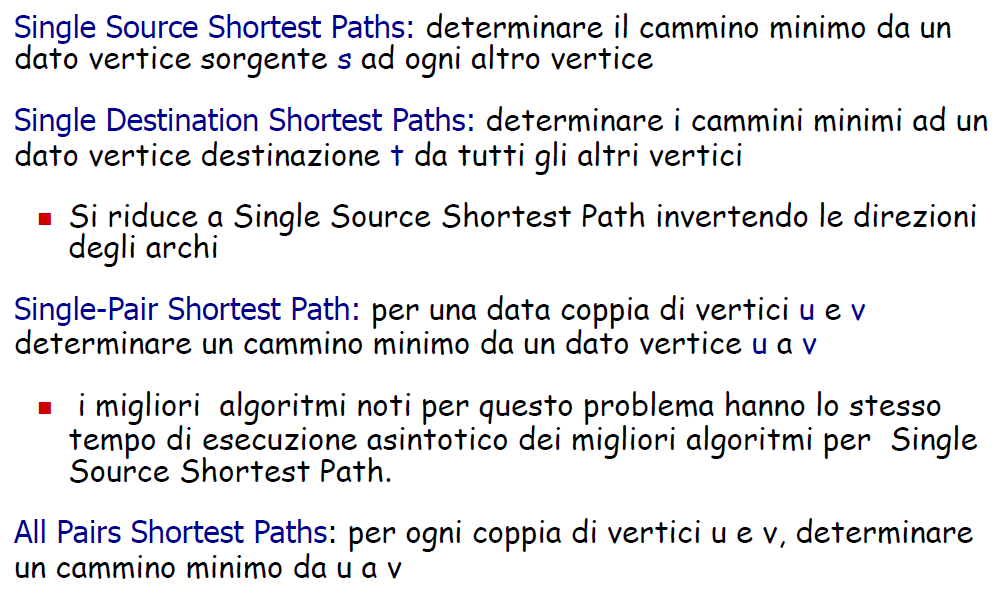


**Eviction schedule ridotto:** è uno scheduling degli elementi da espellere, cioè una sequenza che indica quale elemento espellere quando c’è bisogno di far posto ad un elemento richiesto che non è in cache. Ne esistono di due categorie:

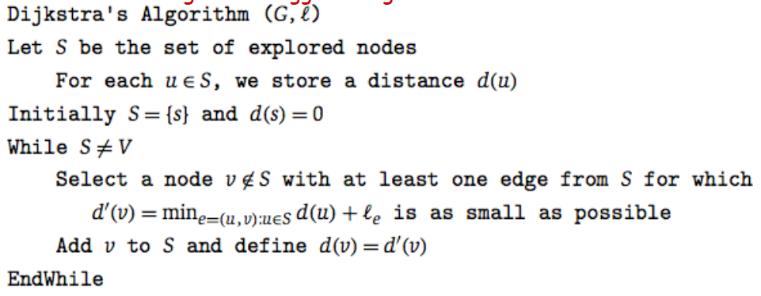
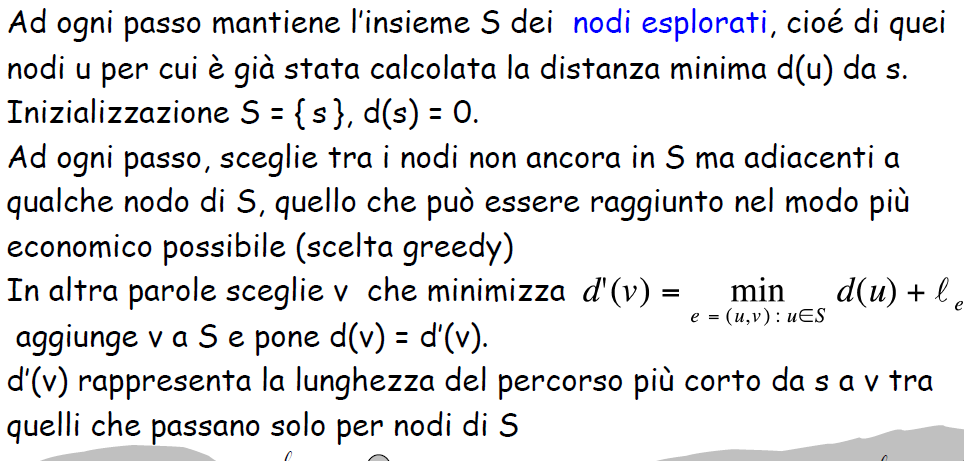
* + **Non ridotti:** lo scheduler può decidere di inserire in cache un elemento che non è stato richiesto.
  + **Ridotto:** inserisce in cache un elemento solo nel momento in cui è richiesto e se non è presente già in cache al momento della richiesta.

**Soluzione ottimale – Farthest-in-future:** Quando viene richiesto un elemento che non è presente in cache, espelli dalla cache l’elemento che sarà richiesto più in là nel tempo o che non sarà più richiesto.

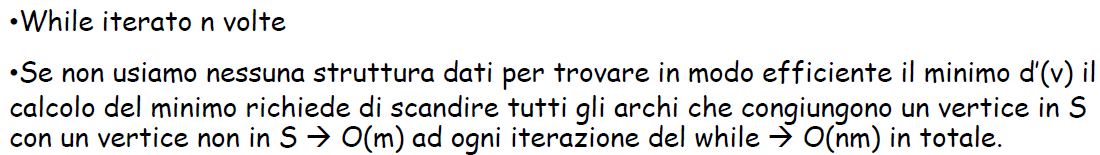
* **Cammini Minimi:** Dato un grafo direzionato **G** all’interno del quale ad ogni arco è associato un peso, si vuole trovare il percorso con peso minore da un nodo sorgente **s** a un nodo destinazione **t**.

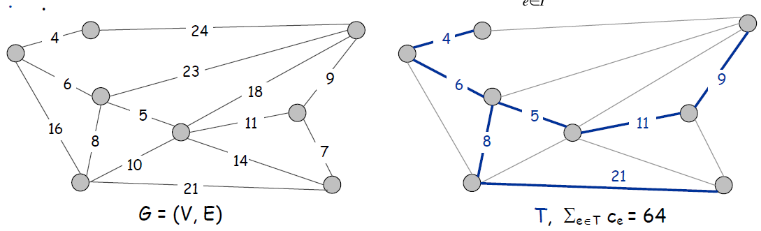
**Varianti:**

**Cicli negativi:** se esiste un ciclo negativo lungo un percorso da **s** a **v,** allora non è possibile definire il cammino minimo da **s** a **v**. Banalmente passando da un ciclo negativo il costo andrebbe a diminuire. O potrebbe essere negativo. (il peso del ciclo in figura è -2)

**Algoritmo di Dijkstra:**

**Analisi tempo di esecuzione:**



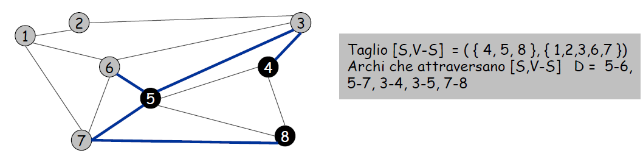
* **Minimo albero ricoprente (Minimum Spanning tree):** presa una rete di nodi interconnessi, abbiamo l'obiettivo di utilizzare **n-1 collegamenti** tra coppie di posizione per connettere l’intera rete minimizzando la somma degli **n-1** collegamenti stabiliti.

**Albero ricoprente:** dato un grafo **G**, uno **Spanning Tree** è un insieme di archi **T** all’interno del quale non sono presenti cicli.

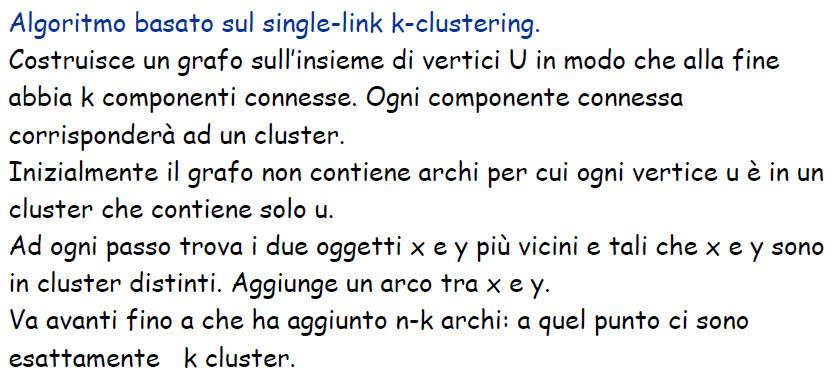
**Diversi algoritmi:**

* + **Kruskal:** Comincia con T = f. Considera gli archi in ordine non decrescente di costo. Inserisce un arco e in T se e solo il suo inserimento non determina la creazione di un ciclo in T.
  + **Inverti-Cancella:** Comincia con T = E. Considera gli archi in ordine non crescente dei costi. Cancella e da T se e solo se la sua cancellazione non rende T disconnesso.
  + **Prim:** Comincia con un certo nodo s e costruisce un albero T avente s come radice. Ad ogni passo aggiunge a T l’arco di peso più basso tra quelli che hanno esattamente una delle due estremità in T ( se un arco avesse entrambe le estremità in T, la sua introduzione in T creerebbe un ciclo).

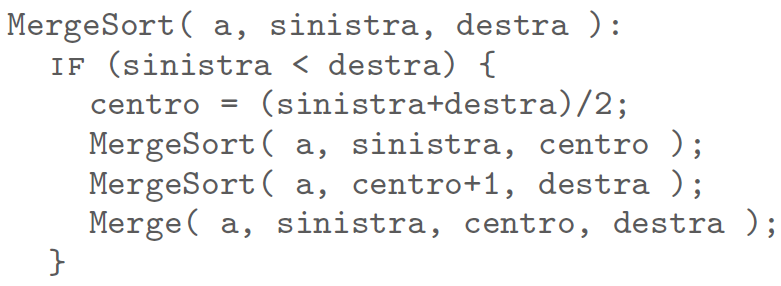
**Taglio:** è una partizione [S,V-S] dell’insieme dei vertici del grafo.

**Insieme di archi che attraversano il taglio [S,V-S]**: Sottoinsieme D di archi che hanno un’estremità in S e una in V-S.

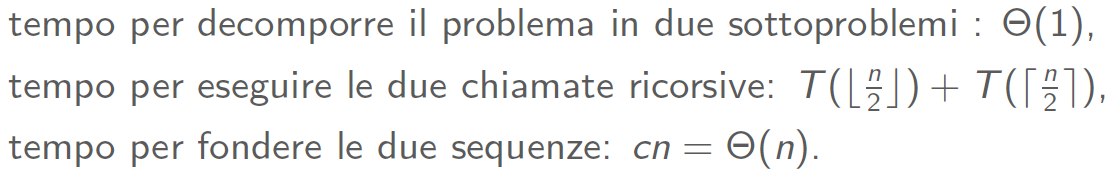
| **Pseudocodice Prim:** | **Pseudocodice Kruskal:** |
| --- | --- |
|  |  |

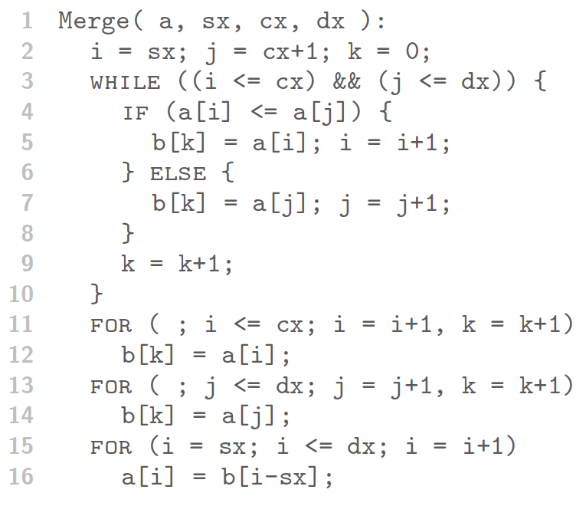
* **Clustering:** Dato un insieme **U** di oggetti P1, P2, … , Pn vogliamo classificarli in gruppi coerenti.

**Dividi et Impera:** consiste nel decomporre il problema in un piccolo numero di **sottoproblemi**, ciascuno dei quali della stessa tipologia ma definito su un insieme di dati più piccolo. Successivamente si risolve ricorsivamente ciascun sottoproblema fino a che non si arriva a risolvere sotto-problemi di **taglia così piccola da poter essere risolti direttamente** (senza effettuare ulteriori chiamate ricorsive) infine si combinano le soluzioni dei sottoproblemi al fine di ottenere una soluzione al problema di partenza.

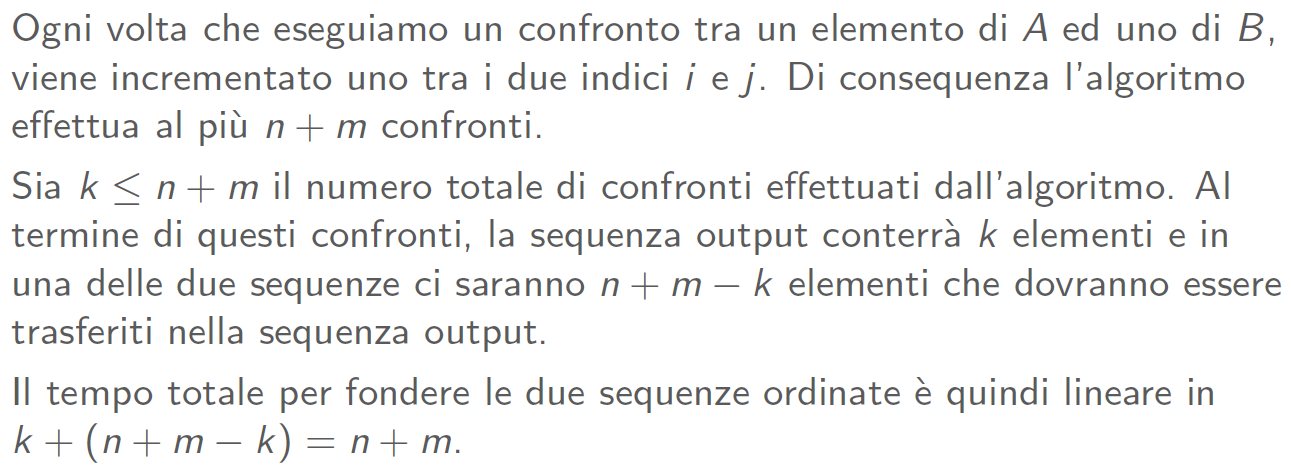
* **MergeSort:** ordina un vettore di numeri in ordine decrescente, dividendo quest’ultimo in parti sempre più piccole (fino ad arrivare a vettori di 2) per poi unirli tra di loro.

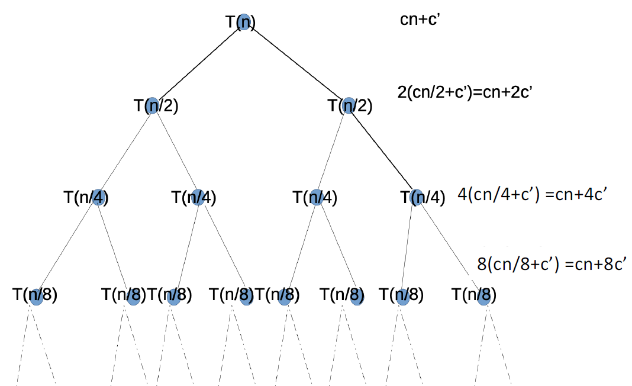
**Tempo di esecuzione:**

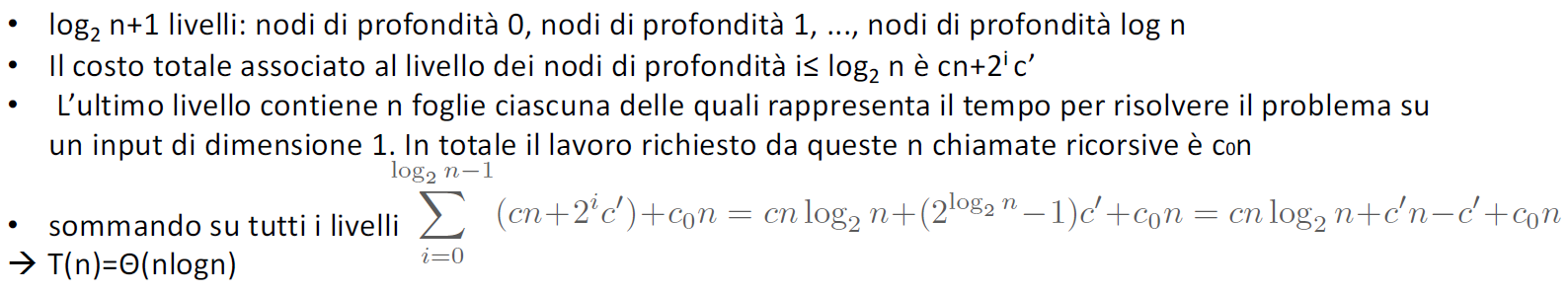


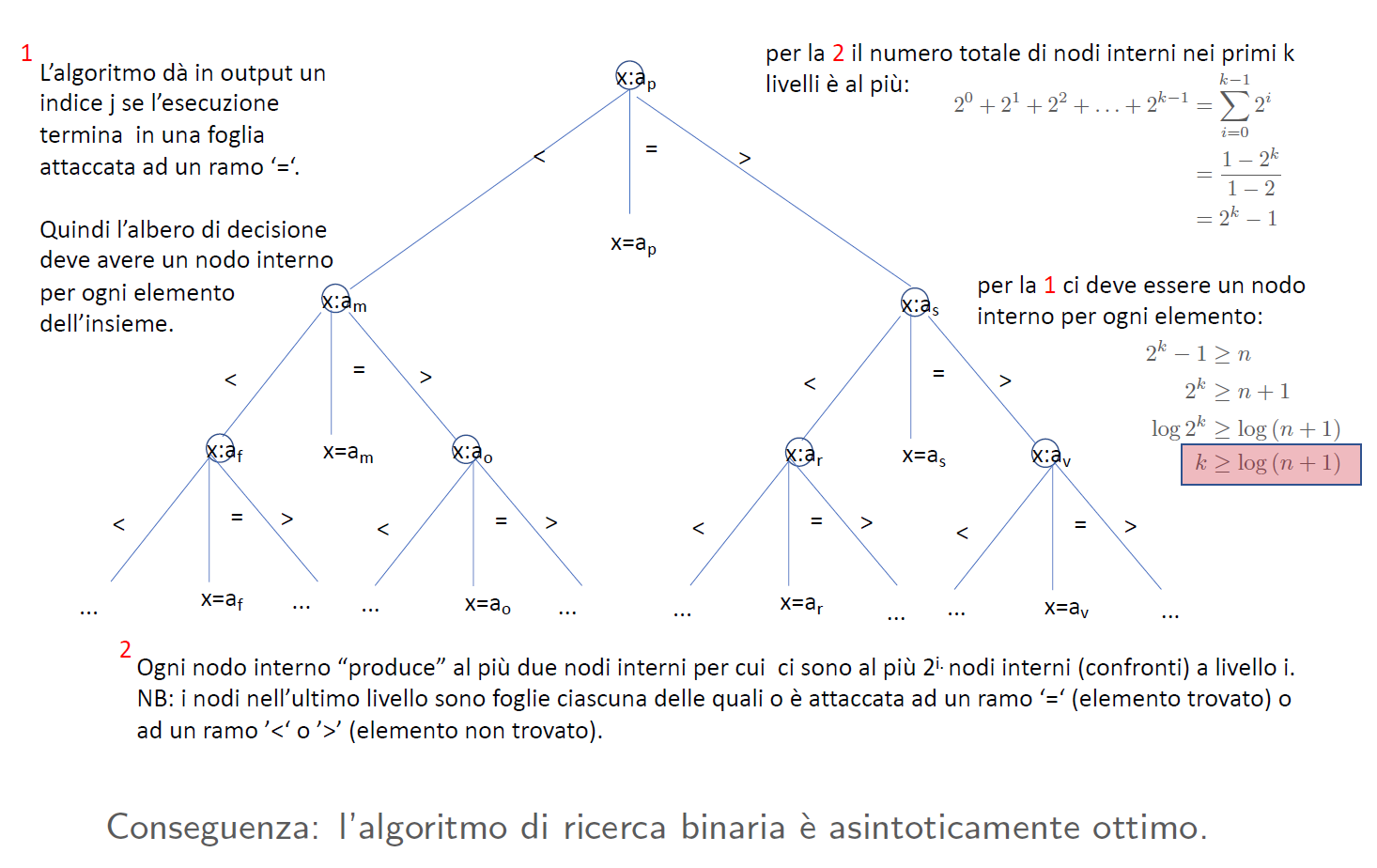
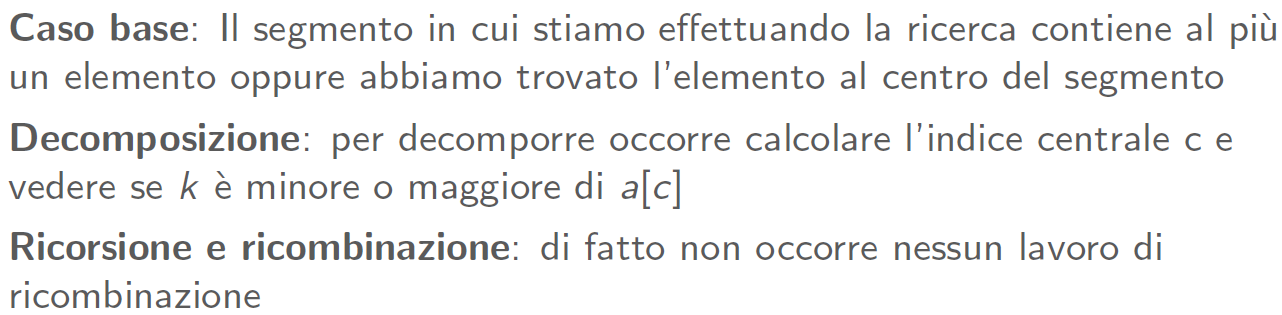
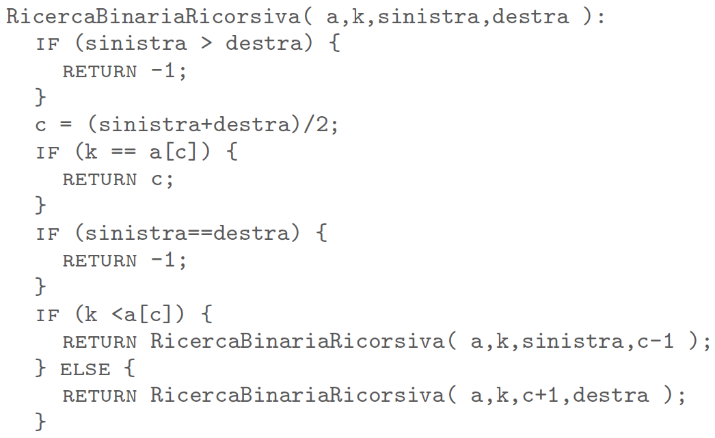
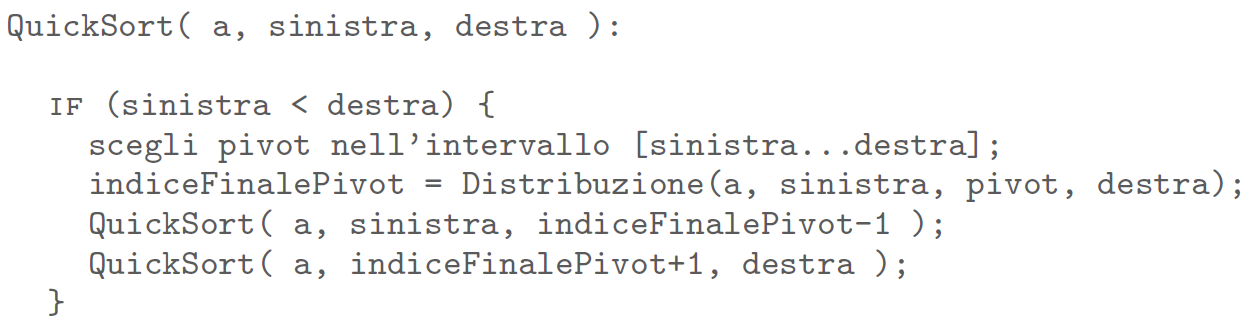
**Algoritmo fusione:** permette di fondere due sequenze di numeri adiacenti in un unico array:

il suo tempo di esecuzione è:

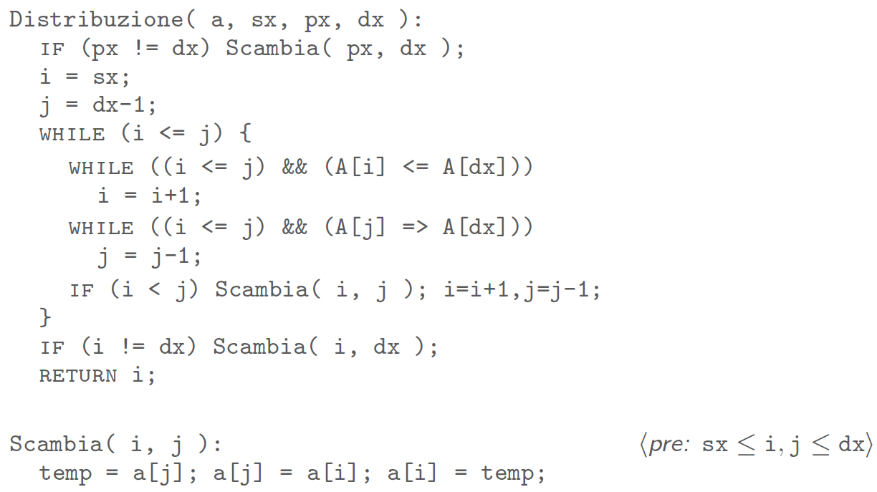






* **Ricerca binaria**: trova all’interno di una sequenza ordinata di numero un determinato elemento, l’algoritmo si basa sulla divisione successiva dell'array in modo da ridurre sempre di più il range di valori da analizzare finché non rimane solo l’elemento cercato.
* **Quicksort (Ordinamento per distribuzione):** se la sequenza ha almeno due elementi, scegli un elemento pivot e dividi la sequenza in due sottosequenze in modo tale che la prima contenga elementi minori o uguali al pivot e la seconda gli elementi maggiori o uguali del pivot. Successivamente ordina le due seggenze.

**Algoritmo Distribuzione:**

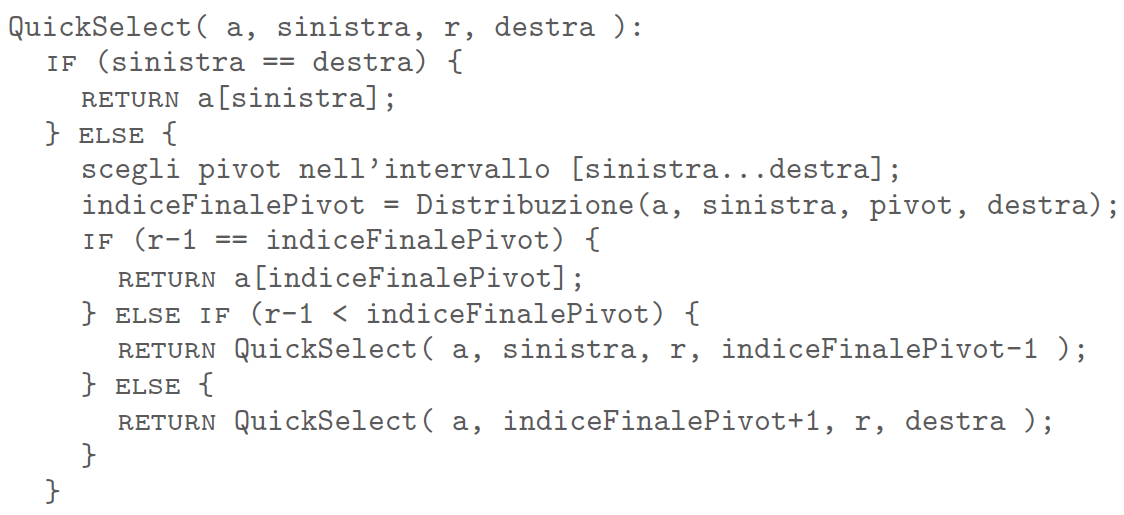


**Tempo di esecuzione:** è uguale a O(n log n), nel caso in cui scegliamo un pivot in modo random.

* **QuickSelect:** L'obiettivo è selezionare un elemento di rango **r** all’interno di un array **a** di **n** elementi distinti. Inoltre:
  + Vogliamo evitare di ordinare **a**;
  + Il problema diventa quello di trovare il minimo quando r = 1 e il massimo quando r = n.

la funzione Distribuzione permette di trovare il rango del pivot, posizionando tutti gli elementi di rango inferiore alla sua sinistra e tutti quelli di rango superiore alla sua destra.

**Pseudocodice:**



**Efficienza del QuickSelect randomizzato:** vale il discorso analogo al quello del Quicksort, ci sono molte possibili scelte del pivot che fanno in modo che l’algoritmo si comporti bene. Scegliendo infatti il pivot in modo random è probabile che si scelga un pivot “ben posizionato”.

* **Moltiplicazione di interi:** esiste un algoritmo per migliorare questa semplice operazione (**Slide 8/14 parte 3**)
* **Sottosequenza di somma massima di un array si numeri:** Dato un array **a**, di n numeri positivi e negativi trovare la sottosequenza di numeri consecutivi la cui somma è massima.

N.B. Se l’array contiene solo numeri positivi, il massimo si ottiene banalmente prendendo come sequenza quella di tutti i numeri dell’array; se l’array contiene solo numeri negativi il massimo si ottiene prendendo come sottosequenza quella formata dalla locazione contenente il numero più grande.