ALGORITMI E STRUTTURE DATI

INDICE

Gli argomenti qui trattati sono frutto degli studi da me compiuti , dell'esperienza personale e delle ricerche svolte in Internet.Non mi assumo alcuna responsabilità sull'uso che ne viene fatto.

Per informazioni nydles@libero.it

TIPI DI DATO ASTRATTO

Il tipo di dato astratto **ADT** (Abstract Data Type), nasce dall'esigenza di voler rappresentare qualcosa di reale che sia lontano dal concetto di macchina.Esso identifica quel modo di agire sugli oggetti di un certo tipo invocando delle *operazioni* (chiamate **metodi**) prestabilite in modo astratto, cioè indipendente dalla rappresentazione prescelta per la loro concreta implementazione.
Ad esempio, un **grafo** è un tipo di dato astratto, poiché è un modello che può farci rappresentare qualcosa di *non particolare*.Grazie ad un *grafo*, infatti, possiamo rappresentare una rete di computer collegati tra loro, un insieme di persone legate tra di loro attraverso determinate relazioni, etc…
Quando dichiariamo gli elementi che fanno parte del nostro tipo di dato astratto, abbiamo definito il suo **dominio** (o ***insieme***); in secondo luogo si definiscono quelle **operazioni** che agiscono su tali elementi:

TIPO DI DATO = INSIEME + OPERAZIONI

I tipi di dato astratto si implementano con i linguaggi di programmazione *orientati agli oggetti* (in inglese **object oriented**).
La programmazione ad oggetti rappresenta un modo di interpretare i concetti proprio come una serie di oggetti.
Le caratteristiche di un linguaggio *object oriented* sono:

- **Data encapsulation** Con tale termine si definisce la possibilità offerta dal linguaggio di collegare strettamente gli elementi contenuti in un dato astratto con le operazioni che lo manipolano.In C++ questo procedura è detta *information hiding* (occultamento dell'informazione)
- **Le strutture di controllo** strumenti che permettono di controllare l'esecuzione dell'algoritmo.
- **L'ereditarietà** E' quel meccanismo che consente ad una tipo di dato di considerarsi erede di un altro, da cui quindi eredita tutti gli attributi ed i metodi che possono essere dunque riutilizzati.
- **La genericità** La possibilità di definire dei dati con uno o più parametri dello stesso tipo.

Una caratteristica dei moderni linguaggi di programmazione è l' **event-driven**, cioè quella tecnica che ci permette di rendere *attivi* gli oggetti, senza che la loro risposta avvenga attraverso la chiamata di qualche metodo.
Questa criterio nasce storicamente intorno agli anni '70, '80 per risolvere problemi legati alla grafica e alla gestione di finestre, mouse, multimedialità, etc…

STRUTTURE DATI E ALGORITMI DI RICERCA

Il modello di classe PILA

La pila è una struttura dati composta da elementi omogenei ordinati in una sequenza.Le operazioni di inserimento di un elemento sono effettuate ad uno stesso estremo della sequenza che viene detto cima della pila.Il tipo di logica che vi si applica è detto L.I.F.O. (last in first out): l'ultimo elemento inserito è quello che per primo viene estratto..Su una pila sono ammissibili due operazioni: PUSH (inserisci un oggetto in cima) e POP (preleva il primo oggetto dalla cima della Pila).

Costruiamo un modello di struttura dati 'Pila', utilizzando la classe `Vettore`.

```
Template <class t_info>
 class Pila{
   public:
         Vector <t_info> V;
         int top; //rappresenta la prima posizione libera (la cima)
   public:
         Pila(int d): V(d) top(1) {}
         bool Push (const t_info & T)
               {if(top<V.dim) return false;
                v[top]=T;
                top++;
                return true;}
         bool Pop (t_info& T)
               {if(top==1) return false;
                T=V[--top];
                return true;}
         bool testa (t_info & T)
               {if(top==1) return false;
                T=V[top-1];
                Return true;}
   };
```

Questa classe non ha bisogno di ditruttore(non vi si alloca memoria dinamica), né di un operatore di assegnamento o di confronto (vano bene quelli di default).

L'implementazione delle funzioni è semplice e non necessita di ulteriori commenti.La funzione `testa` assegna all'elemento passato il valore della testa della pila (se la Pila è vuota ritornerà false)

Per ottenere un oggetto di tipo Pila scriveremo:

```
Pila <int> A(12); //è una pila di interi
Pila <impiegeto> K(50); //è una pila i cui oggetti sono di tipo 'impiegato'
```

Il modello di classe CODA

La classe Coda è, come la Pila, una struttura composta da una sequenza di elementi omogenei.l'operazione di estrazione è effettuata sul primo elemento della sequenza: la coda segue una strategia F.I.F.O (first in first out): il primo elemento inserito è quello che per primo viene estratto.Anche la coda è caratterizzata dalle funzioni Push e Pop.

```
Template <class t_info >
    class Coda{
      public:
            Vector <t_info > V;
            int num_elementi;
            int in;
            int out;
      public:
            Coda(int d): V(d) {}
            bool Push (const t_info & T)
                  {if(num_elementi==V.dim) return false;
                   v[in]=T;
                   in=1+in%V.dim;
                   num_elementi++;
                   return true;}
            bool Pop (t_info & T)
                  {if(num_elementi==0) return false;;
                   T=V[out];
                   out=1+out%V.dim;
                   num_elementi--;
                   return true;}
};
```

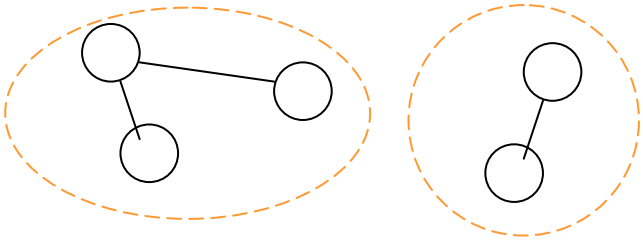
Grazie alle istruzioni `in=1+in%V.dim` e `out=1+out%V.dim` trattiamo il vettore come se avesse una struttura circolare.

GRAFO

Si definisce grafo un insieme N di ***nodi*** ad un insieme A di archi.A è una relazione binaria su N, ovvero un sottoinsieme del prodotto cartesiano NxN.
Ogni arco è caratterizzato da una coppia di nodi. Se tale coppia e' orientata, il grafo prende il nome di grafo orientato.
Se il grafo è non orientato la relazione A è simmetrica: $(n_i,n_j) \in A <=> (n_j,n_i) \in A$
Se il grafo è orientato $(n_i,n_j) \in A \neq > (n_j,n_i) \in A$
Una sequenza di nodi $(n_1, n_2, .. n_k)$ e' un cammino se $(n_i, n_{i+1}) \in A$; se il grafo è orientato (n_i,n_{i+1}) è una coppia ordinata, altrimenti sarà un insieme.
Un cammino si dice semplice se è formato da nodi tutti distinti.
Un cammino e' un ciclo se il nodo finale e' uguale al nodo iniziale.
Un grafo si dice connesso se per ogni coppia di nodi (n_i, n_j) esiste un cammino da n_i da n_j.Un grafo non connesso puo' essere suddiviso in un insieme di grafi connessi
Un arco che ripiega sullo stesso nodo si chiama autociclo.Un grafo privo di cicli si dice aciclico.
Un grafo completo è un grafo non orientato in cui ogni coppia di nodi è adiacente
Se $(n_1,n_2) \in A$ allora si dice che n_1 è adiacente ad n_2
Se esiste un cammino da n_1 ad n_2 si dice che n_2 è raggiungibile da n_1.
In genere si indica con n il numero dei nodi e con m il numero di archi.

- $m = \theta(n)$ se il grafo è sparso (pochi archi)
- $m = \theta(n^2)$ se il grafo è denso

Abbiamo detto che un grafo si dice connesso quando per ogni coppia di nodi esiste un cammino che li congiunge.Un grafo però può non essere connesso, ma può essere sempre espresso come un insieme di componenti connesse.Infatti, sia P una relazione di equivalenza definita in un sottoinsieme N* contenuto in N tale che per ogni $x,y \in N^*$ esiste un cammino tra x ed y.
Si definisce componente connessa un insieme di nodi che sia massimale rispetto alla proprietà P.



Questo è un grafo che ha due componenti connesse

NB: ricordiamo che un insieme A è massimale rispetto ad un proprietà P se A soddisfa P e se non esiste un sovrainsieme di A che soddisfa la proprietà P.
Nel caso di un grafo orientato distinguiamo componenti fortemente connesse e componenti debolmente connesse.Sono fortementi connesse quando la relazione P, definita in un sottoinsieme N* contenuto in N è tale che per ogni $x,y \in N^*$ esiste un cammino tra x ed y e un cammino tra y e x.Per trovare componenti debolmente connesse di un grafo orientato si tolgono gli orientamenti agli archi in modo da ottenere un grafo non orientato.Le componenti connesse di quest'ultimo sono le componenti debolmente connesse del grafo di partenza.
Un grafo si dice connesso anche quando è formato da un'unica componente connessa.Un grafo orientato si dice fortemnente connesso quando ogni coppia di nodi è collegata da un cammino.

Rappresentazione di un grafo

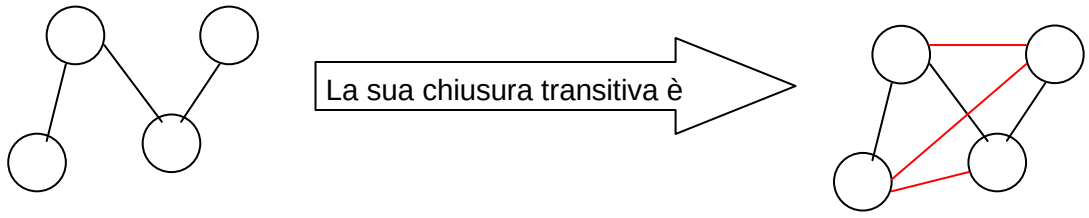
Nella memoria della macchina un grafo può essere rappresentato mediante

- **liste di adiacenza** – per ogni nodo $n \in N$ viene cortuita una lista di nodi ad esso adiacenti $n_1,n_2,\ldots,n_k$ ossia tali che $(n ,n_j) \in A$ per j=1,2,…,k.
- **matrice di adiacenza** – un grafo è rappresentato da una matrice quadrata (di booleani) di ordine n in cui $M_{i,j}=1$ se il nodo n_j è adiacente al nodo n_i , altrimenti vale 0. Se il grafo è non orientato, la matrice sarà simmetrica.

Per grafi sparsi è conveniente utilizzare una lista di adiacenza. A volte la rappresentazione con matrice di adiacenza può contribuire a contenere la complessità dell'algoritmo.

Chiusura transitiva di un grafo

Dato un grafo G=<N,A> si vuole determinare il grafo G*=<N,A*> tale che per ogni coppia di nodi $n_1,n_2 \in N$ esiste l'arco in G* che congiunge n_1 ed n_2 se e solo se esiste un cammino in G che congiunge n_1 ad n_2.
Esempio:



G=<N,A> G*=<N,A*>

Ovviamente $A \subseteq A^*$ (può essere che coincidano).In poche parole, se in G=<N,A> tra il nodo n_i ed il nodo n_j esiste un cammino, metto un arco tra n_i ed n_j.
Per realizzare la chiusura transitiva di un grafo non orientato dovremmo mettere n-1 archi per il primo nodo, n-2 per il secondo…1 per l'ultimo, in totale:

$$(n-1)+(n-2)+\ldots+1= \Sigma\, n(n-i)/2 \text{ dove } i=1,2,\ldots n-1$$

Se il grafo è rappresentato attraverso liste di adiacenza, la chiusura transitiva avrà costo $\theta\ (n^3)$, mentre con matrice di adiacenza sarà $\theta\ (nxm)$.
Abbiamo detto che un grafo non orientato si dice connesso se ammette un'unica componente connessa:questo accade se e solo se la sua chiusura transitiva è un grafo completo.

(NB: la relazione arco non è sempre transitiva, se lo è allora si realizza un cammino).

Costo delle visite in un grafo

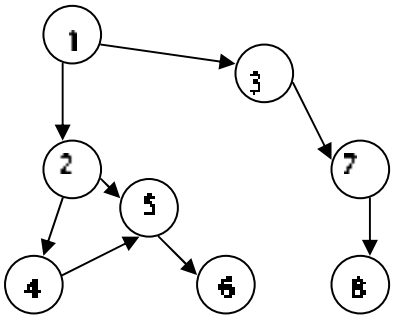
Un grafo può essere visitato in ampiezza o in profondità, ma il costo della visita non dipende dal tipo di visita, bensi dalla reppresentazione del grafo.
Con la rappresentazione a matrice dobbiamo scandire l'intera riga per ogni nodo:
$\theta (nxn)=\theta (n^2)$.Se la matrice è simmetrica sarà $\theta\ (n^2/2)= \theta\ (n^2)$ quindi non cambia nulla.
Con la rappresentazione a liste: detto K_i il numero di adiacenti la complessità sarà: $\theta\ (\Sigma\, K_i)$ ma $\Sigma\, K_i$=numero di archi=m quindi la complessità sarà: $\theta\ (m)$.

Algoritmi di visita in un grafo

Due sono gli algoritmi di visita per un grafo:

- ◊ VISITA IN AMPIEZZA (a ventaglio)
- ◊ VISITA IN PROFONDITA' (a scandaglio)

La visita in ampiezza è una visita per livelli.E'descritta tramite una funzione ricorsiva : preso un nodo, si visitano i suoi adiacenti e poi gli adiacenti degli adiacenti.Esempio:



V.A.di G= (a partire da un nodo n)

- 0 se n non ha adiacenti
- $n_1,n_2,\ldots n_k$,V.A.(n_1),…V.A(n_k) se $n_1,n_2,\ldots n_k$ sono adiacenti (non visitati)

Supponiamo di partire dal nodo 1.Visitiamo i suoi adiacenti (che sono 2 e 3) poi gli adiacenti di 2 (ovvero 4 e 5) e poi gli adiacenti di 3 (solo il 7).Poi gli adiacenti di 4 (il 5, ma si salta perchè è già stato visitato) e gli adiacenti di 5 (solo 6), poi gli adiacenti di 7 (l'ultimo,cioè l'8).
In definitiva avremo questa sequenza:

1	2	3	4	5	7	6	8

Nella **visità in profondità** si procede andando più a sinistra possibile dell'albero e poi risalendo man mano.

V.P.di G= (a partire da un nodo n)

- 0 se n non ha adiacenti
- $n_1,n_2,\ldots n_k$,V.P.(n_1),…V.P(n_k) se $n_1,n_2,\ldots n_k$ sono adiacenti (non visitati)

Nell'esempio del grafo precedente saranno visitati in ordine: 1, 2 e 4; siamo arrivati in profondità, allora risaliamo al 5 e andiamo in profondità visitando il 6.Risaliamo fino all'1 e visitiamo 3 , 7 e 8.
In questo caso gli adiacenti sono rappresentati dai figli.In questo caso la sequenza di visita è:

1	2	4	5	6	3	7	8

La visita di un grafo ci consente di *stabilire l'esistenza di cammini*, e di *trovare tali cammini* ;essa infatti restituisce l'insieme dei nodi che sono collegati ad uno dato in partenza.
Per trovare un cammino invece si costruisce un vettore di 'precedenti', dove l'indice rappresenta un nodo.Nel caso del grafo della figura precedente, si ha

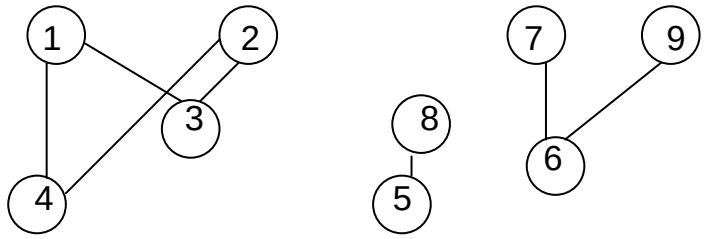
NULL	1	1	2	4	5	3	7
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Vettore di precedenti

E' facile quindi trovare ad esempio il cammino dal nodo *1* al nodo *6*: basta partire dal *6* e risalire 5, 4, 2 e arriviamo ad 1.Per cui dall1 al 6 sarà:1 -> 2 -> 4 -> 5

Determinazione delle componenti connesse in un grafo

Supponiamo di avere il seguente grafo:



Per determinare il numero delle componenti connesse partiamo dal nodo 1 e lanciamo una visita:i suoi adiacenti sono 4 e 3, che hanno 2 come adiacente, poi non ci sono più adicenti; diremo allora che 1,2,3 e 4 fanno parte della prima componente connessa. Poi lanciamo la visita dal 5 (l'unico adiacente è 8 che non ha adicaneti);allora 8 e 5 fanno parte della seconda componente connessa.Poi andiamo al nodo 6 che ha come adiacenti 7 e 9.Questi faranno parte della terza componente connessa. Costruiamo un vettore (i cui indici rappresentano i nodi) marcando gli elementi con 1°,2° ,etc… a seconda che i nodi (cioè gli indici) facciano parte della 1° ,2° ,etc…componente connessa:

1	1	1	1	2	3	3	2	3
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Relazione d'ordine parziale per un grafo

Dato un grafo orientato G=<N,A> definiamo relazione d'ordine parziale sull'insieme dei nodi N, quella quella relazione tale che $n_1 << n_2$ se esiste un cammino da n_1 ad n_2 con $n_1, n_2 \in N$.

Ordinamento topologico per un grafo

Dato un grafo orientato G=<N,A> un ordinamento topologico dei nodi è una permutazione di nodi $<k_1,k_2,\ldots k_n>$ dei nodi tale che non esiste una coppia di nodi k_i e k_j tale che i<j con $k_j << k_i$

Se un grafo (orientato) è aciclico allora esiste almeno un ordinamento topologico; infatti se prendo un grafo aciclico troverò almeno un nodo k_1 che ha grado di entrata 0, a questo nodo tolgo tutti gli archi uscenti e lo escludo dal grafo.Poi trovo un altro nodo k_2 del grafo così ottenuto che ha grado di entrata 0 e gli tolgo gli archi, considero il nuovo grafo e faccio la stessa cosa finquando non troverò più nodi che hanno grado di entrata 0 (significa che ho ottenuto un grafo che non è più aciclico).Otterrò allora una sequenza di nodi $<k_1,k_2,\ldots>$

ALBERO

Un albero e' un grafo connesso e privo di cicli.

Albero con radice

E' un albero tale che:

- esiste un nodo particolare, chiamato ***Root*** (radice).
- ogni nodo, eccetto la radice, e' connesso tramite un arco ad un altro nodo, detto ***parent*** (genitore).

Un nodo che non ha figli prende il nome di ***leaf*** (foglia).

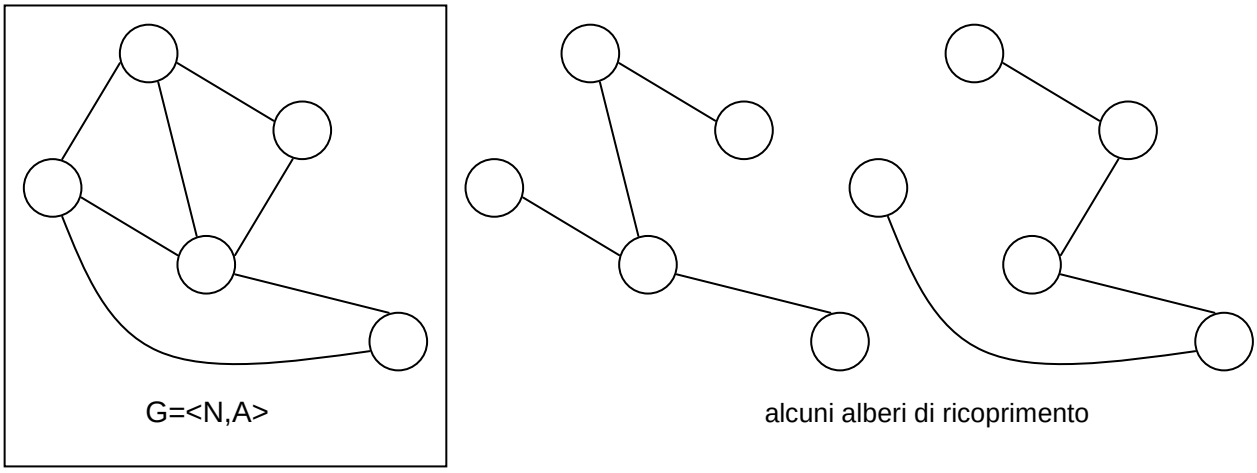
Si indica con n_F il numero di foglie: $n_F = \theta(n)$

Albero non orientato

Un albero non orientato è un un grafo non orientato, connesso e aciclico.In un albero non orientato, il numero di archi è n-1, dove n è il numero di nodi.

Albero ricoprente

Sia G=<N,A> un grafo (non orientato) e sia T=<N,A*> un sottografo connesso di G. Se T è un albero (quindi aciclico), allora si dice che T è un albero ricoprente di G.
Un grafo può avere più di un albero ricoprente.



G=<N,A>

alcuni alberi di ricoprimento

Se G è un grafo orientato, un albero ricoprente per G è un albero ricoprente per G* dove G* è un grafo che si ottiene togliendo gli orientamenti degli archi a G.
Se un grafo orientato è connesso allora esiste un albero ricoprente (vale il viceversa).

Alberi binari

Un binary tree (albero binario) puo' essere rappresentato da un insieme di nodi in cui ciascuno memorizzi l'elemento corrispondente ed i riferimenti al figlio sinistro e destro. La seguente classe ne e' un esempio:

```
class BinaryNode
      {T_info elemento;
       BinaryNode f_sx;
       BinaryNode f_dx;
       ...};
```

Un albero si dice bilanciato se la profondità tra il figlio dx ed il figlio sx è minore di 1.
Un albero è completo se percorrendo l'albero da sinistra verso destra non ci sono nodi vacanti.

Alberi binari di ricerca

Ci sono poi gli **alberi binari di ricerca**, ovvero strutture dati in cui gli elementi sono memorizzati secondo un certo ordine. La regola di ordinamento per alberi di ricerca binari specifica che per ogni nodo dell'albero tutti i nodi del sottoalbero di destra contengono elementi maggiori del nodo corrente, e tutti i nodi del sottoalbero di sinistra contengono elementi minori o uguali del nodo corrente.Questo significa che il valore del figlio sinistro è minore o uguale a quello del padre e che il valore del figlio destro è maggiore di quello del padre. Pertanto nuovi elementi dovranno essere inseriti nell'albero in maniera tale che l'ordinamento venga mantenuto, e lo stesso vale per l'eventuale rimozione di nodi.
L'algoritmo di **ricerca** verifica quindi se l'elemento cercato corrisponde alla radice e prosegue ricorsivamente nel sottoalbero di destra o di sinistra a seconda che l'elemento da cercare sia maggiore o minore dell'elemento rappresentato dalla radice, rispettivamente.
Per **inserire un elemento** in un albero binario di ricerca occorre prima trovare la posizione corretta dove inserirlo (a partire dalla radice). Cio' si fa spostandosi nell'albero seguendo il sottoalbero di sinistra o di destra a seconda che l'elemento da inserire sia minore o maggiore dell'elemento rappresentato dal nodo corrente, rispettivamente.
In termini di complessità, un inserimento di questo tipo coincide con il costo per arrivare in profondità (sia nel caso peggiore che nel caso migliore perché comunque è necessario raggiungere le foglie).Quindi se $k = \theta(\log n)$ è la profondità, allora la complessità dell'inserimento è $\theta(\log n)$.
Per **rimuovere un nodo** da un albero di ricerca occorre dapprima identificarlo, secondo l'algoritmo gia' visto di ricerca.Una volta identificato possono verificarsi i seguenti casi:

- Il nodo da rimuovere e' foglia: in questo caso la rimozione e' ovvia e la sua complessità è $\theta(1)$
- Il nodo da rimuovere ha un solo figlio: in questo caso bastera' collegare il figlio al padre del nodo da eliminare; la sua complessità è sempre $\theta(1)$. Adesso il nodo puo' essere eliminato mantenendo l'ordinamento dell'albero.
- Il nodo da rimuovere e' un nodo interno che ha due figli: in questa caso e' necessario identificare l'elemento immediatamente inferiore al nodo da rimuovere (che si otterra' scendendo nell'albero di sinistra mantenendosi sempre a destra), rimuovere quest'ultimo, e sostituirlo all'elemento da rimuovere. In questo caso la condizione di ordinamento (che i nodi del sottoalbero sinistro siano inferiori alla radice e quelli del sottoalbero di destra maggiori della radice) resta ancora mantenuta.In questo caso (pessimo)la complessità è $\theta(\log n)$

Questa è la tabella riassuntiva della complessità delle operazioni su un albero binario bilanciato:

	Caso pessimo	Caso medio	Caso migliore
Inserzione	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$
Estrazione	$\theta(\log n)$	$\theta(\log n)$	$\theta(1)$

Se fossimo stati nel caso di un albero qualsiasi, la complessità sarebbe stata:

	Caso pessimo	Caso medio	Caso migliore
Inserimento	$\theta(n)$ – ricerca in tutti i nodi	$\theta(\log n)$	$\theta(1)$ – la ricerca termina subito
Estrazione	$\theta(n)$ – ricerca in tutti i nodi	$\theta(\log n)$	$\theta(1)$ – la ricerca termina subito

Per trovare il massimo in un albero binario di ricerca basta scendere sempre a destra fino ad arrivare alle foglie (o ai nodi non pieni).

Classe ‘albero’ (con radice)

Una classe ‘albero’ sarà costituita da un puntatore a nodo (indica chi è la radice) e da una pila di nodi che utilizzeremo come iteratori:

```
class albero {
      nodo* radice;
      Pila<nodo*> i;};
```

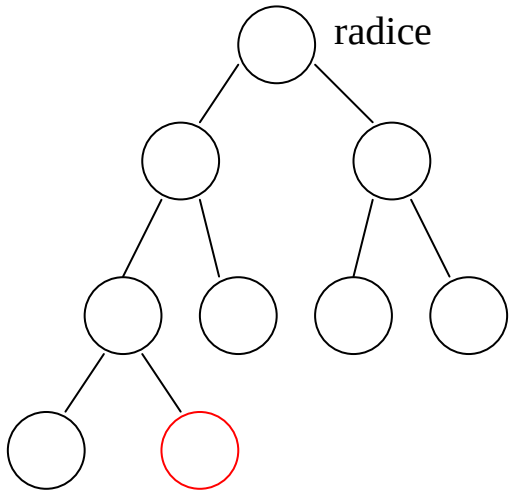
Algoritmi di visita per un albero binario

Visitare un albero significa scandirne i nodi.Vi sono tre algoritmi di visita:

- Visita anticipata: si visita prima la radice, poi il sottoalbero di sinistra con una visita anticipata ed infine quello di destra.
- Visita posticipata: Si visita posticipatamente prima il sottoalbero di sinistra, poi il sottoalbero di destra ed infine la radice.
- Visita infissa: Si visita il sottoalbero di sinistra, poi la radice ed infine il sottoalbero di destra.

HEAP

Un heap è un albero binario completo la cui funzione **val** soddisfa una particolare condizione di ordinamento.Cioè: per ogni t $\in$ al figlio(n) si ha che val(t) $\leq$ val(n).Preso un nodo dell’Heap, la funzione ‘figlio’ va dall’insieme dei nodi N in P(N) , dove P(N) è l’insieme potenza di N.
Da notare che qui non importa chi sia il figlio destro e chi il figlio sinistro, l’importante è che non abbiano un valore maggiore rispetto a quello del padre.
Anche nell’Heap abbiamo le due operazioni di inserimento ed estrazione.



Per inserire un nuovo nodo deve essere rispettata la condizione suddetta, quindi: se il nodo ha un valore maggiore di quello del padre, bisogna cambiarli di posto e così via fino a quando il nodo inserito è più piccolo di suo padre.Nel caso peggiore si arriva alla radice e la complessità è θ (logn) (abbiamo percorso tutto l’albero).Nel caso migliore non si ha nessuno scambio.Nel caso dell’ estrazione si può estrarre solo la radice, la quale verrà sostituita dall’elemento più a destra dell’ultimo livello (vedi figura), quindi si effettuano scambi finchè l’ordinamento non è rispettato.E’ necessario (nel caso pessimo) quindi effettuare scambi fino ad arrivare in profondità (cioè alle foglie) per cui la complessità vale θ (log n) sia nel caso pessimo che nel caso medio.Nel caso migliore (caso raro, quando gli elementi di un certo ramo sono tutti uguali) non si effettua alcuno scambio, quindi la complessità è θ (1).

	Caso pessimo	Caso medio	Caso migliore
Inserimento	θ (log n)	θ (1)	θ (1)
Estrazione	θ (log n)	θ (log n)	θ (1)

Se vogliamo rappresentare un heap tramite un vettore, la regola da seguire è che:

- Se un nodo n sta in posizione i, allora il suo figlio destro starà in posizione 2 x i.
- Se un nodo n sta in posizione i, allora il suo figlio sinistro starà in posizione 2 x i +1.
- Se un nodo n sta in posizione i, allora suo padre starà in posizione i / 2.

Un heap può essere impiegato per implementare code di priorità oppure per realizzare un algoritmo di ordinamento detto “heapsort” che ha una complessità θ (n logn).

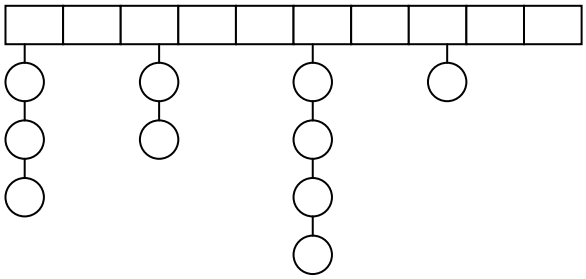
CODA DI PRIORITA’

La coda di priorità è un insieme finito di oggetti S su cui è definita una funzione P:S->T dove T è un insieme su cui è definita una relazione d’ordine totale.La logica di una coda di priorità è HPFO, (highest priority first out) cioè, l’elemento che esce è quello di priorità massima.A parità di priorità si esegue una logica FIFO.Se ad esempio abbiamo una coda in cui la priorità è rappresentata da dal tempo di inserimento e vogliamo estrarre l’elemento che è stato inserito nel tempo maggiore (quindi l’ultimo inserito) seguiremmo una logica LIFO.
La coda di priorità può essere impiegata per realizzare l’ ordinamento di un vettore procedendo in questo modo:

- Svuoto il vettore ed inserisco i suoi elementi in una coda di priorità.
- Estraggo ad uno ad uno gli elementi dalla coda di priorità e riempio il vettore.Infatti l’estrazione nella coda di priorità avviene dall’elemento massimo.

Vettore di code

Un vettore di code è un vettore i cui elementi sono delle code.



La funzione P è definita in O (insieme di oggetti) a valori in un certo dominio T finito che rappresenta l’insieme degli indici del vettore.
Il generico elemento V[i] rappresenta la coda degli elementi che hanno priorità i.

Quando viene inserito un elemento esso va inserito nella coda corrispondente alla sua priorità e quando viene richiesto di estrarre un elemento esso viene prelevato dalla coda non vuota corrispondente alla priorità più elevata.Quindi ogni operazione comporta al più una scansione del vettore di code (per l'estrazione ma non per l'inserimento) e una operazione su una coda che ha complessità costante.Pertanto l'inserimento è effettuato in tempo costante e l'estrazione in tempo $\theta(k)$, dove k è il numero di code.Se k è costante, allora la complessità di entrambe le operazioni è costante $\theta(1)$.

DIZIONARIO

Un dizionario è un insieme finito di oggetti D su cui è definita una funzione (iniettiva) key : D->T cioè, ad ogni oggetto è associata una chiave.
Le operazioni possibili sono:

- ESTRAZIONE è una funzione che riceve una chiave ed estrae l'oggetto a cui è associata quella chiave
- INSERZIONE è una funzione che riceve un oggetto e lo inserisce se non vi è nessun altro oggetto che ha la stessa chiave
- RICERCA è una funzione che riceve una chiave e trova l'oggetto che ha quella chiave associata.

Per esempio in un insieme di studenti, ciascuno di essi è individuato univocamente da una matricola, che può rappresentare la nostra key.
Esempio di una classe dizionario:

```
template <class T_info>
class dizionario {
       Lista_ord <T_info> L;
      public:
       bool inserisci (const T_info& t)
            {if(L.cerca(t)) return false; //se l'oggetto c'è già, ritorna false
             else L.inserisci(t); //altrimenti inserisci l'oggetto
             return true;}
      ...};
```

Supponiamo di avere un oggetto dizionario di studenti:

```
dizionario<studente> D;
```

e vogliamo cercare al suo interno lo studente con la marticola 67245 in questo modo:

```
D.cerca(studente(67245));
```

Affinchè si possa realizzare questo è necessario che l'oggetto ***studente*** abbia un costruttore che inizializzi solo la matricola:

```
class studente {
         int marticola;
         ...
      public:
         studente (const int& m) : matricola(m) {}
         ...
            };
```

Se il dizionario è implementato attraverso una lista non ordinata l'inserimento avrà complessità θ (1), mentre l'estrazione sarà θ (n). Nel caso di un vettore non ordinato si ha che:

	Caso peggiore	Caso medio	Caso migliore
Inserimento (con successo)	θ (n)	θ (n)	θ (1)
Inserimento (senza successo)	θ (log n)	θ (log n)	θ (1)

L'inserimento senza successo si ha quando abbiamo trovato un elementyo uguale a quello che si vuole inserire.Da notare che l'inserimento con successo, nel caso migliore, coincide con il successo della ricerca nel caso migliore.

VETTORE ASSOCIATIVO

Il vettore associativo è un *dizionario* i cui oggetti hanno associata una posizione, oltre alla chiave.
Questa funzione che assegna a ciascun oggetto una posizione è iniettiva.
Molto spesso si utilizza un vettore ausiliario di booleani che chiamiamo '**esiste'** per semplicità.Se l'i-esimo elemento di tale vettore è *true*, significa che nella posizione i-esima il nostro vettore associativo contiene un oggetto, altrimenti non lo contiene.
Quindi, se vogliamo inserire un oggetto nel vettore asscociativo dobbiamo anche passare la posizione come parametro:

```
bool inserisci (const T_info& t,int i)
```

Questa funzione ritornerà false se nel vettore esiste già un oggetto uguale a quello passato (si ricordi infatti che la funzione deve essere iniettiva).
Vediamo meglio la funzione di estrazione:

```
bool estrai (const T_info& t)
      {bool trovato=false;
       for (int i=1; ((i<=V.dim)&&(!trovato));i++)
            trovato=(T==V[i]);
       if(!trovato) return false;
       else   { esiste[i]=false; //dice al vettore 'esiste' che in quell'indice non c'è oggetto
             num_elementi--; //decrementa il numero degli elementi
             return true;}
      }
```

La classe per rappresentare un vettore associativo può essere questa:

```
template<class t_info>
class VettoreAssociativo {
      private:
         Vettore<t_info> V;
         Vettore <bool> esiste;
         int num_elementi;
         int dim;
      public:
         //dato un indice resituisce un oggetto
         t_info& operator[](const int& i)
            {assert(esiste[i]);
             return V[i];}
         //dato un oggetto resituisce l'indice
         int operator[] const t_info& t)
            {bool trovato=false;
             for(int i=0;((i<V.dim)&&(!trovato));i++)
                  trovato=(t==V[i]);
             if(!trovato) return 0;
             else return i;}
        ...};
```

Supponiamo di avere un vettore associativo di studenti:

```
VettoreAssociativo<studente> S;
```

Per sapere la posizione in cui si trova lo studente con matricola 76298, dobbiamo scrivere:

```
cout<<V[studente(76298)];
```

(ammesso sempre che quello studente sia presente nel vettore associativo)
supponiamo di avere nella classe studente una funzione pubblica **cognome()** che stampa il cognome dello studente.Se vogliamo stampare il cognome di uno studente che fa parte del vettore associativo con matricola 25635 scriveremo:

```
V[V[studente(25635)]];
```

NB: `V[studente(25635)]` restituisce un indice

EREDITARIETA', FUNZIONI VIRTUALI E POLIMORFISMO

L' **ereditarietà** è un meccanismo che consente ad una classe di ereditare ogni membro della classe-madre, e poi aggiungere e modificare le cose secondo le nuove funzionalità da realizzare.
Una classe X eredita da una classe Y (oppure *diventa sottoclasse di Y*) se ogni oggetto di classe X condivide variabili di classe, metodi e variabili oggetto della classe Y;una sottoclasse può aggiungere nuove variabili e metodi ovvero ridefinire vecchi metodi (semplicemente dichiarandoli con lo stesso nome).

```
class X:public Y {
      ...
      };
```

Il **polimorfismo** è una funzionalità del C++ grazie alla quale oggetti diversi rispondono in maniera diversa alla medesima chiamata.Esso si realizza grazie alle **funzioni virtual**; la parte di codice da associare all'invocazione di un'operazione su un dato oggetto viene determinata a tempo di esecuzione e non a tempo di compilazione.Questa funzionalità prende il nome di *late binding.*
L'applicazione di funzioni virtual prevede l'implementazione di una classe base (astratta) che contiene solo funzioni virtuali e quella di classi da essa derivate.
Esempio:

```
#include <iostream.h>

class Figura {
  public:
    virtual void disegna();
};

class Cerchio : Figura {
  public:
    void disegna() { cout << "CERCHIO" << endl;}
};

class Triangolo : Figura {
  public:
    void disegna() { cout << "TRIANGOLO" << endl;}
};

void main() {
   Figura *VettoreDiFigure[2] = { new Cerchio,
            new Triangolo}

   for(int i = 0; i < 2; i++)
       VettoreDiFigure[i].disegna();
}
```

l'output di questo programma è:
cerchio
triangolo

La classe base astratta è Figura che contiene una funzione **disegna()** comune alle due classi cerchio e triangolo, che però la 'personalizzano'. Nel ciclo *for* del *main* viene invocata la funzione **disegna**(), ma appropriatamente a seconda dell'oggetto trattato. Si noti che occorre che la classe base Figura dichiari la funzione fittizia **disegna()** tramite la parola chiave **virtual**.

TECNICHE DI PROGRAMMAZIONE

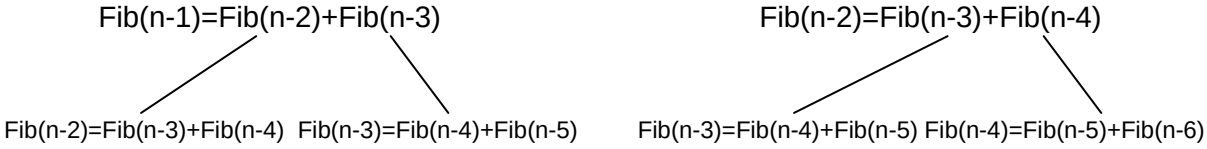
TECNICA BACKTRACKING: La tecnica Backtracking procede in questo modo: supponiamo che i nodi di un albero rappresentino l'insieme dei passi che portano alla soluzione (situata in una foglia dell'albero) di un determinato problema.La tecnica del backtracking va alla ricerca della soluzione seguendo per ogni nodo il primo figlio fino ad arrivare ad una foglia (cioè in profondità). Se questa foglia è la soluzione del problema allora la ricerca ha successo, altrimenti si risale ai livelli minori e si valuta la possibiltà di visitare un altro ramo e si procede allo stesso modo finchè si giunge alla soluzione, oppure quando tutto l'albero è stato visitato senza aver trovato la soluzione (in questo caso la ricerca fallisce).

RICERCA ESAUSTIVA: Nella ricerca esaustiva le soluzioni possibili vengono generate ad una ad una a partire da una soluzione iniziale.Man mano che si genera una soluzione essa viene sottoposta al test di ammissibiltà.La ricerca termina non appena si trova una soluzione ammissibile, oppure quando viene generata e testata l'ultima soluzione

PROGRAMMAZIONE DINAMICA:E' una tecnica di realizzazione di algoritmi che risolvono un problema utilizzano le soluzioni dei sottoproblemi.Essa parte dalla soluzione di tutti i problemi di dimensione atomica per ricomporre via via le soluzioni di tutti i sottoproblemi di dimensioni maggiori, risolvendo ogni sottoproblema solo una volta e memorizzandone la soluzione.Mentre la tecnica divide et impera è di tipo ricorsivo, quest'ultima è di tipo iterativo e in alcuni casi risulta più efficiente della prima.Un esempio è rappresentato dall'algoritmo che calcola la serie di Fibonacci:

Fib(n)=Fib(n-1)+Fib(n-2)

Dividiamo il problema in due sottoproblemi alla volta:



…e così via…

Si può perfettamente vedere che alcune funzioni (Fib(n-2) e Fib(n-3)) vengono calcolate più volte se la risoluzione del problema avviene in maniera ricorsiva (divide et impera) e così accade per le altre andando avanti.Con la programmazione dinamica invece le soluzioni di Fib(n-2) e Fib(n-3) non vengono ricalcolate se lo sono già state.

TECNICA GREEDY (golosa) : Gli algoritmi basati su tecnica golosa determinano la soluzione in N passi e ad ogni passo effettuano la scelta più conveniente al momento. Rispetto alla tecnica del backtracking, che prevede la visita in profondità dell'albero di ricerca, nella tecnica golosa si passa da un livello ad un altro dell'albero, attraverso una scelta golosa. E' utilizzata nella risoluzione di problemi di ottimizzazione:massimizzare o minimizzare una funzione obiettivo attraverso vincoli.

$$\left\{\begin{array}{c} C(x_1,\ldots,x_n) \le C_{\max} \\ 0 \le x_i \le q_i \end{array}\right\} \qquad f = \sum_{i=1}^{n} x_i * V_i(x_1,\ldots,x_n)$$

TECNICA DIVIDE ET IMPERA: Un metodo spesso usato per risolvere un problema consiste nel partizionare i dati d'ingresso in istanze di dimensioni minori, risolvere il problema su tali istanzee combinare opportunamente i risultati parziali fino ad ottenere la soluzione cercata.Qiuesta strategia è chiamata: divde et impera.

TECNICHE GREEDY

PROBLEMA DI KNAPSACK (O DELLA BISACCIA)

*D*ati:

1. Il numero di oggetti è n, ognuno con valore V_i e volume C_i
2. La disponibilità massima (volume) nella bisaccia è C

Massimizziamo la funzione obiettivo rispettando i vincoli:

$$f = \sum_{i=1}^{n} v_i * x_i$$

$$\left\{ \begin{array}{c} \sum_{i=1}^{n} c_i * x_i \leq C \\ x_i \in \{0,1\} \end{array} \right\}$$

Dove $x_i \in \{0,1\}$
0: oggetto non preso
1: oggetto preso

Le scelte vengono effettuate sulla base del valore specifico (per unità di volume) V_i/C_i e comunque l'algoritmo non garantisce soluzione ottima.

ALGORITMO DI PRIM

E' un algoritmo goloso che serve a determinare il minimo albero ricoprente di un grafo,cioè l'albero ricoprente di cammino minimo.L'algoritmo procede nel modo seguente:
Si parte da un nodo n1 e tra tutti i suoi adiacenti scegliamo quel nodo n2 per cui l'arco (n1,n2) ha peso minimo.
A questo punto scegliamo tra gli adiacenti di n_1 o di n_2 quel nodo per cui l'arco (n_1,n_3) o (n_2,n_3) ha peso minimo.In generale se i nodi già scelti sono S={n_1,n_2,….n_k} il prossimo nodo n_{k+1} è da scegliere tra gli adiacenti dei nodi di S per cui l'arco (n_i ,n_{k+1}) ha peso minimo.L'algoritmo di Prim ha una complessità θ (n^2).

ALGORITMO DI DIJKSTRA

Serve per trovare un cammino minimo a partire da un certo nodo.
Assumiamo che i pesi siano non negativi e realizziamo un algoritmo goloso simile a quello di Prim.Come nodo iniziale scegliamo un n=n_1 e tra tutti i suoi adiacenti scegliamo quel nodo n2 per cui l'arco (n_1,n_2) ha peso minimo, cioè distanza minima da n1.
A questo punto scegliamo tra gli adiacenti di n_1 o n_2 il nodo n_3 per cui il cammino da n_1 ad n_3 ha distanza minima: se n_3 è adiacente a n_1, la distanza minima è il peso dell'arco (n_1,n_3) mentre se n_3 è adiacente ad n_2 la distanza minima è la distanza da n_1 ad n_2 più il peso dell'arco (n_2,n_3)..L'algoritmo è simile a quello di Prim, con l'unica differenza di sostituire il peso minimo con la distanza minima da n_1.
La sua complessità è θ (n^2) sia nel caso di liste di adiacenza che per matrici di adiacenza.

ALGORITMO DI KRUSKAL

Anche questo, come Prim, è un algoritmo che serve a determinare il minimo albero ricoprente di un grafo L'algoritmo ordina gli archi inizialmente per peso crescente, quindi ad ogni passo considera l'arco con minor peso e valuta se la sua scelta comporti un ciclo nella soluzione corrente, cioè quando i due nodi dell'arco sono già stati inseriti nella soluzione ed esiste un cammino tra di essi per cui l'arco creerebbe un ciclo.
Per liste di adiacenza ha complessità θ (m^2), mentre con la rappresentazione a matrice di adiacenza vale θ (mxn^2).

TECNICHE DIVIDE ET IMPERA ED EQUAZIONI DI RICORRENZA

Equazioni di ricorrenza

Sia T il tempo richiesto da una tecnica ed n la dimensione dell'input:

(Equazione del primo tipo) $T(n)=aT(n/c)+n^d$

(Equazione del secondo tipo) $T(n)=aT(n-k)+n^d$

- a ➔ numero di sottoproblemi generati
- c ➔ fattore di decomposizione, denominatore di "n" per la determinazione della dimensione di ogni sottoproblema
- n^d ➔ costo della ricomposizione delle soluzioni

Merge-sort

L'equazione è del primo tipo.L'algoritmo è caratterizzato dai seguenti valori: a=2, c=2, d=1

a = c, per cui l'algoritmo ha complessità θ (n log n), infatti:

$$T(n)=2T(n/2)+n=2(2T(n/4)+n/2)+n=….=n+n \log n$$

L'individuazione del problema è costante, la ricombinazione è $\theta(n)$

Ricerca binaria

L'algoritmo è caratterizzato dai seguenti valori (per la prima equazione di ricorrenza): a=1, c=2, d=0

L'algoritmo ha complessità θ (log n), il costo di individuazione e ricombinazione è costante.

$$T(n)=T(n/2)+1=T(n/4)+1+1=T(n/8)+1+1+1=…=1+1+1+1+…+1=\log n$$
(log n volte)

Selection sort

Equazione è del secondo tipo.L'algoritmo è caratterizzato dai seguenti valori:

k=1, a=1, d=1

$$T(n)=T(n-1)+n= T(n-2)+n-1+1=T(n-3)+(n-2)+(n-1)+1==…=1+2+…(n-1)+n=\sum i=n\ (n+1)/2$$

Quindi è $\theta(n^2)$.

La ricombinazione è costante.L'individuazione vale $\theta(n)$.

Serie di Fibonacci

Ricordiamo che : Fib(n)=Fib(n-1)+Fib(n-2)
(l'equazione di ricorrenza è del secondo tipo):

$$T(n)=T(n-1)+T(n-2)+1=T(n-2)+T(n-3)+1+T(n-3)+T(n-4)+1+1=…=2^{n-1}$$

La sua complessità sarà quindi θ (2^n).

La tecnica Divide et Impera non è la più efficiente per risolvere questo problema.Essa infatti è più conveniente quando il numero di sottoproblemi, rilevanti per la soluzione, è minimo.

Con la tecnica di programmazione dinamica si verifica che la complessità per il calcolo della serie di Fibonacci vale $\theta(n)$.

LA COMPLESSITA' ALGORITMICA

La complessità di un algoritmo, intesa come memoria e tempo necessari per l'esecuzione, dipende sia dalla macchina utilizzata che dalla dimensione del problema.La dipendenza dalle caratteristche della macchina influisce al più per una costante di proporzionalità e può essere considerata ininfluente per la valutazione dell'efficienza dell'algoritmo.

Nel valutare il tempo di calcolo di una procedura, è in genere assai difficile quantificare con esattezza il numero di operazioni elementari eseguite. Questa difficoltà viene aggirata valutando il numero di operazioni in ordine di grandezza, cioè esprimendolo come limite di una funzione f(n) al tendere all'infinito della dimensione n, trascurando le costanti moltiplicative ed additive. Si parla così di complessità asintotica.

NOTAZIONE O
A tal fine, si usa la notazione 'O' definita così:
data una funzione f(n), indichiamo con O(f(n)) l'insieme di tutte le funzioni g tali che

$\exists^{no}$ c,m$\in$ N : $\forall$ n $\geq$ m si verifica che g(n) $\leq$ c f(n)

Si dice anche che g(n) $\in$ O(f(n)).
La notazione O fornisce una delimitazione superiore alla complessità, cioè fornisce una valutazione approssimata per eccesso In particolare, si dice che f(n) è un upper bound per g(n).
A questo punto, può sorgere una domanda legittima: poichè si trascurano le costanti, è sempre lecito preferire, tra due algoritmi, quello avente complessità di ordine più basso?Questo dipende dalla dimenzione dell'input.Ad esempio, per n<50 è più veloce un algoritmo che abbia complessità $O(n^3)$ di uno con complessità $O(n^2)$.
Ricapitolando, per stabilire la complessità di un algoritmo, se ne studia l'ordine di grandezza O del tempo di calcolo, inteso come numero di operazioni elementari eseguite nel caso pessimo (o medio) in funzione della dimensione n dei dati di ingresso.

NOTAZIONE Ω
Per risolvere un problema, si cerca di scoprire algoritmi di complessità sempre più bassa.Per stabilire quale sia il limite inferiore (al di sotto del quale non è possibile trovare algoritmi più efficienti) si introduce la notazione Omega:
data una funzione f(n), indichiamo con Ω (f(n)) l'insieme di tutte le funzioni g tali che

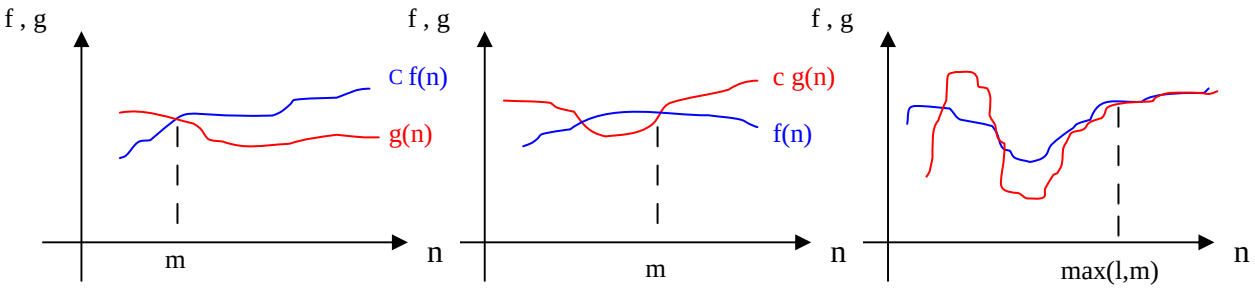
$\exists^{no}$ c,m$\in$ N : $\forall$ n $\geq$ m si verifica che f(n) $\leq$ c g(n)

Si dice anche che g(n) $\in$ Ω(f(n))
La notazione Ω fornisce una delimitazione inferiore alla complessità di un algoritmo. In particolare, si dice che f(n) è un lower bound per g(n).

NOTAZIONE θ
Una valutazione esatta della complessità di un algoritmo è l'intersezione delle due segnature,cioè la notazione θ

$$\theta (f(n))=O(f(n)) \cap \Omega (f(n))$$

θ (f(n)) è l'insieme delle funzioni g tali che esistono c,k,m,l tali che c g(n) $\geq$ f(n) e g(n) $\leq$ k f(n) per ogni n>max(l,m)



Notazione O Notazione Omega Notazione Teta

PROPRIETA'

Si ha la seguente proprietà: $g \in \theta(f) \iff f \in \theta(g)$

Nella notaziona O, si ha un valore di n a partire dal quale g(n) non supera f(n) (a meno di una costante).
La stessa cosa vale per la notazione Omega: si individua un valore a partire dal quale g(n) è sempre maggiore di f(n).
Nella notazione Teta, a partire da un certo valore, g(n) ed f(n) vanno insieme.

Alcuni esempi:

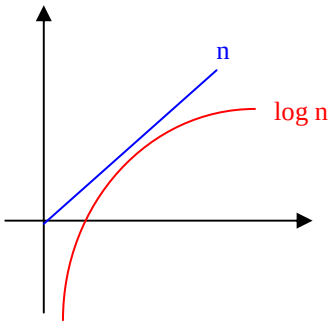
log n = O(n) (quello in figura)

n log n = $O(n^2)$

$n^{100} = O(2^n)$

$n^{100} = O(n^{105})$

$n^{100} \neq O(n^{40})$ (la definizione di O non è verificata)



L' algorimto del **boubble sort**, nel caso peggiore (tutti gli elemnti sono fuori posto) farà n scambi alla prima scansione, n-1 scambi alla seconda, n-3 alla terza e così via…in tutto:

$$\sum i = n(n+1)/2$$

In termini asintotici (quindi trascurando anche le costanti moltiplicative) avremo una complessotà di $O(n^2)$.
Nel caso migliore (il vettore è ordinato), alla prima scansione (da 1 ad n) non effettuiamo nessun scambio:l' algoritmo termina.La sua complessità sarà O(n).

Nel caso del **prodotto tra matrici** (supponiamo di dimensione n), il costo sarà n^3.Infatti per ogni elemento si calcola il un prodotto scalare (nxn operazioni).Essendo gli elementi n, le operazioni saranno n x n x n.

Complessità di un algoritmo che calcola la **serie di Fibonacci:**
Ricordiamo che:
Fib(0)=1, Fib(1)=1, Fib(n)=Fib(n-1)+Fib(n-2)
La funzione che calcola la serie di Fibonacci, riceve valore n in igresso e calcola n-1 ed n-2 e così via fino ad arrivare a zero.In totale, le operazioni saranno:

$$\sum 2^i = 2^{n+1} - 1 \quad \text{con } i=0,1,2,\ldots,n$$

La complessità varrà $\theta(2^n)$.Questa è una stima fatta in funzione del valore di n.
In funzione della dimenzione dell'input, la complessità sarà: $\theta(2^{(2^n)})$

Nel caso del **calcolo del fattoriale**, la complessità in funzione del valore di n sarà $\theta(n)$ in quanto n sono le operazioni da effettuare..In funzione della dimensione dei dati in input, la complessità sarà invece $\theta(2^k)$ indicando k la dimensione dei dati.

LOWER BOUND E UPPER BOUND

Valutare la complessità di un problema in termini di **lower bound** significa stabilire qual è la quantità di risorsa minima necessaria (ma non sufficiente) per realizzare un algoritmo che lo risolva.
Indichiamo con A l'insieme degli algoritmi che possono risolvere il problema P e con $\mathbf{C_p}$ la complessità di P
Se $\forall \mathbf{A} \in \mathbf{A}$ $\mathbf{C_p = \Omega(f(n))}$ allora il problema **P ha un lower bound f(n)** , cioè il costo dell'algoritmo è inferiormente limitato da f(n).
In termini di **upper bound** significa invece stabilire che è possibile risolvere l'algortimo con ***meno di tanto:***
Se $\forall \mathbf{A} \in \mathbf{A}$ $\mathbf{C_p = O(f(n))}$ allora il problema **P ha un upper bound f(n)**.
Se *il lower bound* coincide con l' *upper bound* , allora abbiamo trovato la quantità di risorsa necessaria e sufficiente.

PROBLEMI TRATTABILI ED NP-COMPLETI

Tutti quei problemi per cui esiste un algoritmo di complessità di tipo polinomiale, si dicono "trattabili".Sono considerati trattabili, ad es. i problemi i cui algoritmi hanno complessita n^k.Quelli che non sono di tipo polinomiale si dicono "NP-completi" come ad es.quegli algoritmi di tipo esponenziale (NP: Non Polinomiale)

Tabella riassuntiva delle complessità

Albero qualsiasi	caso pessimo	caso medio	caso migliore
Inserimento	θ (n)	θ (log n)	θ (1)
Eliminazione	θ (n)	θ (log n)	θ (1)

Albero bin. Bilanciato	caso pessimo	caso medio	caso migliore
Inserimento	θ (log n)	θ (log n)	θ (log n)
Eliminazione	θ (log n)	θ (log n)	θ (1)

Lista	caso pessimo	caso medio	caso migliore
Inserimento	θ (1)	θ (1)	θ (1)
Estrazione	θ (n)	θ (n)	θ (n)

Lista ordinata	caso pessimo	caso medio	caso migliore
Inserimento	θ (n)	θ (n)	θ (1)
Estrazione	θ (1)	θ (1)	θ (1)

Vettore	caso pessimo	caso medio	caso migliore
Inserimento	θ (1)	θ (1)	θ (1)
Estrazione	θ (n)	θ (n)	θ (n)

Vettore ordinato	caso pessimo	caso medio	caso migliore
inserimento	θ (n)	θ (n)	θ (1)
Estrazione	θ (1)	θ (1)	θ (1)

Heap	caso pessimo	caso medio	caso migliore
Inserimento	θ (log n)	θ (1)	θ (1)
Estrazione	θ (log n)	θ (log n)	θ (1)