ALGORITMO: descrizione di un processo di calcolo non ambiguo

COMPLESSITA': funzione che valuta la velocità di esecuzione di un algoritmo tenuto conto dell'architettura e della dimensione del problema

Tale parametro deve essere calcolato il più accuratamente possibile ovvero il *limite inferiore* (caso ottimo) deve essere il più vicino possibile al *limite superiore* (caso pessimo).

$f \in O(g)$ con *g* funzione: *f* asintoticamente non è più grande di *g*

$f \in \Omega(g)$ con *g* funzione: *f* asintoticamente non è più piccolo di *g*

Quando caso pessimo e caso ottimo coincidono si dice che la complessità di un algoritmo è definita esattamente.

$$f \in \theta(g) \leftrightarrow f \in O(g) \; e \; f \in \Omega(g)$$ con *g* funzione

Se $A \in O(f)$, con *A* algoritmo, vuol dire che la complessità di $A \in O(f)$ ovvero il tempo di esecuzione di *A* non è superiore ad *O*.

Se $A \in \Omega(f)$, con *A* algoritmo, vuol dire che la complessità di $A \in \Omega(f)$ ovvero il tempo di esecuzione di *A* non è inferiore ad *O*.

TECNICHE PER RICAVARE LA COMPLESSITÀ DI UN ALGORITMO:

- Equazione di ricorrenza → metodo matematico.
- Albero di ricorrenza → ogni ramo rappresenta una chiamata ricorsiva, la complessità è data dalla somma dei valori sui nodi.
- Teorema dell'esperto → quando i sottoproblemi hanno tutti dimensioni paragonabili permette di calcolare la complessità mediante relazioni di ricorrenza. Il teoremafornisce il comportamento asintotico di un algoritmo come funzione della dimensione dell'input, tralasciando le costanti additive e moltiplicative.

COMPLESSITA' AMMORTIZZATA

Dato un algoritmo avente complessità bassa la maggior parte delle volte e alta raramente, posso distribuire la complessità del caso pessimo sugli altri casi. In questo modo, considerando più alta la complessità delle operazioni con complessità bassa, quando arriverò all'operazione con complessità alta potrò considerare che abbia la complessità delle precedenti perché sopravvalutando le complessità precedenti l'ho ammortizzata.

PROBLEMA: relazione tra input e output

Se $P \in O(f)$ implica che esiste un $A \in O(f)$ ovvero un algoritmo in grado di risolvere il problema in questione.

$$Se \; P \in O(f) \rightarrow A \in O(f) \; allora \; \exists \; A \; che \; risolve \; P$$

Se $A \in \Omega(f)$ implica chequalsiasi $P \in \Omega(f)$ ovvero non esistono algoritmi che impiegano meno di *f* a risolvere *P*.

$$Se \; A \in \Omega(f) \rightarrow P \in \Omega(f) \; allora \; \nexists \; A \; che \; risolve \; P \; più \; velocemente \; di \; f$$

Se $P \in \theta(f)$ implica che $\exists$ A che risolve *P* in $\Omega(f)$ ed è unico, quindi ottimo.

ORDINAMENTO → TOTALE
→ PARZIALE

ALGORITMO DI ORDINAMENTO: riceve in input una sequenza di oggetti su cui è definita una relazione di ordinamento e fornisce come output una permutazione di tali oggetti tale che ogni oggetto sia ≥/≤ del successivo.

STABILITA': un algoritmo di ordinamento è detto stabile se non scambia l'ordine relativo di elementi uguali

ORDINAMENTO IN LOCO: un algoritmo ordina in loco se ha bisogno sempre della stessa quantità di memoria per funzionare (esclusi i dati stessi)

ALGORITMI DI ORDINAMENTO

INSERTION SORT (stabile, ordina in loco, complessità n^2)

```
InsertionSort(array A) {
for j ← z to length[A] {
            key ← A(j)
            i ← j-1
while i >= 0 and A[j] >key{
A[j + 1] ← A[j]
j ← j-1
            }
        A[i+1] ← key}}
```

Si assume che la sequenza da ordinare sia partizionata in una sottosequenza già ordinata, all'inizio composta da un solo elemento, e una ancora da ordinare. Alla *k*-esima iterazione, la sequenza già ordinata contiene *k* elementi. In ogni iterazione, viene rimosso un elemento dalla sottosequenza non ordinata (scelto, in generale, arbitrariamente) e inserito (da cui il nome dell'algoritmo) nella posizione corretta della sottosequenza ordinata, estendendola così di un elemento.

MERGE SORT (stabile, non locale, complessità *n lg(n)*)

```
MergeSort(a[], left, right){
ifleft< right then{
center ← (left + right) / 2
mergesort(a, left, center)
mergesort(a, center+1, right)
merge(a, left, center, right)}}
```

Il merge sort è un algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del *Divide et Impera*, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola.

Concettualmente, l'algoritmo funziona nel seguente modo:

1. Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti:
2. La sequenza viene divisa (*divide*) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda)
3. Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo(*impera*)
4. Le due sottosequenze ordinate vengono fuse (*merge*). Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata

HEAPIFY (non stabile, locale, complessità logaritmica)

```
Heapify(A, i){
      l ← Left (i)
      r ← Right (i)
      if (l ≤ A.heap-size and A[l] > A[i])
            largest ← l
      else
            largest ← i
      if r ≤ A.heap-size and A[r] > A[largest]
            largest ← r
      iflargest ≠ i
      exchange A[i] ↔ A[largest]
      Heapify (A, largest)
}
```

Ordina un albero binario facendo risalire ogni elemento verso la radice finchè non ha padre maggiore.
La complessità di questo algoritmo è definita da

$$\sum_{h=0}^{\lg(n)} h\frac{n}{2^{h+1}} \leq \sum_{h=0}^{\infty} h\frac{n}{2^{h+1}}$$

Si noti che l'espressione dopo il simbolo di disuguaglianza è la rappresentazione di una serie geometrica di ragione < 1 che converge ad una costante → complessità lineare

BUILD HEAP(non stabile, non locale, complessità lineare)

```
BuildHeap(A){
      HeapSize(i) ← length(A)
      for i ← length(A)/2 to 1
            Heapify(A,i)
}
```

Partendo da un array crea un albero utilizzando la HEAPIFY. Complessità pari a # iterazioni* complessità(Heapify) → →n * costante → complessità lineare

HEAPSORT(non stabile, ordina in loco, complessità *n lg(n)*)

```
HeapSort(A){
      BuildHeap(A)
      for i ← length(A) to 2 {
            scambia(A[1], A[i])
            HeapSize(A)--
            Heapify(A, i) }}
```

Partendo da un array utilizza Heapify e BuildHeap per costruire un albero binario ordinato.

Complessità pari alla complessità della BuildHeap + quella di ogni Heapify.

$$T(n) \in n + lg(n-1) + lg(n-2) + \cdots + lg(2) = n + \sum_{i=2}^{n-1} lg(i) \rightarrow\ n + lg(n-1)\,! \rightarrow\ n\ + n \cdot lg(n) \rightarrow \boldsymbol{n \cdot lg(n)}$$

QUICKSORT(non stabile, ordina in loco, complessità n^2 o *n lg(n)*)

```
QuickSort(A, p, r){
      if p<r {
            q ← Partition(A, p, r)
            QuickSort(A, p, q)
            QuickSort(A, q+1, r) }}

Partition(A, p, r){
      x ← A[p]
      i ← p-1
      j ← r+1
      while(true){
            do
                  j ← j-1
            while (!(A[j]≤x))
            do
                  i ← i+1
            while (!(A[i]≥x))
            if i<j
                  switch(A[i], A[j])
            else
                  ret j }}
```

Quicksort è un algoritmo di ordinamento; la base del suo funzionamento è l'utilizzo ricorsivo della procedura partition: preso un elemento da una struttura dati, che chiameremo *elemento pivot*, si spostano gli elementi minori a sinistra rispetto a questo e gli elementi maggiori a destra.

Il Quicksort, termine che tradotto letteralmente in italiano indica ordinamento rapido, è l'algoritmo di ordinamento che ha, in generale, prestazioni migliori tra quelli basati su confronto.

Nel caso pessimo l'algoritmo ha complessità quadratica:

$$\boldsymbol{T(n)} = n + T(n-1) + T(1) =$$
$$= n + (n-1) + T(n-2) =$$
$$= n + n - 1 + n - 2 + \cdots + 1 =$$
$$= \frac{n(n-2)}{2} \in \boldsymbol{\theta(n^2)}$$

Prendendo l'elemento pivot a caso, la quasi totalità delle volte ($\approx \frac{1}{n!}$) l'array verrà diviso in parti non vuote e la probabilità di ritrovarmi nel caso pessimo diminuisce di molto.

```
RandomPartition(A, p, r){
      k ← Random(p, r)
      switch(A[p], A[k])
      Partition(A, p ,r) }
```

L'algoritmo QuickSort con RandomPartition è più veloce di HeapSort nella maggior parte dei casi e per questo motivo utilizzabile in situazioni nelle quali è necessario ordinare spesso. Un ordinamento dal costo maggiore non compromette nel complesso le prestazioni del sistema.

Complessità di QuickSort con RandomPartition:

$$\boldsymbol{T(n)} = n + \frac{1}{n}\big(T(1) + T(n-1)\big) + \frac{1}{n}\big(T(2) + T(n-2)\big) + \cdots + \frac{1}{n}\big(T(n-1) + T(1)\big)$$
$$= n + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) + \big(T(n-i)\big) \approx \boldsymbol{n \cdot \mathbf{lg}(n)}$$

Come si nota dagli algoritmi riportati in precedenza, il problema dell'ordinamento tramite confronti $\in \Omega(n \cdot \lg(n))$
Tramite ipotesi aggiuntive e/o conoscendo a priori determinate proprietà dei dati di partenza è possibile ordinare in tempo minore.

COUNTING SORT(stabile, non ordina in loco, complessità lineare)

```
CountingSort(A, B, k){
      for i ← 1 to k
            C[i] ← 0
      for j ← 1 to length(A)
            C[a[j]]++
      for i ← 2 to k
            C[i] ← C[i]+C[i-1]
      for j ← length(A) to 1
            B[C[A[j]]]
            C[A[j]]--   }
```

Algoritmo di ordinamento per valori numerici interi con complessità lineare. L'algoritmo si basa sulla conoscenza a priori dell'intervallo in cui sono compresi i valori da ordinare.

L'algoritmo conta il numero di occorrenze di ciascun valore presente nell'array da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervallo di valori. Il numero di ripetizioni dei valori inferiori indica la posizione del valore immediatamente successivo.

La complessità dipende dalla lunghezza dell'array da ordinare "*n*" e dall'intervallo preso in considerazione "*k*":

$$\boldsymbol{T(n) = n + k}$$

È importante considerare che per valori di k >> n conviene ordinare in *n lg(n)*.

RADIX SORT (ordina in loco, complessità lineare)

1° PASSO		2° PASSO		3° PASSO		4° PASSO		
253		10		5		5		5
346		253		10		10		10
1034	→	1034	→	127	→	1034	→	127
10		5		1034		127		253
5		346		346		253		346
127		127		253		346		1034

Radixsort utilizza un procedimento controintuitivo per l'uomo, ma più facilmente implementabile per ordinare valori numerici interi. Esegue gli ordinamenti per posizione della cifra partendo dalla cifra meno significativa.

La complessità $\in O(nk)$, dove n è la lunghezza dell'array,*k* è la media del numero di cifre degli *n* numeri.

$$\boldsymbol{T(n) \in O(nk)}$$

BUCKET SORT

Il Bucketsort è un algoritmo di ordinamento per valori numerici che si assume siano distribuiti uniformemente in un intervallo semichiuso (0,1).
L'intervallo dei valori, noto a priori, è diviso in intervalli più piccoli, detti bucket (cesto). Ciascun valore dell'array è quindi inserito nel bucket a cui appartiene, i valori all'interno di ogni bucket vengono ordinati e l'algoritmo si conclude con la concatenazione dei valori contenuti nei bucket.

La complessità del bucketsort è *O(n)* per tutti i cicli, a parte l'ordinamento dei singoli bucket. Date le premesse sull'input, utilizzando insertionsort l'ordinamento di ogni bucket è dell'ordine di *Θ(1),* quindi la complessità totale è *O(n)* per tutto l'algoritmo.

$$\boldsymbol{T(n) \in O(n)}$$

SELEZIONE

INPUT: Array di elementi su cui è definita una relazione di ordinamento.

OUTPUT: elemento "x" che nell'ordinamento di A occupa la posizione i.

Il problema della selezione può essere risolto in tempo $\mathbf{O}(\boldsymbol{n}\ \mathbf{lg}(\boldsymbol{n}))$.

Per determinare il minimo o il massimo in un insieme di elementi sono necessari n-1 confronti: complessità lineare. Gli algoritmi sono ottimi.

```
Minimum(A){
      min ← A[i]
      for i ← 2 to length(A)
            ifmin> A[i]
                  min ← A[i]
      retmin }
```

```
Maximum(A){
      max ← A[i]
      for i ← 2 to length(A)
            ifmax< A[i]
                  max ← A[i]
      retmax }
```

RANDOM SELECT

L'algoritmo segue lo stesso modello del QuickSort per determinare l'i-esimo elemento più piccolo.

```
RandomSelect(A, p, r, i){
      if p = r
            ret A[p]
      q ← RandomPartition(A, p, r)
      k ← q-p+1
      if i ≤ k
            retRandomSelect(A, p, q, i)
      else
            retRandomSelect(A, q+1, r, i-k) }
```

La complessità media dell'algoritmo è lineare perchè lavora soltanto su un lato della partizione:$\boldsymbol{T(n) \in \theta(n)}$

Caso migliore: $\boldsymbol{T(n) = n + \sum_{i=1}^{n-1} \frac{1}{n} T(i) + T(1) \rightarrow lineare}$

Caso peggiore: $\boldsymbol{T(n) = n + T(n-1) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = n^2}$

Avendo una complessità lineare nella maggior parte ma non nella totalità dei casi, se non mi posso permettere di avere ricerche lente, sebbene sporadiche, meglio cercare in *n lg(n)* (ordino con *Merge Sort* ed estraggo).
Altrimenti utilizzo *Random Select.*

STRUTTURE DATI

→*FONDAMENTALI*: Alberi binari di ricerca, RB-Alberi, Tabelle Hash.
→*EVOLUTE*: Heap binomiali, Strutture dati per insiemi disgiunti.

Al variare della rappresentazione dei dati in memoria la complessità di un algoritmo può cambiare molto. Per definire una struttura dati è necessario stabilire un insieme di elementi sui quali definire delle operazioni.

Dato un insieme di assiomi e di relazioni tra essi vi sono molte espressioni con lo stesso significato. All'interno di queste classi di equivalenza posso scegliere ogni volta un rappresentante che prende il nome di **forma canonica**.

ALBERI BINARI DI RICERCA

Fanno parte delle strutture dati fondamentali per rappresentare liste concatenate. Si rappresenta ogni nodo di un albero binario con un oggetto e si assume che ogni nodo contenga un campo key e dei campi puntatori ai nodi padre e figli destro e sinistro.

Se l'albero è bilanciato ha profondità lg(n) con n il numero dei nodi dell'albero e di conseguenza la ricerca si esegue in tempo logaritmico.

RB-ALBERI

Un RB-Albero è un albero binario di ricerca con in più un campo binario in ogni nodo che ne indica il colore (RED / BLACK) che soddisfa le seguenti proprietà:

- ogni nodo è RED o BLACK
- ogni foglia (NIL) è nera
- i figli di un nodo RED sono BLACK
- ogni cammino radice-foglia contiene lo stesso numero di nodi BLACK

B-ALTEZZA: numero di nodi neri su un cammino da un nodo ad una foglia senza contare il nodo di partenza

NODO INTERNO: nodo non foglia

Ogni sottoalbero radicato contiene almeno $2^{BH(x)} - 1$ nodi interni

INSERZIONE (complessità lg(n))

Una volta inserito un elemento all'interno dell'albero lo si colora di rosso e quindi si fa in modo che le proprietà di cui sopra vengano rispettate sistemando i colori dei vari nodi ed eseguendo rotazioni.

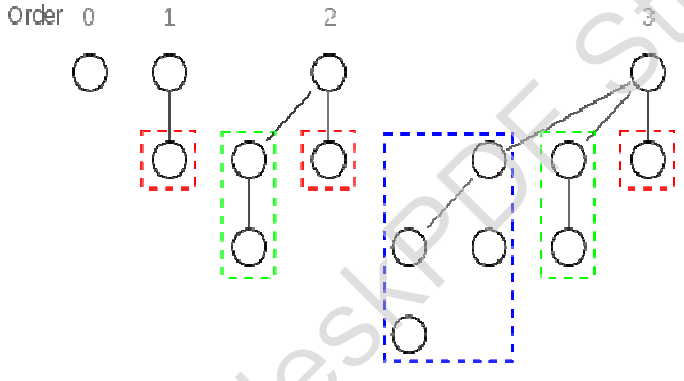
ESTRAZIONE(complessità lg(n))

Dopo aver estratto un nodo, viene richiamata una procedura che ha il compito di cambiare i colori ed eseguire le rotazioni necessarie affinchè vengano mantenute le proprietà RB.

In generale, un campo è mantenibile in tempo logaritmico, se dipende solo dalla propria chiave e dalla chiave del sotto albero radicato nel nodo stesso.

HEAP BINOMIALI→ Lista di alberi binomiali

Sono strutture dinamiche ampiamente ricorsive, basate sull'aritmetica binaria e con forma simile agli heap.



Proprietà in B_k:

- Ci sono 2^k nodi (il numero di nodi cresce in modo esponenziale con la dimensione)
- L'altezza è *k*
- A profondità i corrispondono $\binom{k}{i}$ nodi Coeff. Binomiale $\binom{k}{i} = \frac{k!}{i!\cdot(k-i)!}$
- La radice ha grado *k*
- I figli da sinistra a destra sono radici di alberi di dimensione via via decrescente (B_{k-1}, B_{k-2},…, B_0)

- ∀ dimensione *k* al più un albero B_k
- I nodi di ogni albero soddisfano le proprietà di ordinamento di uno heap
- I figli hanno chiave ≥ del padre
- Dato *n* è possibile costruire uno heap binomiale di *n* nodi
- È possibile trovare l'elemento minimo in tempo logaritmico

TABELLE HASH

Funzione che mappa grandi quantità di chiavi in array molto più piccoli.
Per rappresentare un insieme dinamico, si usa un array, o una tabella ad indirizzamento diretto in cui ogni posizione corrisponde ad una chiave. Quando l'insieme *k* di chiavi da memorizzare è molto più piccolo dell'insieme di tutte le possibili chiavi, una tabella *Hash* è più efficiente di un array o di una tabella ad indirizzamento diretto.

Con l'indirizzamento diretto un elemento di chiave *k* è memorizzato nella posizione *k* mentre in una tabella *Hash* l'elemento *k* è memorizzato nella posizione *h(k)*:cioè si usa una *funzione Hash h* che definisce una corrispondenza tra l'insieme di chiavi e la posizione in cui saranno memorizzate.

La funzione Hash ottima deve essere in grado di mappare l'insieme delle chiavi in maniera omogenea nell'array ovvero il ***fattore di carico α*** di ogni posizione nell'array deve essere lo stesso.

$$\alpha = \frac{N}{n} \; N \rightarrow numero\ di\ elemento\ totali \; n \rightarrow num\ di\ posizioni\ dell'array$$

INSIEMI DISGIUNTI

Struttura dati caratterizzata da 3 operazioni:

1. MakeSet(x) crea un insieme composto dal solo elemento x → complessità costante
2. Union(x,y) unisce due insiemi → complessità lineare
3. FindSet(x) trova l'insieme di appartenenza di x → complessità costante

Ogni insieme è identificato da un rappresentante, che è un dato membro dell'insieme.

→ *UNIONE PER RANGO*: per ottimizzare l'operazione di unione aggiungo il campo RANGO alla struttura. Questo campo indica il numero di elementi dell'insieme ed è popolato solo nel rappresentante. Affinchè questo accorgimento sia realmente efficace occorre unire sempre la lista con rango minore all'altra in modo da dover cambiare il minor numero di puntatori possibile.Perché l'unione per rango abbia complessità lineare è necessario aggiungere un ulteriore campo, popolato anch'esso solo nel rappresentante, che punta all'ultimo elemento della lista: LAST.

Il costo di *m* operazioni di cui *n*MakeSet nel caso pessimo è:

- SENZA unione per rango $m + n^2$
- CON unione per rango $m + n\ lg(n)$

PROGRAMMAZIONE DINAMICA

La programmazione dinamica risolve problemi computazionali mettendo assieme le soluzioni di un certo numero di sottoproblemi. Questo metodo si può applicare anche quando i sottoproblemi non sono indipendenti, ovvero quando la soluzione di alcuni sottoproblemi richiede di risolvere i sottoproblemi stessi. In questi casi un algoritmo divide-et-impera fa più lavoro di quello necessario dato che risolve più volte i sottoproblemi comuni. Gli algoritmi di programmazione dinamica invece risolvono ciascun sottoproblema una sola volta e memorizzano la soluzione in una tabella: in questo modo si evita di calcolare nuovamente la soluzione ogni volta che deve essere risolto lo stesso sottoproblema. Questa tecnica è detta *MEMOIZZAZIONE*.

La programmazione dinamica generalmente viene adottata per risolvere problemi di ottimizzazione. Tuttavia, per alcuni problemi di ottimizzazione l'uso di tecniche di programmazione dinamica per decidere la scelta migliore risulta inutilmente oneroso: lo stesso risultato potrebbe infatti essere ottenuto con algoritmi più semplici ed efficaci.

PROGRAMMAZIONE GREEDY

Gli algoritmi greedy adottano la strategia di prendere in ogni punto di scelta dell'algoritmo quella decisione che, al momento, appare come la migliore senza preoccuparsi se tale decisione porterà ad una soluzione ottima del problema nella sua globalità.

GRAFI: strutture dati formate da nodi e archi. G = (V , E)
Se due nodi sono collegati da un arco **non orientato** si dice che sono un **insieme di due nodi**, se l'arco è **orientato o directed**si parla di **coppia di nodi.**

BIPARTITO: tale che l'insieme dei suoi vertici si può partizionare in due sottoinsiemi tali che ogni vertice di una di queste due parti è collegato solo a vertici dell'altra
SPARSO: ha pochi archi rispetto ai nodi
CONNESSO: se per ogni insieme di nodi esiste un cammino che li collega
FORTEMENTE CONNESSO: se per ogni coppia di nodi esiste un cammino che li collega
ACICLICO: grafo che non contiene cicli
GRADO USCENTE: numero di archi che escono da un nodo
GRADO ENTRANTE: numero di archi che entrano in un nodo

CAMMINO: Si chiama cammino (path) di lunghezza p di estremi a e b in un grafo G = (V,E) una sequenza di p+1 vertici (u0, u1,..., up) tale che a = u0, b = up, e (ui-1, ui)∈E
CICLO o **CIRCUITO**(se non ci sono nodi ripetuti): un cammino di lunghezza non nulla che va da un nodo a sé stesso
CAMMINO HAMILTONIANO: un cammino che passa da tutti i nodi una sola volta
CAMMINO CICLICO: se al suo interno contiene uno o più cicli; viceversa è detto SEMPLICE

RAPPRESENTAZIONE DI GRAFI

MATRICE DI ADIACENZA: dato un qualsiasi grafo la sua matrice delle adiacenze è costituita da una matrice binaria quadrata che ha come indici di righe e colonne i nomi dei vertici del grafo. Nel posto (i,j) della matrice si trova un 1 se e solo se esiste nel grafo un arco che va dal vertice i al vertice j, altrimenti si trova uno 0.

N° righe/colonne rappresenta il numero di nodi del grafo
1 → presenza di un arco tra i nodi i, j
0 →assenza di un arco tra i nodi i, j

La memoria necessaria per rappresentare un grafo tramite matrice di adiacenza è V^2 per questo motivo è conveniente utilizzare matrici di adiacenza quando il grafo non è sparso o quando è necessario determinare velocemente se esiste un arco che collega due vertici dati.
La complessità per calcolare il grado uscente e il grado entrante usando matrici di adiacenza è **lineare nel numero di nodi** (V).

LISTE DI ADIACENZA: consistono in un vettore Adj di V liste, una per ogni vertice in V contenente tutti e soli i vertici *w* tali che esista l'arco da *v* a *w*. Per ogni nodo elenco in una lista gli elementi ad esso adiacenti
La rappresentazione con liste di adiacenza è preferibile quando si devono rappresentare grafi sparsi.
Se G è un grafo orientato, la somma della lunghezza di tutte le liste è E, se G non è orientato è 2E.

VISITA DI UN GRAFO

AMPIEZZA (BFS) - Complessità V+E: analizzo a partire da un nodo tutti i vicini e ricorsivamente i vicini dei vicini prestando attenzione a non analizzare due volte lo stesso nodo.
Dato un grafo con uno specifico vertice s sorgente, la visita in ampiezza esplora sistematicamente gli archi di G per scoprire ogni vertice che sia raggiungibile da s. Essa calcola la distanza cioè il minimo numero di archi da s ad ognuno dei vertici raggiungibili e produce un albero BFS che ha s come radice e comprende tutti i nodi raggiungibili da s.
Lemma I: per ogni arco (u, v) ∈ E il cammino minimo $\partial(s,v) \leq \partial(s,u) + 1$.
Lemma II: al termine della visita, per ogni v ∈ V $d[v] \geq \partial(s,v)$ con d distanza
Infatti $d[v] = d[u] + 1 \geq \partial(s,u) + 1 \geq \partial(s,v)$
Lemma III: Durante l'esecuzione di BFS su di un grafo G = V, E la coda contiene i vertici $<v_1, \ldots, v_r>$ dove v_1 è la testa e v_r il fondo. Allora $d[v_r] \leq d[v_1] + 1$ e $d[v_i] \leq d[v_{i+1}] + 1$ per i = 1, ..., r-1

PROFONDITÀ (DFS) - Complessità V+E: gli archi vengono esplorati a partire dall'ultimo vertice scoperto v che abbia ancora degli archi non esplorati uscenti da esso. Quando tutti gli archi di v sono stati esplorati la visita torna indietro per esplorare gli archi uscenti dal vertice dal quale v era stato scoperto. Questo processo termina soltanto quando vengono scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originario e, se rimane qualche vertice non scoperto allora esso diventa il nuovo vertice sorgente e l'intero processo viene ripetuto finchè non vengono scoperti tutti i vertici del grafo.
Un grafo è ciclico sse DFS trova un grafo all'indietro.

TEOREMA DELLE PARENTESI
Per ogni u, v gli intervalli [d[u], f[u]], [d[v], f[v]] sono totalmente disgiunti o un sottointervallo

TEOREMA DEL CAMMINO BIANCO
In una foresta DFS(G) un vertice v discende da u sse al tempo d[u] esiste un cammino da u a v fatto di soli nodi bianchi.

ORDINAMENTO TOPOLOGICO (uso della visita in profondità): linearizzazione dei nodi di un grafo tale che non esistano nodi all'indietro. (Il grafo deve essere aciclico).

ALBERI DI COPERTURA MINIMI

Determinare quali e quanti archi creare per connettere n nodi di modo che il grafo sia connesso e il costo totale degli archi sia minore possibile.

TAGLIO DI UN GRAFO: Bipartizione del grafo tale che $G1 \cup G2 = G$ e $G1 \cap G2 = 0$. Preso un taglio qualsiasi è necessario che almeno un arco attraversi il grafo. Un arco è sicuro per un taglio se attraversa il taglio e ha costo minimo.

ALGORITMO DI KRUSKAL:

ALGORITMO DI PRIM: