Introduzione

Nel mondo reale gli algoritmi intervengono, in modo più o meno nascosto, e in svariati contesti, alcuni esempi di natura informatica potrebbero interessare l'internet moderna, i DBMS per le basi di dati, i motori di ricerca per il web, l'analisi di documenti in formato digitale etc…
Nel mondo di Internet ad esempio il protocollo di routing ha come obiettivo la ricerca di un buon cammino fra la sorgente (che solitamente fa richiesta di una determinata pagina) e la destinazione (ad esempio un server). Usando un astrazione adottata nel campo della telematica e basata su nodi (che rappresentano i router) e su archi (che rappresentano i link fisici) l'algoritmo di routing detto anche di instradamento deve determinare un cammino che spesso viene detto "buono" e che nella maggior parte dei casi corrisponde con quello di "costo minimo" (ad ogni link fisico è associato un costo che tiene conto del livello di saturazione del link e della distanza fisica fra vari router).
Nelle basi di dati vengono formulate spesso delle interrogazioni (query) che hanno come obiettivo la restituzione di un dato che è contenuto nella base di dati. L'algoritmo in questo caso, ed anche la struttura dati, ha il compito di agevolare l'immissione dei dati e di fornire una risposta che sia possibilmente rapida diminuendo i tempi di attesa per l'utente. Gli esempi appena citati possono quindi suggerire l'importanza degli algoritmi e delle strutture che nel corso degli anni si sono sempre affinate per supportare con più efficienza gli algoritmi implementati. Nel corso di queste note avremo perciò come obiettivo i seguenti punti:

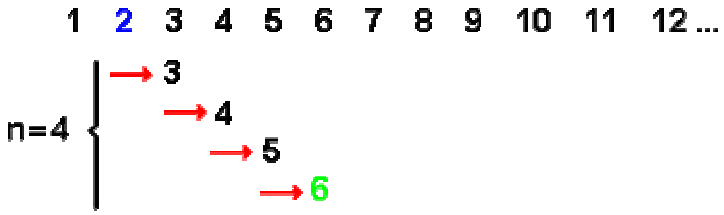
- utilità e progettazione di algoritmi e strutture dati;
- valutare l'efficienza delle strutture dati e degli algoritmi;
- scelta di strutture dati idonee alla soluzioni di problemi;
- implementazione di algoritmi per strutture dati;

1.1 Un po' di storia: la macchina universale di Turing

Nel 1854, il matematico britannico George Boole (1815 - 1864), elaborò una matematica algebrica che da lui prese il nome. Nell'algebra di Boole le procedure di calcolo si possono effettuare grazie a operatori matematici (AND, OR, NOT, ecc.) di natura logica. L'algebra di Boole entrò prepotentemente alla ribalta nel 1936, quando il matematico britannico Alan Mathison Turing (1912-1954), immaginò una "macchina" o "automa" (esistente unicamente a livello teorico) con la quale dimostrò formalmente la possibilità di eseguire qualsiasi algoritmo: una procedura di calcolo o, più in generale, la sequenza delle operazioni necessarie per risolvere un problema in un numero finito di operazioni. In tal modo, Turing aprì la strada al campo di quelle ricerche informatiche che prendono il nome di intelligenza artificiale (esiste anche un test di Turing), e l'algebra di Boole si rivelò di fondamentale importanza nella progettazione degli odierni computer.

1.2 Ricerca di un algoritmo

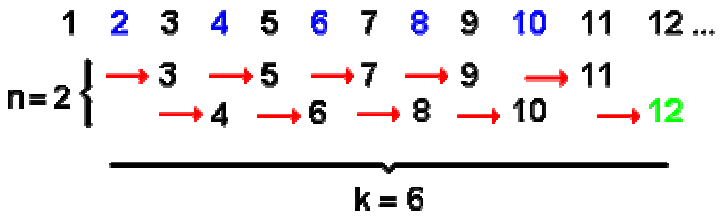
Per Alan Turing, la prima questione da risolvere consisteva nel precisare le azione elementari che compiamo quando eseguiamo un calcolo. In effetti, una semplice operazione di somma viene appresa facilmente fin dalle scuole elementari: è un'operazione che effettuiamo meccanicamente ogni giorno, quando per esempio acquistiamo due oggetti dallo stesso fornitore, o quando contiamo il denaro che abbiamo in tasca. L'addizione è dunque un'operazione banale... ma sappiamo precisare i passaggi necessari per effettuarla, sappiamo cioè definirne l'algoritmo?
"Programmi" per risolvere manualmente problemi numerici sono noti fin dal 1800 a.C., quando i matematici babilonesi del tempo di Hammurabi precisarono le regole per risolvere alcuni tipi di equazioni. Le regole consistevano in procedimenti dettagliati passo dopo passo applicati dettagliatamente a particolari esempi numerici. In particolare, il termine "algoritmo" si rintraccia dall'ultima parte del nome del matematico persiano Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî, il cui testo di aritmetica esercitò una notevole influenza per molti secoli. Per far questo, supponiamo di eseguire una somma di due numeri (naturali), per fissare le idee, 2 + 4:



Visualizzando una sequenza di numeri naturali ordinati in successione, consideriamo il numero 2 e ripetiamo 4 volte l'operazione di passaggio al numero successivo: da 2 a 3, da 3 a 4, da 4 a 5, da 5 a 6. Il numero 6 è il risultato richiesto. Per ottenere il risultato della somma proposta, dobbiamo quindi seguire questi passaggi:

- individuare un numero (quello di partenza);
- passare al numero immediatamente successivo;
- controllare se si è raggiunto il numero di ripetizioni previste (n = 4, in questo caso);
- ripetere un'operazione (quella di passaggio al numero successivo)

Questo è dunque l'algoritmo della somma di due numeri. La sequenza di operazioni esaminata, si presta anche all'esecuzione di moltiplicazioni. Per esempio, 2 x 6 equivale ad addizionare 6 volte il numero 2 (o 2 volte il numero 6):



Per ottenere il risultato della moltiplicazione proposta, dobbiamo seguire questi passaggi:

- individuare un numero (quello di partenza);
- passare al numero immediatamente successivo;
- controllare se si è raggiunto il numero di ripetizioni previste per il primo ciclo (n = 2, in questo caso);
- ripetere un'operazione (quella di passaggio al numero successivo);
- controllare se si è raggiunto il numero di ripetizioni previste per il secondo ciclo (k = 6, in questo caso)

Come si vede, si è aggiunto un secondo controllo. Con analogo procedimento, è possibile anche l'elevazione a potenza. Per esempio, 2^3 = 2 x 2 ripetuta 3 volte. In questo caso, si dovrà aggiungere un terzo controllo. Si potrebbe vedere (ma la questione è irrilevante per questa discussione) che il procedimento utilizzato per addizioni, moltiplicazioni ed elevazione a potenza, vale anche per le operazioni inverse: sottrazione, divisione (una successione di sottrazioni: 17/5 = 17 - 5 - 5 - 5 = 3 con il riporto di 2), estrazione di radice etc..

1.3 La macchina di Turing

Dimostrata l'esistenza di algoritmi per effettuare le operazione matematiche fondamentali (gli algoritmi nascono dall'esigenza di dare una definizione formale ai teoremi), la seconda questione da

risolvere consisteva nel precisare l'occorrente per sviluppare la sequenza di operazioni previste, naturalmente non ci riferiamo a carta e penna, ma a qualcosa di più raffinato.
Ricordando che la prima calcolatrice, la Pascalina, fu inventata nel 1642 da Blaise Pascal, per comprendere l'importanza della macchina o automa di Turing, dobbiamo fare una distinzione tra le calcolatrici ed i moderni calcolatori. La linea di separazione può essere individuata nel fatto che le calcolatrici riescono ad eseguire un certo numero di operazioni ma non possono essere programmate: manca la possibilità di specificare alla macchina processi di calcolo articolati e complessi, definiti in base ad opportune sequenze di comandi che possano essere eseguiti in modo completamente automatico. L'idea fondamentale alla base del calcolatore programmabile (elaboratore o computer), è che qualsiasi tipo di computazione consiste nella manipolazione di simboli (ne bastano solo 2 come nei calcolatori digitali) seguendo un'insieme di regole (algoritmo). Con questo modello nasce l'idea di "calcolatore universale". Il metodo di programmazione di Turing, essenzialmente descriveva una macchina che utilizzava alcuni semplici istruzioni. Fare sviluppare ad un computer un compito particolare, era soltanto una questione di dividere il problema in una serie di problemi più semplici (un'operazione identica al processo affrontato dagli odierni programmatori) risolvibili appunto con semplici operazioni.
Una macchina o automa di Turing è definita da un insieme di regole che definiscono il comportamento della macchina su un nastro di Input-Output (lettura e scrittura). Il nastro può essere immaginato come una sottile striscia di carta divisa in quadratini dette celle e di lunghezza adeguata per eseguire qualsiasi algoritmo. Ogni cella contiene un simbolo oppure è vuota. L'automa utilizza una testina che si sposta lungo il nastro leggendo, scrivendo oppure cancellando simboli nelle celle del nastro. La macchina analizza il nastro, una cella alla volta, iniziando dalla cella che contiene il simbolo più a sinistra nel nastro. Da quanto esposto, una macchina di Turing può essere immaginata come una sorta di registratore a nastro con una testina di lettura, scrittura e cancellazione. La testina possiede un indicatore che determina, per ogni passaggio di calcolo, lo stato specifico in cui la macchina si trova mentre sta leggendo il simbolo che corrisponde alla particolare cella del nastro sul quale è posizionata.

Definita la macchina, devono essere specificate le regole – l'equivalente del moderno *software* – che indichino che cosa fare in corrispondenza della combinazione di stati e di simboli letti sul nastro. In generale, un automa di Turing deve poter effettuare le seguenti operazioni:

- conservare la memoria del suo stato interno;
- leggere il simbolo scritto in una cella;
- sovrascrivere o cancellare il simbolo scritto in una cella;
- scorrere il nastro cella dopo cella, verso destra o verso sinistra;
- non fare alcuna operazione.

Per esempio, la seguente istruzione contenuta in tabella, impone che se l'indicatore di stato è in posizione 1 e la testina sta leggendo il simbolo *A*, l'istruzione consiste nel cancellare il simbolo, far avanzare la testina e cambiare lo stato della macchina a 2.

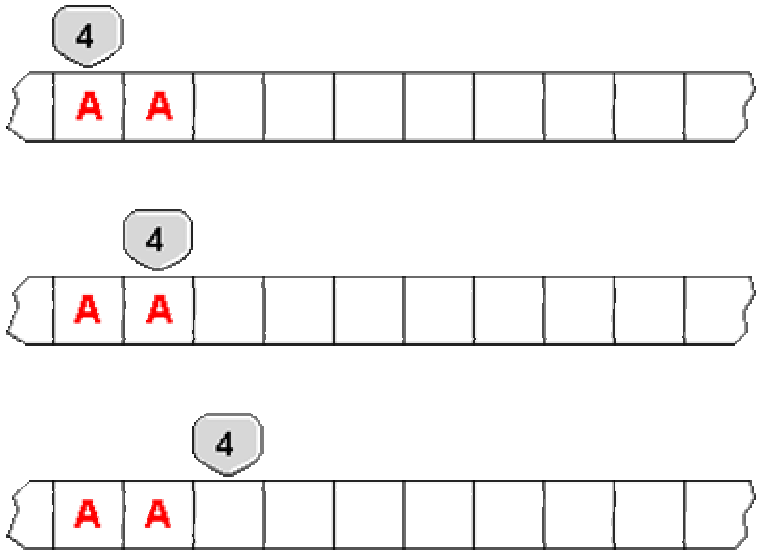
stato iniziale	legge	scrive	azione	stato finale
1	A	cancella	>	2

Per dare ragione dell'universalità della macchina di Turing, occorrerebbe mostrare come effettuare le 4 operazioni fondamentali (addizione, sottrazione, moltiplicazione e divisione) in quanto tutte le altre sono riducibili a queste. Tuttavia, ci limiteremo ad esaminare solo l'operazione di somma perché la programmazione di una macchina di Turing è laboriosa in quanto richiede un linguaggio di basso livello, "comprensibile" dalla macchina (oggi gli elaboratori possono utilizzare linguaggi di alto livello che sono più vicini alla logica umana).

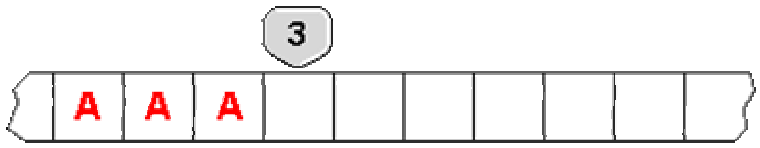
La tabella seguente contiene le istruzioni per addizionare un certo numero di "A" (per esempio, 4) ad una sequenza di "A" (per esempio, 2). E' da notare che le istruzioni non vengono necessariamente eseguite in maniera sequenziale, ovverosia una dopo l'altra, bensì in base allo stato dell'automa.

stato iniziale	legge	scrive	azione	stato finale
4	A	A	>	4
4	-	A	>	3
3	-	A	>	2
2	-	A	>	1
1	-	A	STOP	0

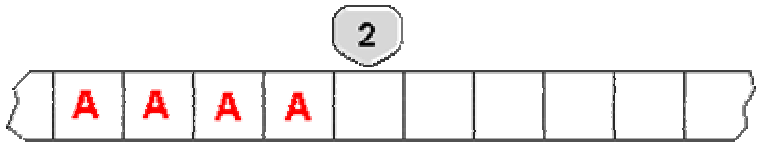
Per esempio, nel caso della somma proposta (addizionare 4 volte "A"), la prima istruzione prevede che se l'automa si trova nello stato iniziale 4 e legge la lettera "A", deve riscrivere la lettera, poi l'automa si sposta di una casella verso destra e mantiene il suo stato interno (4) fintanto che trova una A da leggere.



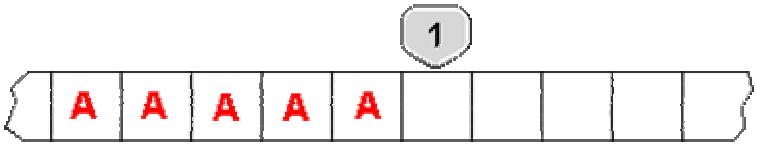
A questo punto, poiché l'automa (nello stato 4) non legge una "A" ma trova una casella vuota, deve scrivere nella casella sottostante una lettera A, spostarsi a destra e passare allo stato 3:



l'automa nello stato 3 non legge una "A" ma trova una casella vuota, quindi deve scrivere nella casella sottostante una lettera A, spostarsi a destra e passare allo stato 2:



l'automa nello stato 2 non legge una "A" ma trova una casella vuota, quindi deve scrivere nella casella sottostante una lettera A, spostarsi a destra e passare allo stato 1:



Infine, quando lo stato interno è 1, l'automa scrive una A, assume lo stato n = 0 e si ferma.
Il programma visto può essere realizzato con una sequenza di istruzioni dinamiche: le istruzioni non sono predefinite, ma vengono via via specificate. In questo modo, un programma di addizione è valido per qualsiasi combinazione di addendi (nel caso si vogliano addizionare quattro "A", si imporrà n = 4). E' da notare che l'istruzione n = n - 1 non è un'equazione algebrica, ma implica che lo stato n deve diventare n - 1: se n = 4, diventa n = 4 - 1 = 3.

stato iniziale	legge	scrive	azione	stato finale
N	A	A	>	N
N	-	A		N=N-1
1	-	A	STOP	0

Con la macchina di Turing (che, per quanto possa sembrare strano, riassume la struttura funzionale di un computer) è possibile risolvere anche problemi non numerici; infatti basta associare ai simboli un significato alfabetico o alfanumerico. Turing non costruì materialmente la sua "macchina universale". Infatti, per quanto tecnicamente realizzabile, il meccanismo (elettromeccanico) sarebbe stato lento e poco affidabile per impieghi pratici. D'altra parte, data la sua semplicità, una macchina di Turing può essere simulata con carta e penna: se un operatore umano può eseguire le istruzioni senza incertezze e ambiguità, allora la procedura è automatizzabile. Nel 1943, viene realizzato in Pennsylvania l'E.N.I.A.C .(Electronic Numerical Integrator And Calculator): il primo elaboratore senza parti meccaniche in movimento (all'epoca i calcolatori erano elettromeccanici). Il giorno della sua presentazione, l'ENIAC (col sistema della scheda perforata) moltiplicò il numero 97.367 per se stesso 5.000 volte, completando l'operazione in meno di un secondo. Con l'ENIAC, che funzionò fino al 1955, nasce l'era informatica vera e con essa anche il termine BIT (Binary Digit = Cifra Binaria).
Fonte: http://www.nemesi.net/turing.htm

1.4 Il test di Turing
Nel 1949, il famoso neurochirurgo Sir Geoffrey Jefferson (1886-1961), nel suo scritto “No Mind for Mechanical Man” (Nessuna mente per l'uomo meccanico), esponeva una serrata critica ad un precedente articolo che riguardava la macchina di Turing universale.
«Fino a quando una macchina non potrà scrivere un sonetto o comporre un concerto suggeriti da emozioni realmente provati, e non per una scelta casuale di simboli, non potremo ammettere che una macchina eguagli il cervello umano; cioè che non solo scriva queste cose, ma che sappia di averle scritte. E' certo che nessun meccanismo potrebbe provare piacere (e neppure manifestarlo artificialmente, un facile espediente) verso i propri successi e angosce quando gli saltano le valvole, né animarsi davanti alle lusinghe, o rattristarsi per i propri errori, o essere affascinato dal sesso, o incollerirsi o deprimersi quando non può ottenere ciò che desidera».
Queste argomentazioni, apparentemente logiche, erano in realtà facilmente confutabili, e la traccia per farlo era stata indirettamente suggerita nel MacBeth di William Shakespeare...

(SCENA QUARTA)
Duncan (Re di Scozia): Non c'è arte per leggere nella faccia la costituzione della mente...

Ecco, a dispetto di quanto sosteneva Jafferson, le "emozioni" possono essere manifestate artificialmente, e non c'è alcun modo per distinguerle da quelle reali... tralasciando il fatto che vi sono persone che non si compiacciono davanti
alle lusinghe, né si rattristano per i propri errori, né soggiacciono ai piaceri del sesso ed ai desideri materiali.
Alan Turing, in un articolo del 1950 Computing Machinery and Intelligence (Macchine calcolatrici e intelligenza), propose un criterio - oggi noto come "test di Turing" - per determinare se un computer fosse in grado di pensare. Turing era convinto che la sua macchina potesse effettuare qualsiasi operazione logica e, programmata con la necessaria abilità, entro il duemila avrebbe potuto simulare l'intelligenza umana.
Ecco come si dovrebbe svolgere il test di controllo. Una persona si trova davanti ad un terminale e con la tastiera scrive delle domande e riceve delle risposte. Dall'altro capo del terminale ci sono una macchina ed un operatore umano che forniscono alternativamente le risposte alle domande. Se la persona non è in grado di distinguere quando sta interloquendo con una macchina e quando con un operatore umano, allora la macchina è intelligente.
Finora nessun programma ha superato il test di Turing. Il primo ad aver ottenuto un certo successo, è Eliza, un programma scritto nel 1966 da Joseph Weizenbaum. Eliza è una psicoterapeuta che simula una conversazione tra lei (il medico), e voi (il paziente).

paziente: gli uomini sono tutti uguali
computer: in che senso
paziente: ci fregano sempre in qualche modo
computer: puoi fare un esempio specifico
paziente: un mio amico mi ha fatto venire qui
computer: un tuo amico ti ha fatto venire qui
paziente: dice che sono quasi sempre depressa
computer: mi dispiace sentire che sei depressa
paziente: è vero sono infelice
computer: credi che venire qui ti aiuterà a non essere infelice
paziente: ho bisogno di aiuto, questo è sicuro
computer: cosa intendi come aiuto
paziente: forse potrei riuscire ad andare d'accordo con mia madre
computer: parlami di tua madre

Una conversazione di questo tipo porta a domandarsi se si può realmente ritenere se è il computer che pensa veramente, o piuttosto sono le persone che possono sostenere una conversazione senza bisogno di pensare!
In effetti, il programma Eliza non era molto convincente; tuttavia, ai primordi dei computer domestici molte persone erano convinte che un computer fosse un "cervello" elettronico e quindi non facevano molto caso alla piega bizzarra che ben presto delineava la "seduta". D'altra parte, il test di Turing non prevedeva l'ingenuità della persona incaricata di saggiare la macchina: doveva essere un operatore esperto. Dopo Eliza sono stati realizzati molti programmi per simulare l'intelligenza; sebbene alcuni siano progettati per argomenti ben definiti (per es. teatro di Shakespeare), nessuno è stato in grado di ingannare un giudice esperto.
Fonte: http://www.nemesi.net/turingtest.htm

2.1 La nozione di algoritmo

Una definizione di algoritmo è la seguente: un algoritmo è una procedura passo per passo grazie alla quale un'operazione può essere svolta senza alcun esercizio di intelligenza e quindi, per esempio, da

una macchina. Questa definizione pone quindi l'algoritmo come la chiave risolutiva di un problema, il problema deve ovviamente essere formalizzato ed anche ben appreso da chi lo vuole risolvere.
Un algoritmo per funzionare deve richiedere un numero finito di passi, è questo un requisito fondamentale di un algoritmo. L'utente che si avvale di un algoritmo si attende una risposta in termini computazionale e dei dati immessi, è quindi ragionevole aspettarsi da un algoritmo questa proprietà la quale deve valere su qualsiasi istanza del problema (e quindi non solo su un numero ristretto di casi che si presentano). Per esprimere un algoritmo si ricorre ad un linguaggio di programmazione che deve essere adeguato a descrivere il procedimento, preciso e non ambiguo, comprensibile all'esecutore.
Sono esempi di algoritmo una ricetta di cucina, istruzione per il montaggio di un mobile dell'IKEA, le regole aritmetiche per effettuare una somma di due numeri, le procedure di rilascio di un passaporto. Come si può vedere quindi il concetto di algoritmo non è principalmente legato al mondo dell'informatica ma esprime un concetto molto più generale e può essere applicato in molti settori, ogni cosa che richiede una soluzione prevede un algoritmo.

3.1 La ricorsione

La ricorsione (recursion) è una tecnica di programmazione molto potente che sfrutta l'idea di suddividere un problema da risolvere in sottoproblemi simili a quello originale, ma più semplici. Un algoritmo ricorsivo per la risoluzione di un dato problema deve essere definito nel modo seguente:

- prima si definisce come risolvere dei problemi analoghi a quello di partenza, ma che hanno "dimensione piccola" e possono essere risolti in maniera estremamente semplice (detti casi base);
- poi si definisce come ottenere la soluzione del problema di partenza combinando la soluzione di uno o più problemi analoghi, ma di "dimensione inferiore";

Un classico esempio di descrizione ricorsiva è la definizione del fattoriale di un numero intero. Immaginiamo di dover calcolare il fattoriale di un numero n:

$$n! = n * (n - 1)* ... * 3 * 2 * 1$$

Per convenzione 0! = 1. Inoltre, il fattoriale non è definito per i numeri negativi. Come possiamo scrivere un metodo che calcoli la funzione fattoriale? Osserviamo che:

$$n! = n * (n-1) * ... * 2 * 1 = n * (n-1)!$$

Quindi la versione "ricorsiva" della definizione di fattoriale è:

$$0! = 1 \quad \text{(caso base)}$$
$$n! = n * (n-1)! \quad \text{(se } n > 0\text{)}$$

3.2 Ricorsione diretta

Un metodo (procedura/funzione) si dice ricorsivo quando all'interno della propria definizione compare una chiamata direttamente al metodo stesso. Questa forma di ricorsione si chiama ricorsione diretta. Un esempio di ricorsione diretta è la seguente realizzazione del fattoriale di un numero intero:

```
Public class MyRecursiveMethods
{
      // altri metodi
      …
      Public static int factorial(int n)
      {
      Int result;
      If (n<0)
            result = -1;                  // situazione anomala
      else if (n==0)
            result = 1;             // caso base
      else
            result = n*factorial(n-1);    // ricorsione
      return result;
      }
      …
      // altri metodi
}
```

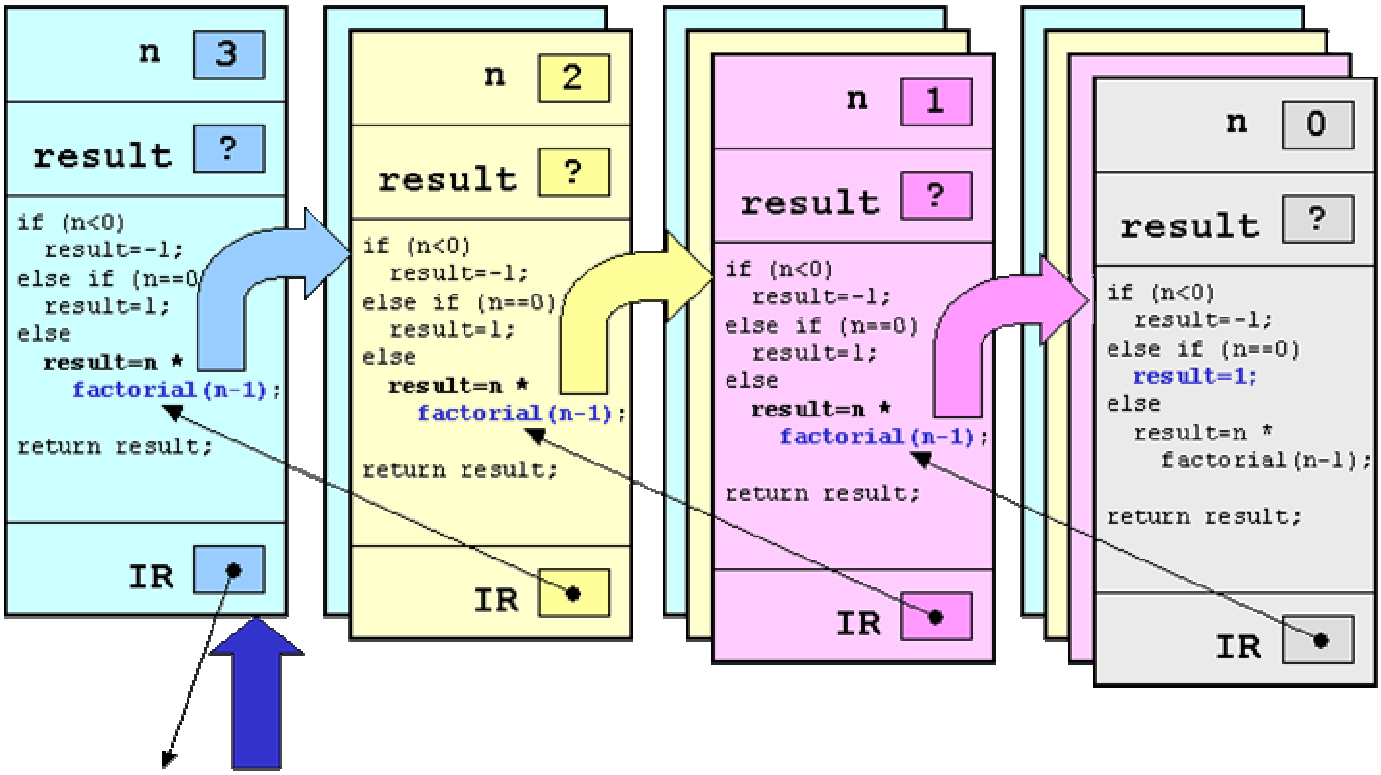
Condizioni come (n == 0) si chiamano clausole di chiusura o casi base perché garantiscono che la ricorsione termini.
Notare che, nell'esempio sopra, il metodo restituisce -1 quando l'argomento è un numero negativo (lasciando al chiamante la responsabilità di controllare che il parametro attuale della chiamata a fattoriale sia maggiore o uguale a zero).

3.3 Come funziona la ricorsione

Supponiamo di eseguire il metodo:

```
public static void main(String[] args)
{
…
      int i = MyRecursiveMethods.factorial(3);
      …
}
```

Per l'invocazione di factorial(3), il record di attivazione è aggiunto in cima alla pila (con una operazione di push), e i record relativi alle successive invocazioni di factorial vengono "impilati" su di esso (mediante operazioni di push successive).



Arrivati al caso base i record iniziano ad essere "scaricati" dalla pila restituendo mano a mano i valori ottenuti e passando il controllo al corrispondente indirizzo di ritorno. Quando una chiamata di factorial termina, il corrispondente record di attivazione è eliminato (operazione di pop). Alla fine, il flusso continua dall'indirizzo di ritorno nel main da dove il metodo è stato chiamato per la prima volta.

3.4 Ricorsione diretta

Si parla di ricorsione indiretta quando nella definizione di un metodo compare la chiamata ad un altro metodo il quale direttamente o indirettamente chiama il metodo iniziale. Un esempio di ricorsione indiretta:

```
public class MyRecursiveMethods
{
	…
	// altri metodi

	// notare che si restituisce direttamente il risultato, evitando l'uso della
variabile locale result

	public static int ping(int n)
	{
		if (n<1)
			return 1;
		else
			return pong(n-1);	// chiamata di pong
	}

	public static int pong(int n)
	{
		if (n<0)
			return 0;
		else
			return ping(n/2);	// chiamata di ping
```

```
	}

	// altri metodi
	…
}
```

Notare che, nell'esempio sopra, i metodi sono cooperanti nel senso che si invocano ripetutamente a vicenda (indirettamente), dando luogo ad un caso particolare di ricorsione indiretta, detto ricorsione mutua.

3.5 Ricorsione mutua

Un metodo implementa una ricorsione multipla quando all'interno della propria definizione compare la chiamata direttamente al metodo stesso almeno due volte. Un classico esempio di ricorsione multipla è l'implementazione dei numeri di Fibonacci, la cui definizione è riportata sotto:

```
	fib(0) = 0
	fib(1) = 1
	fib(n) = fib(n - 1) + fib(n - 2)    (se n>1)
```

Notare che, come per il fattoriale, la funzione è definita solo su interi non negativi. In Java il metodo per la implementazione dei numeri di Fibonacci può essere definito come:

```
public class MyRecursiveMethods
{
	…
	// altri metodi

	public static int fib(int n)
	{
	if (n<0)
		return -1;              //anomalie
	else if (n==0||n==1)
		return n;               //casi base
	else
		return fib(n-1)+fib(n-2);     //ricorsione
	}

	// altri metodi
	…
}
```

3.6 Problemi con la ricorsione

La ricorsione è una tecnica molto potente per decomporre problemi complessi, ma quando si scrivono programmi ricorsivi si incontrano tre problemi comuni:

1. ricorsione infinita: è un grave errore di programmazione che tipicamente si verifica perché manca la clausola di chiusura per terminare (errata gestione di anomalie e casi base) o perché i valori del parametro non si semplificano (errata gestione delle chiamate ricorsive);
2. spazio sprecato: si scrivono metodi che contengono variabili locali non necessarie, oppure non usate, e così la memoria del sistema non è utilizzata in modo efficiente;
3. complessità alta: per certi problemi le soluzioni ricorsive hanno intrinsecamente complessità non lineare;

Adesso, per ognuna delle categorie di problemi elencate sopra, vediamo dei semplici esempi che mostrano il fulcro del problema.

3.6.1 Ricorsione infinita

In una ricorsione infinita un metodo chiama se stesso infinite volte. Come abbiamo già detto, ciò si verifica perché i valori del parametro non si semplificano, o perché manca la clausola di chiusura per terminare. Per esempio, vediamo cosa succede nell'implementazione del fattoriale quando omettiamo la gestione dei casi base:

```
public class MyRecursiveMethods
{
	…
	// altri metodi

	public static int badFactorial(int n)
	{
		return (n*badFactorial(n-1));
	}
	…
	// altri metodi
}
```

Dopo un certo numero di chiamate la memoria disponibile per questo scopo si esaurisce e il programma termina automaticamente segnalando un errore di trabocco della pila (stack overflow error).

3.6.2 Spazio sprecato

Spesso, quando si sviluppano soluzioni ricorsive si scrivono metodi che contengono variabili locali non necessarie, oppure non usate. Per esempio:

```
public class MyRecursiveMethods
{
	…
	// altri metodi

	public static int stackGreedyFactorial(int input)
	{
		int result, n;
		double nonUsato;
		n=input;

		if(n==0)
			result=1;
		else
			result=n* stackGreedyFactorial(n-1);
		return result;
	}

	// altri metodi
	…
}
```

Conviene sempre eliminare le variabili locali che non sono necessarie oppure non usate perché, come abbiamo visto nelle figure della pila di attivazione, con ogni chiamata del metodo l'ambiente di programma alloca spazio per tutte le variabili locali. Da notare che esistono versioni più semplici del fattoriale di un numero che non hanno nessuna variabile locale. Per esempio:

```
public class MyRecursiveMethods
{
	…
	// altri metodi
```

```
	public static int simplerFactorial(int n)
	{
		if(n<0)
			return-1;
		if(n==0)
			return 1;
		else
			return (n*simplerFactorial(n-1));
	}

	// altri metodi
	…
}
```

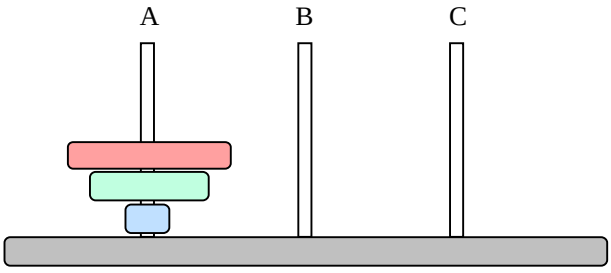
4.1.1 Esercitazione: la ricerca binaria

```
import javax.swing.*;

public class Test_BinSearch
{
	public static void main(String[] args)
	{
		int lung;
		int[] vettore;
		int elemento;
		int posizione;
		System.out.println("Ricerca binaria");
		String input1 = JOptionPane.showInputDialog("Inserisci la lunghezza
del vettore: ");
		lung = Integer.parseInt(input1);
		vettore=new int[lung];
		for (int i=0;i<lung;i++)
		{
			String input2 = JOptionPane.showInputDialog("Inserisci
l'elemento N°"+(i+1)+": ");
			vettore[i] = Integer.parseInt(input2);
		}
		Arrays.sort(vettore);
		String input3 = JOptionPane.showInputDialog("Inserisci l'elemento da
cercare: ");
		elemento = Integer.parseInt(input3);

		posizione=BinSearch(vettore,0,lung,elemento);
		if(posizione==-1)
			System.out.println("Elemento non trovato");
		else
			System.out.println("Elemento "+elemento+" in posizione
"+posizione+".");
	}

	public static int BinSearch(int[] a, int inf,int sup, int x)
	{
		if (inf>sup)
			return -1;
		int i = (inf+sup)/2;
		if (a[i]==x)
			return i;
		else if (a[i] < x)
			return BinSearch(a,i+1,sup,x);
		else
			return BinSearch(a,inf,i-1,x);
	}
}
```

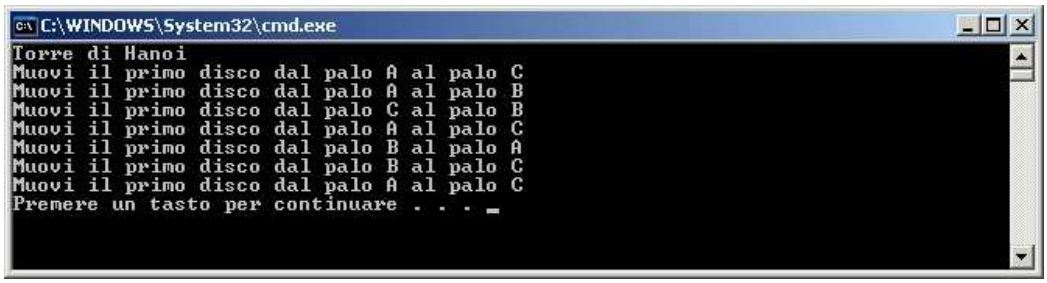
Notare come si sia utilizzato il metodo sort della classe Arrays, tale chiamata è essenziale poiché la ricerca binaria opera su un vettore già ordinato.

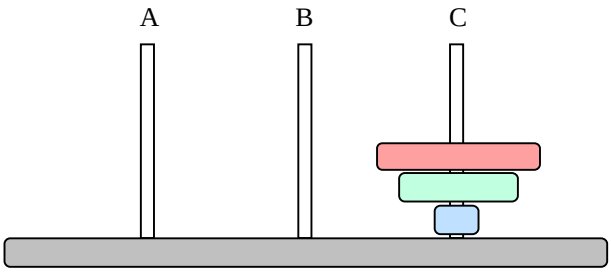
4.1.2 Esercitazione: la torre di Hanoi

```
import javax.swing.*;

public class Test_hanoi
{
      public static void main(String[] args)
      {
            int n;
            System.out.println("Torre di Hanoi");
            String input = JOptionPane.showInputDialog("Inserisci il livello della
torre di Hanoi: ");
            n = Integer.parseInt(input);
            runHanoi(n,'A','B','C');
      }

      public static void runHanoi(int n, char x, char y, char z)
      {
            if (n==1)   // base
                  System.out.println("Muovi il primo disco dal palo " + x + " al
palo " + z);
            else        // ricorsione
            {
                  runHanoi(n-1,x,z,y);
                  runHanoi(1,x,y,z);
                  runHanoi(n-1,y,x,z);
            }
      }
}
```



Ecco la soluzione proposta dal programma `Test_Hanoi`:





5.1 La notazione asintotica

Affinché sia possibile confrontare l'efficienza di un algoritmo con un altro e stabilire quindi quale dei due abbia una maggiore efficacia è necessario definire un modello di calcolo appropriato. Il modello deve essere il più generale possibile, deve poi basare il confronto anziché sul tempo impiegato da un algoritmo ma dal numero di istruzioni che esso richiede per conseguire il suo scopo. Quindi esclusa l'esigenza di una macchina di riferimento come criterio di confronto fra gli algoritmi ed accettato il modello di confronto basato sul numero dei passi o di operazioni di un algoritmo cerchiamo adesso di aggiungere al modello pensato ulteriori modifiche o sue complessità. Se dunque al tempo abbiamo preferito il numero di passi o di istruzioni è subito bene precisare che la durata di una istruzione non è quasi mai uguale a tutte le altre ed ha una forte dipendenza con gli operandi dell'istruzione stessa. Questa considerazione non rientra nel modello da noi pensato e per questo motivo prende il nome di modello di misurazione del tempo con misura di costo uniforme. Basti pensare al calcolo dei numeri di Fibonacci, all'inizio un numero di Fibonacci può essere rappresentato mediante un certo numero di bit, tuttavia, man mano che il calcolo procede lo stesso numero potrebbe addirittura venire rappresentato con più di una cella di memoria. Come conseguenza di questa cosa le istruzioni successive avranno allora una durata diversa dalle prime. Per ovviare a questa cosa il modello viene complicato ed è stato proposto un altro criterio di misurazione noto come misura di costo logaritmico e che tiene quindi conto della dimensione degli operandi coinvolti. In questo modello l'esecuzione di una operazione su un numero `n` implica un costo proporzionale a `log n`. Definizione: si definisce complessità asintotica di un problema la complessità assunta da questo problema quando le sue dimensioni (n) tendono a valori molto grandi. L'idea di base è quella di raggruppare la complessità degli algoritmi in categorie di funzioni soggette a certe limitazioni sul loro comportamento all'infinito. Dette g(n) e f(n) due funzioni dai naturali ai reali non negativi, si dice che g(n) appartiene alla classe di funzioni O(f(n)), o che è di ordine f(n), se esistono due costanti positive c e n_0 per cui: $g(n) \leq c \cdot f(n)$ per ogni $n \geq n_0$. Esempi di complessità sono:

- O(1) complessità costante;
- O(n) complessità lineare;
- O(n2) complessità quadratica;
- ...
- O(nlogn) complessità sottolineare;
- Le precedenti sono considerate complessità polinomiali
- O(kn) complessità esponenziale.

La complessità costante si ha quando gli algoritmi fanno sempre lo stesso numero di operazioni indipendentemente dal numero di dati. Tipiche complessità nlogn le hanno gli algoritmi di ordinamento. Problemi complessi come la torre di Hanoi hanno complessità esponenziale. Il Bubble Sort è uno dei più intuitivi algoritmi di ordinamento per scambio:

- si confrontano i primi due elementi di una lista;
- se non sono ordinati li si scambiano;
- si considerano i successivi elementi e si ripetono i primi due passi;

- terminata la scansione della lista si ripete senza considerare l'ultimo elemento della lista.

Il Quick Sort è un algoritmo di ordinamento che si basa sulla divisione iterativa della lista da ordinare:

- si sceglie un elemento della lista detto perno (pivot);
- si suddivide la lista in maniera che tutti gli elementi minori del perno si trovino alla sua sinistra e i maggiori alla sua destra;
- si ripete la procedura sui due sottoinsiemi individuati;

Per il Bubble sort si ha che:

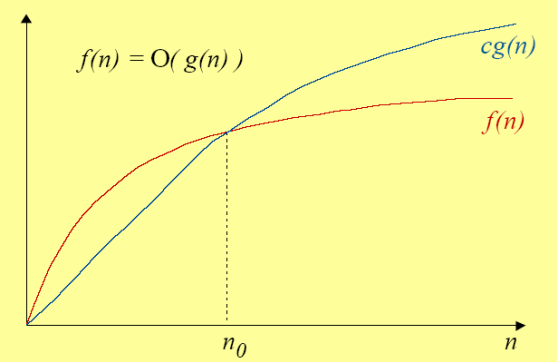
- Caso ottimo (vettore già ordinato): T(n)=n-1
- Caso pessimo (vettore ordinato in senso inverso): T(n)=n(n-1)/2
- Caso medio: T(n)=(n2+n·ln(n))/2

La complessità asintotica è $O(n^2)$. Per il Quick sort:

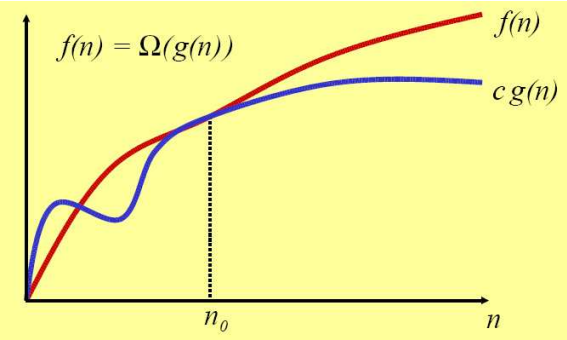
- Caso ottimo (vettore disordinato): T(n)=2T((n-1)/2)+O(nlogn)
- Caso pessimo (vettore ordinato): $T(n)=T(n-1)+O(n^2)$

Più in generale, data una funzione $f(n)$, definiamo:

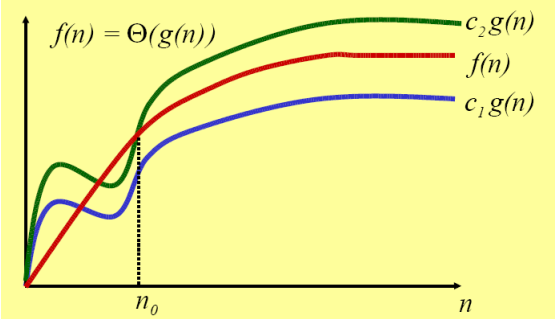
- $O(f(n))=\left\{g(n):\exists c>0 \;\; e \;\; n_0 \geq 0 \;\; tali \;\; che \;\; g(n) \leq cf(n) \;\; per \;\; ogni \;\; n \geq n_0\right\}$;



- $\Omega(f(n))=\left\{g(n):\exists c>0 \;\; e \;\; n_0 \geq 0 \;\; tali \;\; che \;\; g(n) \geq cf(n) \;\; per \;\; ogni \;\; n \geq n_0\right\}$;



- $\Theta(f(n)) = \left\{ g(n) : \exists c_1, c_2 > 0 \;\; e \;\; n_0 \geq 0 \;\; tali \;\; che \;\; per \;\; ogni \;\; n \geq n_0 \;\; c_1 f(n) \leq g(n) \leq c_2 f(n) \right\}$;



Al di là del formalismo si può anche dire che:

- Se una funzione $g(n) \in (Of(n))$ allora $g(n)$ cresce al più come $f(n)$;
- Se una funzione $g(n) \in (\Omega f(n))$ allora $g(n)$ cresce almeno come $f(n)$;
- Se una funzione $g(n) \in (\Theta f(n))$ allora $g(n)$ cresce esattamente come $f(n)$;

6.1 Strutture dati elementari

Le strutture dati, elementari, consentono una gestione efficiente di collezioni di oggetti. Gli algoritmi manipolano queste collezioni di dati operando sull'insieme talvolta detto anche insieme dinamico. Un insieme è dinamico poiché con il passare del tempo esso può aumentare la sua dimensione ma la può anche diminuire. Gli algoritmi effettuano quindi sull'insieme operazioni tipiche di cancellazione, rimozione e ricerca. Volendo poi essere più precisi bisogna subito dire che ogni struttura dati ha una serie di istruzioni tipiche, ad esempio, le istruzioni prima elencate sono tipiche di un tipo di dato dizionario. Spesso gli elementi di un insieme dinamico sono oggetti strutturati che contengono:

- una "chiave" identificativa k dell'elemento all'interno dell'insieme;
- altri "dati satellite", contenuti in opportuni campi di cui sono costituiti gli elementi dell'insieme;

In genere i dati satelliti non vengono utilizzati in maniera diretta per realizzare operazioni sull'insieme. Quindi, riassumendo quanto fin qui detto, per struttura dati intendiamo una particolare organizzazione delle informazioni che permette di supportare in modo efficiente le operazioni di un tipo di dato. Un semplice esempio di struttura dati è l'array ordinato di n elementi. Su di esso è possibile sviluppare algoritmi di ricerca oppure operazioni di inserimento e/o cancellazione. Altre strutture dati sono le pile, le code e gli alberi. In seguito vedremo come implementare queste strutture.

6.2 Strutture dati indicizzate

La realizzazione delle strutture dati può avvenire secondo due tecniche fondamentali, una tecnica basata su strutture indicizzate (array) ed una tecnica basata su strutture collegate (record e puntatori). Ovviamente la scelta di una tecnica piuttosto che un'altra ha un impatto sulle prestazioni di molte operazioni fondamentali come ricerca, inserimento e cancellazione. La struttura indicizzata più semplice è un array, esso è costituito da una collezione di celle tutte numerate le quali possono contenere un tipo prestabilito. Gli indici delle celle di un array sono numeri consecutivi e consentono l'accesso al singolo elemento dell'array. Un prima nota negativa di questa struttura deriva dal fatto che non è possibile aggiungere nuove celle ad un array. Il ridimensionamento di un array è possibile solo mediante riallocazione, ciò avviene creando un nuovo array con la dimensione voluta.
Esiste inoltre una tecnica, detta del raddoppiamento-dimezzamento, che consente di mantenere efficientemente in un array una collezione non ordinata. L'array è occupato nelle prime posizioni dagli elementi della collezione, il restante contenuto è indefinito. Ad ogni inserimento/cancellazione non

sempre avviene una riallocazione ma ciò è necessario solo se la lunghezza del vettore è invariante rispetto alla relazione $n \leq h \leq 4n$ dove n solo le celle occupate ed h è la dimensione del vettore.

6.3 Strutture dati collegate

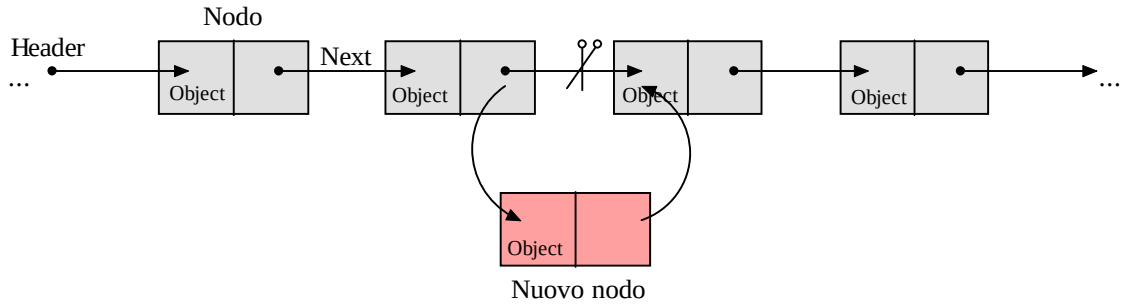
Gli elementi di base di una struttura collegata sono i record, che come le celle degli array sono numerati e contengono ciascuno un oggetto della collezione. Mentre la numerazione delle celle degli array è locale al singolo array, i numeri associati ai record sono tipicamente i loro indirizzi in memoria, e quindi sono globali nell'ambito di un programma. Pertanto, diversamente dagli indici degli array, gli indirizzi dei record non sono necessariamente consecutivi. Il collegamento fra due record è generalmente realizzato tramite un puntatore. Questa struttura permette di inserire e cancellare record in modo molto flessibile, ciò avviene semplicemente aggiornando i puntatori nella struttura. L'operazione più costosa per questa struttura è la ricerca che nel caso peggiore richiede un tempo costante, a tale proposito è bene osservare che l'operazione di cancellazione richiede prima una ricerca dell'elemento, se invece si conosce già l'indirizzo dell'elemento da cancellare tale operazione è immediata. Infine una struttura dati può essere semplicemente collegata, doppiamente collegate e circolare doppiamente collegata.

6.4 La classe nodo

```
class Nodo
{
    /* Attributi dell'istanza */
    Object elemento;
    Nodo next;

    /* Costruttore 1: aggiunge il primo elemento */
    Nodo(Object elemento)
    {
        this.elemento=elemento;
        this.next=null;
    }

    /* Costruttore 2:aggiunge altri elementi in testa*/
    Nodo(Object elemento,Nodo next)
    {
        this.elemento=elemento;
        this.next=next;
    }

    /* Stampa tutti gli elementi effettuando una chiamata ricorsiva dello stesso
metodo */
    public void print()
    {
        System.out.println(elemento);
        if (next!=null) next.print();
    }

    /* Restituisce l'elemeto di indice richiesto */
    public int get(int indice)
    {
        if(indice==0)
            return elemento;
        else
            return next.get(--indice);
    }

    /* Aggiunge altri elementi*/
    public void add(Object o,int pos)
    {
```

```
        if(pos==1)
        {
            Nodo NuovoNodo;
            if(next!=null)
            {
                NuovoNodo = new Nodo(o,next);
            }
            else
            {
                NuovoNodo = new Nodo(o);
            }
            next=NuovoNodo;
        }
        else
        {
            pos--;
            next.add(o,pos);
        }
    }

    /* Conta il numero di nodi */
    public int conta()
    {
        if(next==null)
        {
           return 1;
        }
        else
        {
            return (1 + next.conta());
        }
    }

    /* Effettua una ricerca binaria di un elemento */
    Nodo search(Object ele)
    {
        Nodo NuovoNodo= new Nodo(-1);
        if(this==null)
        {
            return null;
        }
        else if(ele!=elemento)
        {
            next.search(ele);
        }
        else
        {
            NuovoNodo=this;
        }
        return NuovoNodo;
    }
}
```

L’utilità del secondo costruttore in particolare è spiegata dalla seguente figura:



Indicato dunque l’elemento a cui si vuole far precedere un nuovo nodo si effettua un aggiornamento sui puntatori affinché questi indichino in `next` l’`object` desiderato.

6.5 La classe Lista

Quella che segue è una realizzazione ricorsiva di una lista in java:

```
public class Lista
{
	private Nodo header;
	private int size;

	public Lista()
	{
		size=0;
	}
}
```

A partire dalla classe nodo si possono creare dei nodi, vi sono due costruttori, il primo richiede la conoscenza del solo elemento da aggiungere, nel secondo invece si può dare al costruttore anche il prossimo elemento che deve seguire il primo. Adesso completeremo questa classe e quindi anche questa struttura inserendo in essa dei metodi utili alla gestione di una lista.. Uno dei primissimi metodi da aggiungere è il metodo add che consente l’inserimento di un nuovo elemento nella lista:

```
	public void add(Object o)
	{
		if(size==0)
		{
			header = new Nodo(o);
		}
		else
		{
			Nodo NuovoNodo=new Nodo(o,header);
		}
		size++;
	}
```

Il metodo inserisce l’oggetto nella lista, ponendolo nell’header, se la misura/size della lista è zero e dunque se la lista è vuota. Altrimenti, se la lista non è vuota il metodo crea un nuovo nodo utilizzando l’altro costruttore che inserisce l’elemento facendolo seguire all’attuale header della lista. Metodo remove():

```
	public Object remove()
	{
		Object elemento;
		if(size==0)
```

```
        {
                elemento=null;
        }
        else
        {
                elemento=header.elemento;
                header=header.next;
                size--;
        }
        return elemento;
}
```

Il metodo restituisce l'elemento eliminato, se la lista è vuota viene ritornato il riferimento a null, altrimenti si procede assegnando l'header della lista all'elemento successivo dopodichè si decrementa la misura/size della lista.
Il prossimo metodo che vedremo si occupa di fornire al programma chiamante la possibilità di interrogare la lista affinché si possa stabilire se essa sia vuota oppure no, il metodo si chiama isEmpty():

```
boolean isEmpty()
{
        if(size>0)
                return false;
        else
                return true;
}
```

Metodo che inserisce un elemento in una data posizione:

```
public void add(Object o,int pos)
{
        if(pos==0)
        {
                Nodo NuovoNodo;
                if(next!=null)
                {
                        NuovoNodo = new Nodo(o,next);
                }
                else
                {
                        NuovoNodo = new Nodo(o);
                }
                next=NuovoNodo;
        }
        else
        {
                pos--;
                next.add(o,pos);
        }
}
```

Metodo che restituisce l'elemento in posizione dell'indice dato:

```
Object get(int indice)
{
        if(indice==0)
                return elemento;
        else
                return next.get(--indice);
}
```

Metodo che restituisce la misura/size della lista:

```
	public int getSize()
	{
		return size;
	}
```

Segue il listato completo per generare la struttura dati Lista:

```
public class Lista
{
	private Nodo header;
	private int size;

	public Lista()
	{
		size=0;
	}

	public void add(Object o)
	{
		if(size==0)
		{
			header = new Nodo(o);
		}
		else
		{
			Nodo NuovoNodo=new Nodo(o,header);
		}
		size++;
	}
	public void add(Object o,int pos)
	{
		add(o,pos);
	}
	public Object remove()
	{
		Object elemento;
		if(size==0)
		{
			elemento=null;
		}
		else
		{
			elemento=header.elemento;
			header=header.next;
			size--;
		}
		return elemento;
	}
	boolean isEmpty()
	{
		if(size>0)
			return false;
		else
			return true;

	}
	void print()
	{
		header.print();
```

```
	}
	public int getSize()
	{
		return size;
	}
	Object get(int indice)
	{
		return get(indice);
	}
}
```

Riassumendo molto brevemente, una lista è una struttura sequenziale nella quale si possono inserire o togliere elementi in un tempo costante, ossia indipendente dalla dimensione della struttura stessa; è la struttura dati ideale per applicazioni che non richiedono un accesso casuale ai dati. Un treno merci rappresenta un buon esempio di lista: ogni vagone può essere tolto semplicemente scollegandolo dai due adiacenti, che poi vengono collegati tra loro.
(Vedi John R.Hubbard Pag.146)

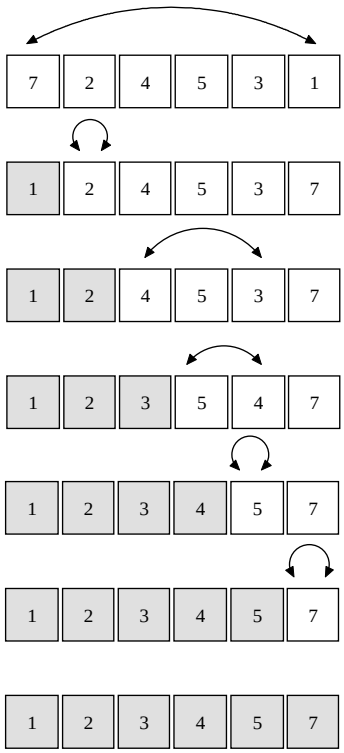
7.1 Algoritmi di ordinamento

In questo capitolo affrontiamo il problema dell'ordinamento di un array. Si tratta di un problema ricorrente in informatica e ciò presuppone almeno una relazione d'ordine sugli elementi da ordinare, tipicamente l'ordine avviene mediante una chiave e problemi di questo tipo sono validi sia per strutture dati collegate con puntatori che per strutture dati sequenziali come gli array.
Esistono numerosi algoritmi di ordinamento, alcuni di questi sono efficienti solo se la struttura dati da ordinare è di tipo sequenziale, esistono tuttavia (anche se con maggiore complessità) le versioni equivalenti per strutture dati a puntatori. Inoltre, alcuni algoritmi di ordinamento si basano su confronti tra elementi, altri invece sono di natura logica ed effettuano un ordinamento senza effettuare confronti fra elementi. I primi algoritmi di ordinamento che vedremo sono di una complessità $O(n^2)$, mentre i successivi algoritmi che si vedranno hanno una maggiore efficienza ed una complessità $O(n\log n)$. La maggior parte degli algoritmi che raggiungono una complessità $O(n\log n)$ si basano sul concetto del "dividi e conquista", l'ordinamento è cioè raggiunto per raffinamenti successivi sull'insieme di partenza. Vedremo che $O(n\log n)$ è quanto meglio possiamo ottenere in un modello basato su confronti, mentre $O(n^2)$ è l'andamento peggiore (equivale a fare tutti i confronti tra le coppie di elementi). In alcuni casi infine, se gli elementi da ordinare godono di determinate proprietà (gli elementi sono ad esempio interi oppure sono confinati ad assumere valori in ristretto intervallo) si possono mettere in atto delle metodologie che spingono la complessità fino a diventare O(n).

n	10	100	1000	10^6	10^9
$n\log_2 n$	≈ 33	≈ 665	$\approx 10^4$	$\approx 2\cdot 10^7$	$\approx 3\cdot 10^7$
n^2	100	10^4	10^6	10^{12}	10^{18}

7.2 Selection sort

Il selection sort è uno degli algoritmi più semplici ed intuitivi, dato un array di elementi da ordinare il selection sort sceglie il minimo fra tutti gli elementi dell'array e lo colloca in testa all'elenco. Le successive ricerche del minimo collocheranno il valore individuato ordinandolo in base a quelli già sistemati in precedenza. La figura che segue mostra i passi per ogni iterazione del metodo:



Supponendo di applicare il selection sort a partire dall'indice k di un array il metodo sceglie il minimo tra gli $n-k$ elementi non ancora ordinati e lo assegna alla posizione $k+1$.

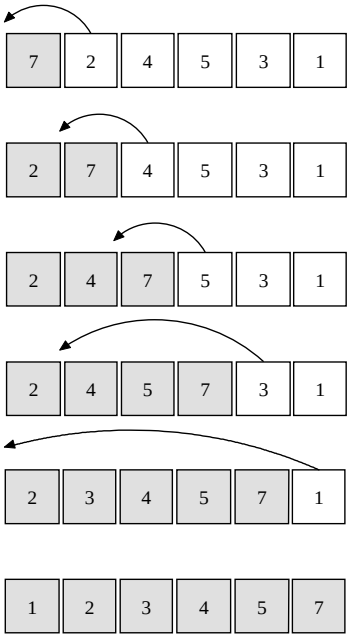
L'algoritmo selection sort ordina n elementi eseguendo nel caso peggiore $\Theta(n^2)$ confronti (eseguo esattamente n^2 confronti). Tuttavia, se gli elementi sono già ordinati non ho in questo algoritmo la possibilità di fermare l'ordinamento e si eseguiranno necessariamente tutti i confronti. Pertanto sia nel caso peggiore, medio o inferiore la complessità è al più come $O(n^2)$.

Supponendo di implementare un metodo di ordinamento basato su un vettore A di elementi si seguiranno le seguenti linee di codice:

```
/* Selection sort */
public void SelectionSort()
{
      int temp;
      for(int i=0; i<A.length-1; i++)
      {
            for(int j=i+1; j<A.length; j++)
            {
                  if(A[i]>A[j])
                  {
                        temp=A[i];
                        A[i]=A[j];
                        A[j]=temp;
                  }
            }
      }
}
```

7.3 Insertion sort

Nella figura sono mostrati i passi necessari all'ordinamento:



Supponendo di applicare l'insertion sort a partire dall'indice k di un array il metodo seleziona il $(k+1)$-esimo elemento e lo colloca in posizione corretta rispetto ai primi k elementi già ordinati.

Anche in questo algoritmo, nel caso peggiore e/o medio la complessità è di $O(n^2)$.

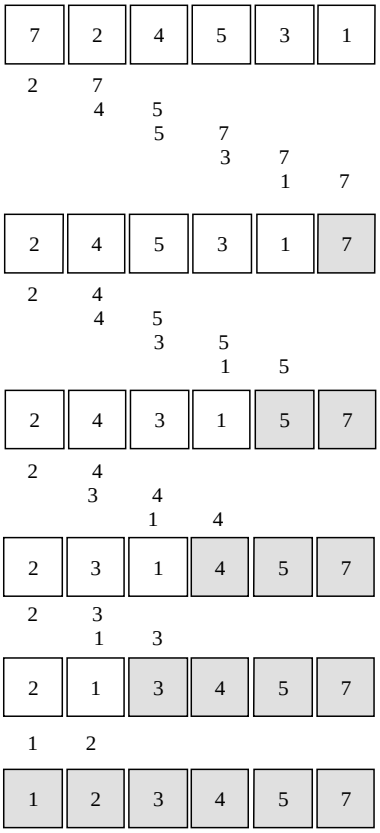
I due algoritmi finora visti si dicono essere incrementali poiché lavorano estendendo in maniera graduale e progressiva una sottosequenza ordinata finché essa non comprende tutti gli elementi.. Entrambi gli algoritmi inoltre sono lenti poiché abbiamo visto che il tempo di esecuzione e quindi di ordinamento è proporzionale ad $O(n^2)$, e ciò si traduce (nel caso peggiore) ad effettuare tutti i possibili confronti tra le coppie di elementi. Supponendo di implementare un metodo di ordinamento basato su un vettore A di elementi si seguiranno le seguenti linee di codice:

```
/* Inserction sort */
public void InserctionSort()
      {
            int temp,i,j;
            for(j=1; j<V.length; j++)
            {
                  i=j-1;
                  temp=V[j];
                  while( i>=0 && temp<V[i] )
                  {
                        V[i+1]=V[i];
                        i--;
                  }
                  V[i+1]=temp;
            }
      }
```

Questo algoritmo viene spesso volte usato per effettuare l'inserimento in ordine di un elemento in una lista già ordinata. Supponendo che siano stati ordinati i primi i-1 elementi e che occorra inserire un successivo elemento, a tale scopo occorrerà confrontare il nuovo elemento da inserire con la sottosequenza ordinata V[1]…V[i-1] individuando così la giusta posizione j per l'elemento all'interno di questa. Nel caso medio si effettuano n)n-1)/2 numeri di operazioni mentre la complessità, come già detto, è del tipo $O(n^2)$.

7.4 Bubble sort

L'algoritmo prevede una serie di scansioni dell'array, per ogni scansione si confrontano le coppie di elementi adiacenti, si effettua uno scambio fra gli elementi confrontati se essi risultano non ordinate fra loro. Alla fine di una scansione fatta sull'intero array l'elemento più grande occuperà l'ultima posizione, sprofondando quindi man mano verso il basso. Se durante un ciclo di scansione non si effettua nessuno scambio allora l'array risulta essere ordinato e l'algoritmo si arresta.
Nella figura sono mostrati i passi necessari all'ordinamento:



Anche per il bubble sort la complessità è dell'ordine di $O(n^2)$.
Supponendo di implementare un metodo di ordinamento basato su un vettore A di elementi si seguiranno le seguenti linee di codice:

```
/* Bubble sort */
public void BubbleSort()
{
      int i,j,temp;
      for(i=0;i<V.length-1;i++)
      {
            for(j=0;j<V.length-i-1;j++)
            {
                  if(V[j]>V[j+1])
                  {
                        temp=V[j];
                        V[j]=V[j+1];
                        V[j+1]=temp;
                  }
            }
      }
}
```

L'ordinamento per scambi detto anche bubble sort si applica tipicamente al caso di una lista di n elementi di tipo ordinato memorizzati in un array ed ha per obiettivo il riordinamento degli elementi mediante spostamento fisico degli elementi. L'algoritmo prevede l'ordinamento a mezzo di scambi tra coppie successive, in particolare esso esamina la prima coppia di elementi V[j] e V[j+1] ed effettua uno scambio se essa non è ordinata. Si prosegue in questo modo fino alla coppia V[n-1], V[n]. Al termine del ciclo l'elemento massimo occupa la posizione n-sima, il procedimento viene poi applicato alla sottolista V[1] … V[n-1]. Il numero di operazioni nel caso medio è n(n-1)/4 la complessità è invece $O(n^2)$.

7.5 Quick sort

Gli algoritmi di ordinamento che vedremo nei prossimi paragrafi seguono un approccio di tipo divide et impera. Il quick sort in particolare partiziona gli elementi in due sequenze che vengono ordinate ricorsivamente e poi ricombinate. Nonostante le cattive prestazioni nel caso peggiore (si comporta infatti come un selection sort) il quick sort rimane il miglior algoritmo nel caso medio. Nella fase di divide l'algoritmo sceglie un elemento x detto pivot (perno) e partiziona la sequenza in due sottoinsiemi o per meglio dire in due sottovettori che si suppone essere non vuoti. I due sottovettori devono contenere rispettivamente elementi minori o uguali ad x (il perno scelto) e maggiori ad x. Pertanto, supposto ad esempio che q sia l'indice del perno scelto i due sottovettori avranno come indici A[1,2,…,q] il primo sottovettore e B[q+1,…,r] il secondo sottovettore.
Nella successiva fase di "conquista" i due sottovettori vengono ordinati ricorsivamente con il metodo quick sort fino a giungere ai due casi base della ricorsione. In effetti la parte più grossa del lavoro di ordinamento è svolta dall'algoritmo che partiziona il vettore e che ha questi casi base:

- L'algoritmo partiziona non effettua alcun spostamento nei confronti del perno;
- L'algoritmo partiziona effettua al più uno spostamento;

Nella fase finale di "impera" le sequenze ordinate vengono ordinate ed a partire dal caso base che costituisce un particolare sottovettore già ordinato si ricostruisce il vettore interamente ordinato. La partizione viene effettuata inizializzando due indici che scorreranno il vettore uno da sinistra verso destra e l'altro da destra verso sinistra. Ogni elemento indicato dall'indice (potremo ad esempio chiamarli indice inferiore e superiore) viene confrontato con il perno scelto, in particolare gli indici che puntano agli elementi del vettore saranno incrementati (indice inferiore) e decrementati (indice superiore) fintanto che l'elemento del vettore è minore o uguale al perno scelto (per il primo sottovettore) e maggiore al perno scelto (per il secondo sottovettore). Quando a seguito di questi aggiornamenti i due indici si incrociano o si scambiano posizione tra di loro la partizione termina. In altre parole, quando l'indice inferiore supera quella superiore si ha la partizione desiderata.

```
public void QuickSort()
{
      sort(V,0,V.length-1);
}
public void sort(int[] V,int left,int right)
{
      int i,j,mid,tmp;
      i=left;
      j=right;
      mid=V[(left+right)/2];
      do
      {
            while(V[i]<=mid)
            {
                  i++;
            }
            while(mid<=V[i])
            {
```

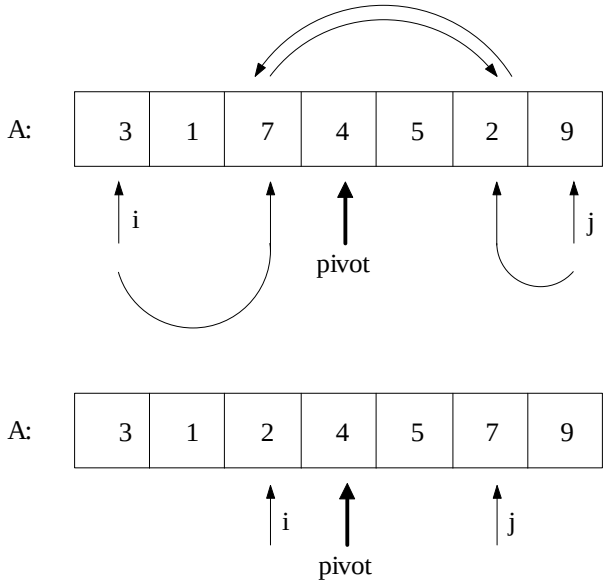
```
                j--;
            }
            if(i<=j)
            {
                tmp=V[i];
                V[i]=V[j];
                V[j]=tmp;
                i++;
                j--;
            }
        }
        while(i<=j);
        if(left<j)
            sort(V,left,j);
        if(i<right)
            sort(V,i,right);
}
```

Per quanto riguarda la complessità possiamo dire che l'operazione di partizionamento richiede (n-1) confronti, per cui:

$$C(n) = n - 1 + C(a) + C(b)$$

Le dimensioni dei due sottovettori sono tali che a+b=n-1 (escludendo cioè il perno la dimensione dei due sottovettori deve dare n-1). Nel caso peggiore, se l'elemento x (il perno scelto) può essere l'elemento più piccolo del vettore, in tali circostanze il primo sottovettore avrà dimensione a=0 mentre il secondo sottovettore avrà dimensione b=n-1:

$$C(n) = n - 1 + C(n-1) = O(n^2)$$

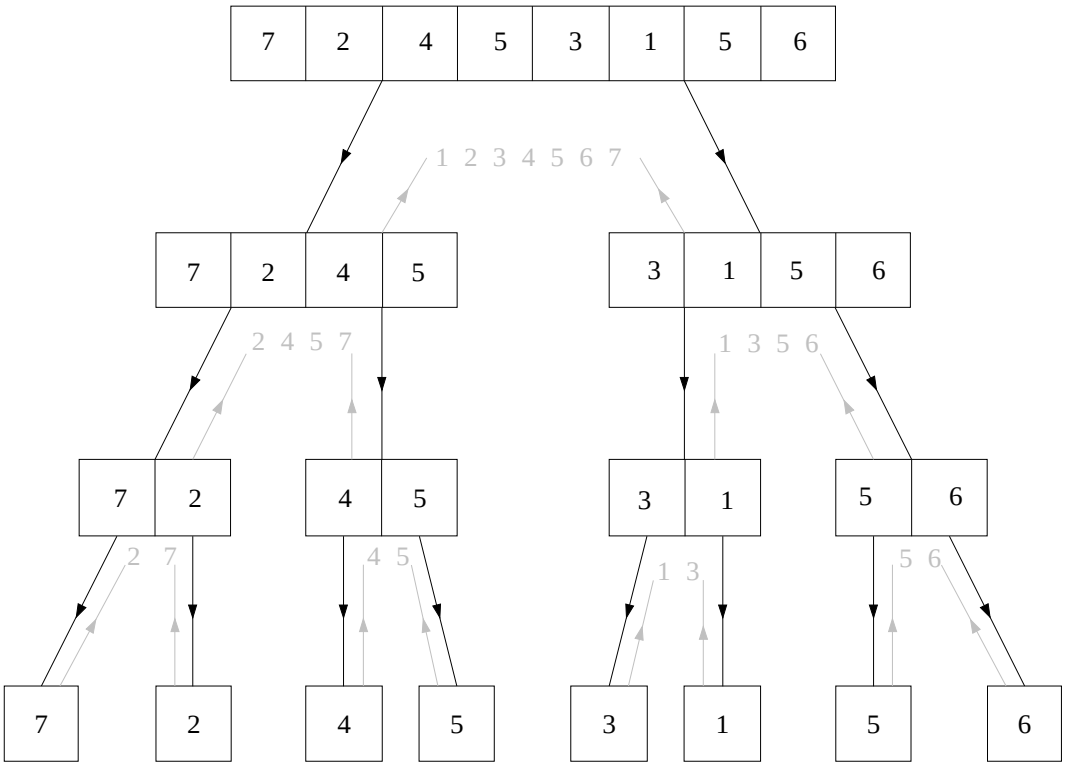


7.6 Merge sort

Il merge sort è un algoritmo di ordinamento che a differenza del quick sort opera con una complessità che è *nlogn* anche nel caso peggiore (quindi è leggermente migliore del quick sort). Tale complessità è ottenuta scegliendo le partizioni sempre in maniera bilanciata. La procedura principale è il metodo *merge* che fonde due sequenze di lunghezza l_1 ed l_2 eseguendo al più (l_1+l_2-1) confronti.

```
public void MergeSort()
{
      mSort(V,0,V.length);
}
public void mSort(int[] V,int k,int n)
{
      if(n<2) return;
      mSort(V,k,n/2);
      mSort(V,k+n/2,n-n/2);
      merge(V,k,n);
}
public void merge(int[] V,int k,int n)
{
      int[] temp=new int[n];
      int i=0, lo=k, hi=k+n/2;
      while(lo<k+n/2 && hi<k+n)
      {
            if(V[lo]<=V[hi])
                  temp[i++]=V[lo++];
            else
                  temp[i++]=V[hi++];
      }
      while(lo<k+n/2)
            temp[i++]=V[lo++];
      while(hi<k+n)
            temp[i++]=V[hi++];
      for(i=0;i<n;i++)
            V[k+i]=temp[i];
}
```

Il merge sort si serve di una struttura ausiliaria per fondere le due sequenze. Per motivare la complessità è opportuno ragionare sull'albero delle chiamate ricorsive dell'algoritmo. Tale albero, si dimostra, essere dotato di profondità $log_2 n$ (la profondità è definita come la distanza misurata in numero di archi tra la radice dell'albero e le sue foglie, torneremo su argomenti di questo tipo quando si studierà l'albero come struttura dati). Siccome il numero di confronti per fondere le due sequenze è *n-1* (trascurando quindi le costanti come si è solito fare nella notazione asintotica), e quindi la complessità del merge sort è$\theta nlogn$.

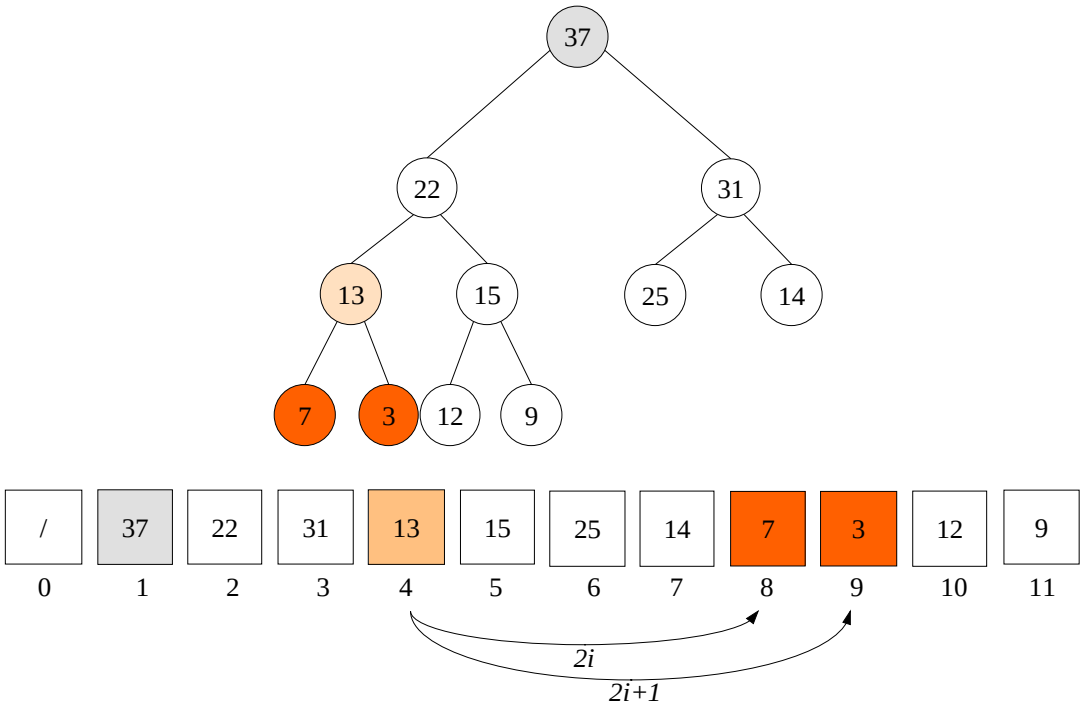


7.7 Heap sort

L'heap sort usa lo stesso approccio su cui si basa il selection sort ma esegue nel caso peggiore un numero inferiore di confronti poiché usa una struttura dati assi efficiente quale è l'heap. La creazione di un heap deve permettere alcune elementari operazioni quali, la generazione appunto della struttura dati dato un vettore, la ricerca dell'elemento più piccolo, la rimozione dell'elemento più piccolo. Un heap associato ad un insieme di elementi è un albero binario radicato con le seguenti proprietà:

- L'albero è completo fino al penultimo livello;
- Gli elementi sono memorizzati nei nodi dell'albero, ed ogni nodo può contenere uno ed un solo elemento;
- Il valore dell'elemento in un nodo è sempre maggiore o uguale al valore degli elementi nei figli;

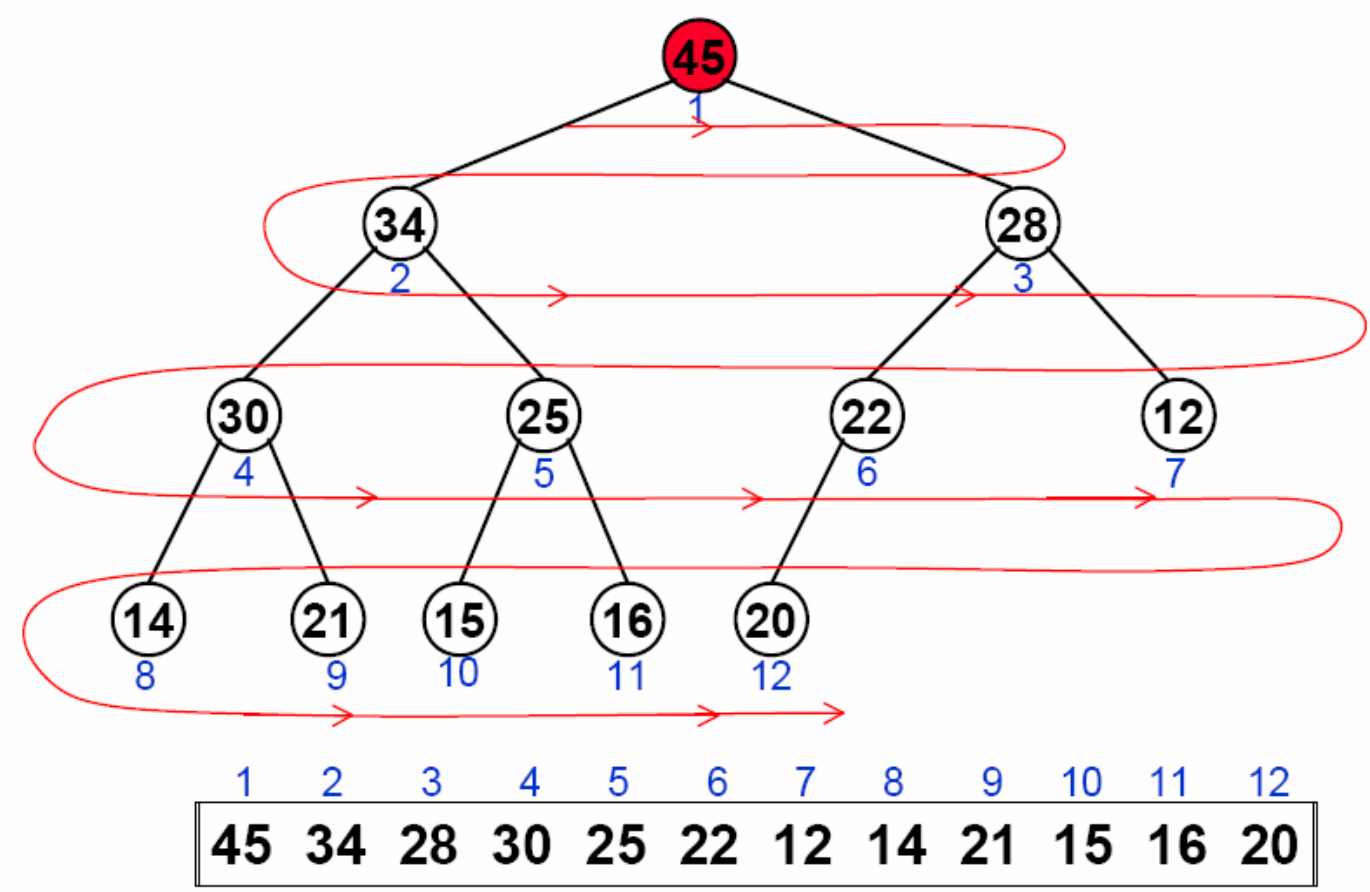
Da quanto detto si intuisce che l'heap è una variante della struttura dati albero (a differenza quindi delle proprietà elencate), in un albero binario il tempo di accesso è *O(1)* quindi costante. Un albero, e quindi anche un heap, ha una profondità o altezza di $\theta(\log n)$. Un heap rappresenta quindi un ordinamento parziale del vettore.
Supponendo di collocare il massimo in testa al vettore e quindi nella radice dell'albero, l'heap genera il vettore parzialmente ordinato rispettando le proprietà prima elencate, in particolare il nodo *i* avrà i suoi figli (che ricordiamo essere più piccoli del padre) nelle posizioni *2i* e *2i+1*.




```
public void HeapSort()
{
	int N=V.length;
	int T;
	for(int k=N/2;k>0;k--)
	{
		downHeap(V,k,N);
	}
	do
	{
		T=V[0];
		V[0]=V[N-1];
		V[N-1]=T;
		N=N-1;
		downHeap(V,1,N);
	}
	while(N>1);
}
public void downHeap(int[] V,int k,int N)
{
	int T=V[k-1];
	while(k<N/2)
	{
		int j=k+k;
		if( (j<N) && (V[j-1]<V[j]) )
		{
			j++;
		}
		if(T>=V[j-1])
		{
			break;
		}
		else
		{
			V[k-1]=V[j-1];
			k=j;
		}
```

```
        }
        V[k-1]=T;
}
```

Uno Heap può essere implementato in vari modi: come un albero a puntatori o come un array.



Alcune operazioni su heap:

- Heapify(A,i): ripristina la proprietà di Heap al sottoalbero radicato in i assumendo che i suoi sottoalberi destro e sinistro siano già degli Heap;
- Costruisci-Heap(A): produce uno Heap a partire dall'array A non ordinato;
- HeapSort(A): ordina l'array A sul posto;

Concludendo:

- L'algoritmo di ordinamento sul posto per confronto impiega un tempo O(nlogn);
- Algoritmo non immediato ne ottimo;
- Sfrutta le proprietà della struttura dati astratta heap;

L'heap sort dimostra anche che scegliere una buona rappresentazione per i dati spesso facilita la progettazione di buoni algoritmi.

8.1 Iteratore per la classe lista

Un iteratore può essere considerato un tipo di puntatore specializzato che fornisce un punto di accesso sequenziale agli elementi di un oggetto che contiene un numero finito di oggetti più semplici, detto aggregato. L'iteratore offre due operazioni fondamentali:

- Accesso all'elemento dell'aggregato attualmente puntato;
- Aggiornamento del puntatore così che punti all'elemento successivo nella sequenza;

Queste semplici operazioni permettono di accedere agli elementi di un aggregato in modo uniforme e indipendente dalla struttura interna dell'aggregato, che può essere ben più complessa delle sequenze lineari implementate da array e liste. Esempi di aggregati complessi sono l'array associativo, l'albero e la hash table. Oltre all'accesso e all'aggiornamento, un iteratore deve fornire come minimo due operazioni:

- Inizializzazione o ripristino dello stato iniziale, in cui l'elemento puntato dall'iteratore è il primo della sequenza;
- Verifica se l'iteratore ha esaurito tutti gli elementi dell'aggregato, cioè se è stato aggiornato oltre l'ultimo elemento della sequenza;

A seconda del linguaggio e delle necessità, gli iteratori possono fornire operazioni aggiuntive o esibire comportamenti diversi. Un esempio di iteratori specializzati è offerto dagli iteratori bi-direzionali, che permettono di visitare l'insieme degli elementi di un aggregato partendo dall'ultimo elemento e procedendo verso il primo. Un altro esempio è offerto dagli iteratori filtranti, che consentono di visitare soltanto il sottoinsieme degli elementi di un aggregato che soddisfa a condizioni pre-impostate all'interno dell'iteratore. Una classe iteratore viene solitamente progettata in stretta coordinazione con la corrispondente classe contenitore. Solitamente il contenitore fornisce i metodi per creare iteratori su di esso. Lo scopo primario di un iteratore è di consentire al codice che ne fruisce di visitare ogni elemento di un contenitore senza dipendere dalla struttura interna e dai dettagli di implementazione del contenitore stesso. Questo permette di riutilizzare, con variazioni minime, il codice che accede ai dati. È possibile cioè modificare o sostituire un contenitore con uno di struttura diversa senza compromettere la correttezza del codice che visita i suoi elementi. In seguito vedremo un iteratore per la classe lista.
Un iteratore deve essere una classe indipendente dalla classe lista in maniera tale che, chi vuole fare una iterazione sulla lista genera l'iteratore. Tuttavia, dall'altro latro vi è l'esigenza di avere la classe lista visibile all'interno della classe iteratore. Siccome una classe interna ha la visibilità degli attributi della classe che la contiene si sfrutta questa caratteristica della programmazione ad oggetti per modellare l'aspetto prima detto (ossia la classe iteratore deve avere la visibilità degli attributi della classe lista). In base a quanto detto dunque, la classe iteratore sarà una classe interna alla classe lista. Infine, siccome l'oggetto della classe interna non è visto all'esterno non si possono chiamare su di esso gli opportuni metodi. L'accorgimento che si usa è quello di far partecipare la classe iteratore all'implementazione di una interfaccia. In questo modo nell'interfaccia sono visibili i metodi della classe interna.

```
/* Classe interna per l'iteratore */
public class List_Iterator implements L_Iterator
{
        /* Campi dell'istanza */
        private int cursore;
        private Nodo elementoCorrente;

        /* Costruttore */
        public List_Iterator()
        {
              cursore=size-1;
```

```
                elementoCorrente=header;
            }

            /* Metodo hasNext */
            public boolean hasNext()
            {
                if (cursore<0)
                    return false;
                else
                    return true;
            }

            /* Metodo che punta al successivo elemento */
            public Object next()
            {
                Object elemento=null;
                if (cursore>=0)
                {
                    elemento=elementoCorrente.elemento;
                    cursore--;
                    elementoCorrente=elementoCorrente.next;
                }
                return elemento;
            }
    }
}

/* Interfaccia */
interface L_Iterator
{
      Object next();
      boolean hasNext();
}
```

9.1 Realizzazione del tipo dati astratto Dizionario

Nei paragrafi che seguiranno daremo le definizioni di tre diversi tipi di dato astratto, per ciascuno si mostreranno le azioni tipiche e l'implementazione. Per permettere un certo tipo di azioni si dovranno prevedere nelle classi di base `Nodo` e `Lista` altri metodi, ciò sarà fatto immediatamente dopo la presentazione dei TDA. Il TDA Dizionario è un insieme S di coppie `(elemento,chiave)`, operazioni tipiche sono l'inserimento, la cancellazione e la ricerca:

```
class Dizionario extends Lista
{
      void insert(Object o)
      {
            addOrd(o);
      }
      void delete(Object o)
      {
            remove(o);
      }
      Object search(Object o)
      {
            return search(o);
      }
}
```

9.2 Realizzazione del tipo dati astratto Pila

Il TDA Pila modella bene una struttura dati che segue la politica LIFO, last-in first-out. Operazioni tipiche di questa struttura sono: gli inserimenti in pila detti `push` che aggiungono elemento alla fine

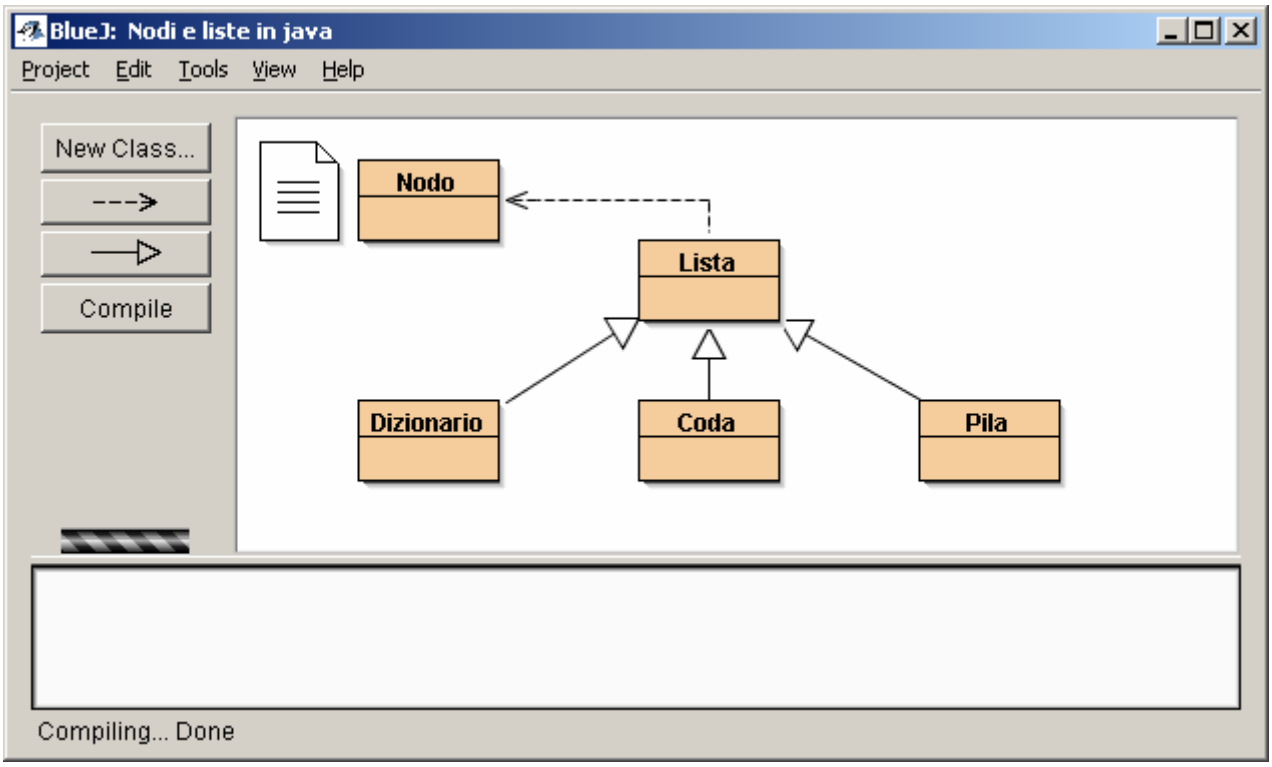
della sequenza di oggetti; la cancellazione dell'ultimo elemento inserito detta operazione di pop; Concludiamo dicendo dunque che in un pila gli accessi avvengono sempre ad una estremità della sequenza e nessun elemento interno può essere estratto prima che tutti quelli che lo seguono siano stati estratti.

```
class Pila extends Lista
{
	public boolean isEMPTY()
	{
		return isEmpty();
	}
	public void push(Object oggetto)
	{
		add(oggetto);
	}
	public Object pop()
	{
		return remove();
	}
	public Object top()
	{
		return get(0);
	}
}
```

9.3 Realizzazione del tipo dati astratto Coda

Il TDA Coda modella invece le strutture dati che seguono la politica di tipo FIFO, vale a dire first-in first-out. La differenza con il TDA precedente è che qui gli inserimenti, detti enqueue, avvengono ad una estremità della sequenza, mentre l'estrazione avviene all'estremità opposta della sequenza con operazioni di dequeue. Le cancellazioni rimuovono sempre il primo elemento.

```
class Coda extends Lista
{
	boolean isEmpty()
	{
		return isEmpty();
	}
	void enqueue(Object o)
	{
		add(o);
	}
	Object dequeue()
	{
		return removeTail();
	}
	Object first()
	{
		return getTail();
	}
}
```

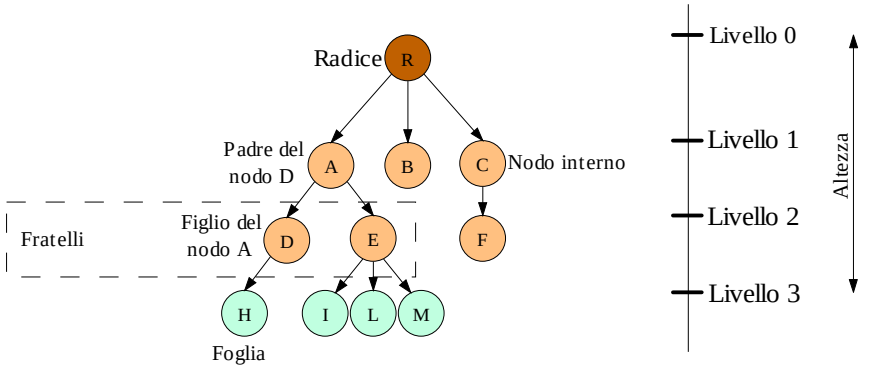


Abbastanza significativa è la figura riportata sopra che illustra le dipendenze delle strutture dati fin qui discusse usando un formalismo tipico dell'UML. A partire dalla classe `Nodo` si osserva infatti la dipendenza della classe `Lista` (linea tratteggiata) da quest'ultima. I TDA `Dizionario`, `Coda` e `Pila` invece possono essere viste come delle generalizzazioni della classe `Lista` la quale mette a disposizione di queste tutta una serie di metodi per implementarle e gestirle.

9.4 Alberi

Accade spesso di manipolare collezioni di oggetti in maniera anche abbastanza naturale, tutto ciò è reso possibile da una naturale relazione gerarchica che è stabilita fra gli oggetti. Ad esempio, ogni persona è in relazione ad un'altra seconda una relazione di discendenza che porta alla definizione di genitore, figli, fratelli e vari discendenti… (albero genealogico). La struttura dati che ci apprestiamo a studiare meglio modella l'esempio visto prima ed è particolarmente usato qualora esiste una relazione gerarchica fra gli oggetti di una lista. La definizione rigorosa di albero definisce l'albero come la coppia $T=(N,A)$ costituita da N insieme di nodi e da un insieme $A \subseteq N \times N$ che definisce l'insieme degli archi (un arco è comunque appoggiato a due nodi, l'insieme degli archi è quindi il prodotto cartesiano dell'insieme N con se stesso). In seguito daremo una definizione ricorsiva di albero e ciò ci consentirà anche una facile implementazione in java. Anche se la parola stessa della struttura dati si lascia paragonare all'albero in natura occorre subito precisare che gli alberi di cui parleremo hanno per semplicità un verso di crescita che va dall'alto verso il basso. A parte questa piccola differenza possiamo quindi accostare il concetto di albero, che ci apprestiamo a definire, ad un vero e proprio albero. Le definizioni ed i termini che lo caratterizzano derivano appunto da questo paragone che la natura ci offre.

Si definisce radice di un albero il nodo di più alto livello che non ha genitore. Ogni nodo, tranne la radice, ha un solo genitore. Un nodo può avere zero o più figli ed il numero di figli da il grado al nodo (nel corso di queste note tratteremo alberi binari e quindi nodi che hanno al più due figli). Un nodo senza figli è anche chiamato foglia, mentre i nodi che non sono né foglie e né radice si dicono nodi interni. Il figlio di un nodo può anche essere definito come la radice di un suo sottoalbero La profondità (o livello) di un nodo è definita come il numero di archi che bisogna percorrere per giungere alla radice. Da questa ultima definizione definiamo l'altezza di un albero come la massima profondità a cui si trova una foglia. La seguente figura riassume quanto finora detto:



Definizione ricorsiva di albero:

- Un albero è un albero vuoto;
- Un albero è insieme di N-alberi;

La prima ipotesi di albero costituisce il caso base o condizione di fuga di una definizione ricorsiva, l'albero è vuoto e coincide con il nodo che come vedremo in seguito lo definisce. La seconda ipotesi è invece più generica è permette la definizione di albero come insieme di tanti sottoalberi. Tuttavia, prima di affrontare la definizione e quindi anche l'implementazione in java della struttura dati albero e della classe nodo che la definisce motiviamo la necessità di una struttura dati idonea (quale è appunto l'albero) che non sia sempre e solo il solito array di oggetti.

Un albero potrebbe essere implementato con un array in cui ogni nodo dell'albero occupa una cella dell'array, tuttavia la stessa cella di array deve poi ospitare informazioni associati al nodo quali ad esempio l'eventualità di un arco verso un figlio. In seguito vedremo due soluzioni a questo problema che definiscono il vettore padre ed il vettore posizione come alternativa all'albero. Sebbene di facile

realizzazione, le rappresentazioni basate su array rendono tipicamente difficoltoso l'inserimento e la cancellazione di nodi nell'albero.

Vettore padri: immaginiamo di voler rappresentare un albero *T=(N,A)* di *n* nodi mediante un vettore dotato di celle che vanno dall'indice *0* all'indice *n-1*. Ad ogni cella dell'array è associato un contenuto informativo legato al singolo nodo più un contenuto informativo che descrive la discendenza o la relazione di gerarchia. Ad esempio, se V è il vettore, con *V[0].info* si accede al campo *info* del nodo 0 mentre con l'istruzione *V[0].parent* si accede alla dipendenza gerarchica del nodo 0. E' opportuna osservare che una tale dipendenza può essere trovata in un nodo solo se ad esso confluisce un arco, in caso contrario si può ad esempio trovare un riferimento a *null*.

Vettore posizione: : immaginiamo di voler rappresentare un albero *T=(N,A)* di *n* nodi mediante un vettore dotato di celle che vanno dall'indice *0* all'indice *n-1*. Per fare ciò si può usare una rappresentazione indicizzata dove ogni nodo ha una posizione prestabilita all'interno dell'array e tale che *V[i]* contiene l'informazione associata al nodo mentre *V[D·i]* intercetta l'informazione associata all'i-esimo figlio (*D* è il grado dell'albero, nel caso di albero binario *D=2* ed allora se alla posizione caratterizzata dall'indice *i=2* è associata l'informazione di un nodo, alla posizione caratterizzata dall'indice *i=2·i* , e dunque 4, si intercetta l'informazione del figlio, è come l'albero heap usato dall'algoritmo di ordinamento heapSort).

Se dunque riusciamo a rappresentare un albero anche con un array si deve comunque riconoscere l'enorme problema che una struttura sequenziale quale è l'array impone e che è la dimensione massima di elementi rappresentabile che rimane comunque limitata. Con una struttura dati collegata la rappresentazione collegata di alberi è più flessibile ed in ogni momento si può sempre aggiungere un nodo ad un albero, cosa impossibile da fare ad un vettore quando questo ha raggiunto la sua capienza massima.
In una rappresentazione collegata, così come visto per la classe *lista*, si definisce un record contenente l'informazione del nodo più altri puntatori che consentono di raggiungere altri nodi dell'albero. In alcuni casi, per permettere la risalita al padre di un figlio in tempo costante si usa ampliare ulteriormente il record prima descritto aggiungendo ad esso un ulteriore puntatore al padre. L'implementazione di albero e di nodo per la classe albero che vedremo non considerano questa eventualità che può comunque essere aggiunta in ogni momento.

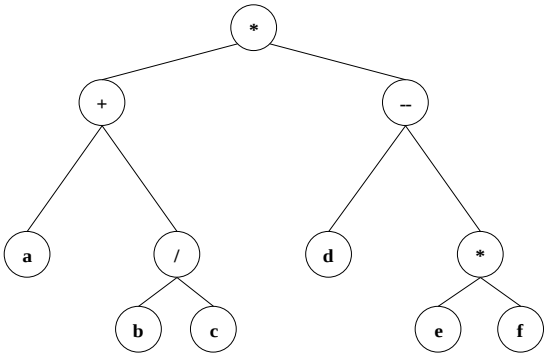
```
class Albero
{
      Nodo radice;
      public Tree(Object elemento)
      {
            radice=Nodo(elemento);
      }
}

class Nodo
{
      Object elemento;
      Nodo left;
      Nodo right;
      public Nodo(Object elemento)
      {
            this.elemento=elemento;
            left=right=null;
      }
}
```

In seguito avremo modo di aggiungere alle classi basi sopra illustrate i metodi che ne consentono una gestione efficiente. In molti casi è necessario attraversare o scansionare un albero visitandone tutti i

nodi. Tale operazione, a differenza delle collezioni di oggetti o liste in cui ciò avveniva in modo lineare, nel caso degli alberi è più articolata e bisogna seguire tutti i rami possibili a partire dalla radice. Anche se è possibile implementare una versione iterativa di un algoritmo di scansionamento in seguito definiremo alcuni metodi ricorsivi di attraversamento per la classe albero.

Visita in pre-ordine:
Nella classe nodo:

```
public void visitaPRE()
{
      System.out.println(elemento);
      if(left!=null) left.visitaPRE();
      if(right!=null) right.visitaPRE();
}
```



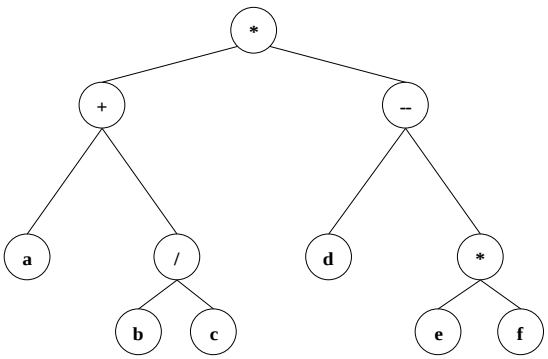
Data l'albero in figura, la visita in pre-ordine provocherebbe la seguente stampa:

$$*(+(a,/(b,c)),-(d,*(e,f)))$$

Si visita prima la radice, poi si effettuano le chiamate ricorsive sul figlio sinistro e destro.

Visita in-ordine
Nella classe nodo:

```
public void visitaORD()
{
      if(left!=null) left.visitaORD();
      System.out.println(elemento);
      if(right!=null) right.visitaORD();
}
```



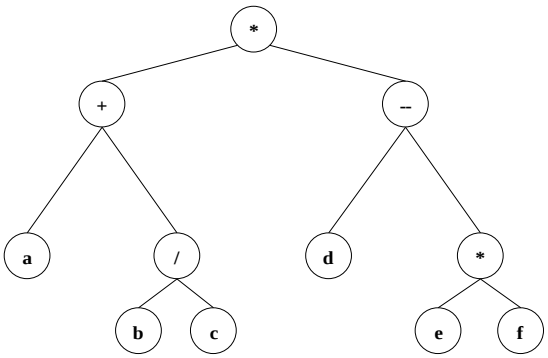
Data l'albero in figura, la visita in-ordine provocherebbe la seguente stampa:

$$(a+b/c)*(d-e*f)$$

Si visita prima il figlio sinistro effettuando la chiamata ricorsiva su di esso, poi la radice e successivamente il figlio destro.

Visita in post-ordine
Nella classe nodo:

```
public void visitaPOST()
{
      if(left!=null) left.visitaPOST();
      if(right!=null) right.visitaPOST();
      System.out.println(elemento);
}
```



Data l'albero in figura, la visita in post-ordine provocherebbe la seguente stampa:

$$abc/+def*-*$$

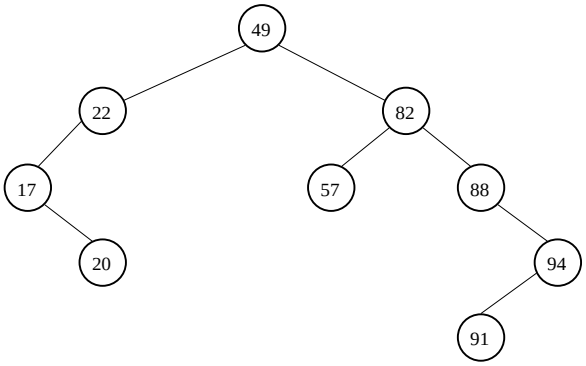
Si effettuano prima le chiamate ricorsive sul figlio sinistro e destro, e infine si visita la radice.

9.5 Alberi binari di ricerca

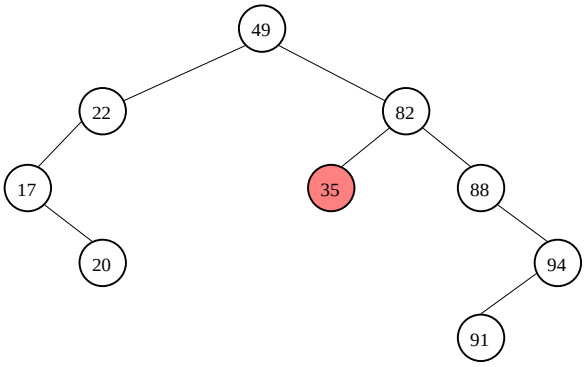
Ancora prima di introdurre il concetto di albero binario di ricerca occorre un momento ricordare l'algoritmo di ricerca binario visto per un array ordinato di elemento. In quella circostanza infatti si confrontava l'elemento da cercare con l'elemento nella posizione n/2 (con n dimensione massima del vettore) e se questo era minore di tale elemento si procedeva in maniera ricorsiva ad applicare l'algoritmo alla prima metà di sinistra del vettore, se invece l'elemento di confronto era maggiore dell'elemento "perno" l'algoritmo veniva applicato alla seconda metà del vettore.
Avendo introdotto la struttura dati di albero possiamo adesso associare al vettore ordinato un albero di ricerca in cui la radice contiene il valore nella posizione n/2 mentre i valori più piccoli e più grandi di n/2 sono assegnati rispettivamente ai sottoalbero sinistro e destro della radice. In tal caso una visita in ordine produrrà come risultato la stampa del vettore dall'elemento più piccolo a quello più grande. Una definizione più rigorosa di albero è la seguente:

- Ogni nodo V di un albero di ricerca contiene un elemento a cui è associata una chiave presa da un dominio ordinato;
- Le chiavi nel sottoalbero sinistro di V sono minori o al più uguali alla chiave di V;
- Le chiavi nel sottoalbero destro di V sono maggiori o al più uguali alla chiave di V;

Ad esempio, l'albero che segue rispetta le proprietà appena dette ed è pertanto un albero di ricerca:



Mentre l'abero quest'altro che segue in figura non è un albero di ricerca poiché il nodo con chiave35 non rispetta la proprietà di ricerca rispetto alla radice:



Tale struttura dati ci permette di implementare l'operazione di ricerca in maniera molto semplice. E' infatti necessario confrontare la chiave da cercare con la radice dell'albero, se questa è uguale alla chiave allora vuol dire che abbiamo trovato l'elemento (che fortuna…) altrimenti si prosegue la ricerca al sottoalbero sinistro (nel caso in cui la chiave dell'elemento da cercare è minore o uguale alla chiave della radice) oppure al sottoalbero destro (nel caso in cui la chiave dell'elemento da cercare è maggiore o uguale alla chiave della radice). Questo algoritmo ha solitamente un tempo O(h) se h è l'altezza dell'albero. Se però l'albero è profondo e sbilanciato allora la complessità si avvicina ad O(n) ed in tal caso è equivalente usare una lista o un array disordinato.

9.6 Alberi AVL

Se dunque un albero risulta essere profondo e sbilanciato si rischia di ottenere prestazioni simili a quelle di un array disordinato. E' allora opportuno non solo poter conferire ad un albero una struttura bilanciata ma mantenerne anche le proprietà di bilanciamento nel tempo a seguito di operazioni di inserimento e cancellazioni. Occorre tuttavia dare una definizione di bilanciamento (in altezza): Un albero è bilanciato in altezza se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono di al più un unità.

Gli alberi binari di ricerca sono anche detti alberi AVL, dai nomi degli ideatori Adel'son-Vel'skii e Landis, che li hanno introdotti nel 1962. In un albero AVL, oltre all'elemento e alla chiave, ciascun nodo mantiene anche un informazione sul bilanciamento così definita. Si definisce poi fattore di bilanciamento B(v) di un nodo v la differenza tra l'altezza del sottoalbero sinistro e quella del sottoalbero destro di v:

$$\beta(v) = altezza(sx(v)) - altezza(dx(v))$$

Il fattore di bilanciamento p tanto migliore quanto più basso è il suo valore assoluto, mentre è uguale all'altezza dell'albero quando questo degenera in lista. In particolare, in un albero binario completo il fattore di bilanciamento è 0 su ogni nodo. Un albero binario di ricerca come quello prima introdotto è quindi anche un albero AVL in base alla definizione adesso data.

9.7 Bilanciamenti tramite rotazioni

Siccome a causa di cancellazioni o inserimenti la proprietà di bilanciamento per un albero può essere persa sono previsti dei metodi che ripristinano tale proprietà effettuando delle opportune rotazioni. La rotazione avviene nei confronti di un nodo usato come perno e possono andare o verso sinistra oppure verso destra. I due casi sono simmetrici tra loro.
Le rotazioni mantengono le proprietà di bilanciamento se applicate a nodi bilanciati ma come è normale pensare esse vanno applicate a quei nodi che invece risultano sbilanciati e che quindi hanno un coefficiente di sbilanciamento che può essere di più o meno 2. In maniera del tutto generale in un albero sbilanciato esisterà un sottoalbero di v che è troppo alto, ciò vuol dire che il sottoalbero è talmente profondo da sbilanciare il nodo delll'albero. Un ipotetico sottoalbero che sbilancio un nodo può stare nei quattro casi elencati:

- Sinistra – sinistra (SS): T è il sottoalbero sinistro del figlio sinistro di v;
- Destra – destra (DD): T è il sottoalbero destro del figlio destro di v;
- Sinistra – destra (SD): T è il sottoalbero destro del figlio sinistro di v;
- Destra – sinistra (DS): T è il sottoalbero sinistro del figlio destro di v.

I quattro casi sono simmetrici tra loro, più precisamente SS è simmetrico con il caso DD mentre SD è simmetrico con DX. Quindi i casi da analizzare sono solamente due.
Caso SS: per bilanciare il nodo v basterà applicare una rotazione semplice verso destra su v, il cui fattore di sbilanciamento passerà da 2 a 0;
Caso SC: questo caso è più complesso e richiede l'applicazione di una rotazione doppia. Sia z il figlio sinistro di v: l'albero T che sbilancia v è il sottoalbero destro di z, radicato nel nodo w. La rotazione SD consiste in una rotazione di base verso sinistra con perno in z seguita da una rotazione di base verso destra con perno in v.

10.1 Grafi: introduzione

Il grafo è una struttura dati particolarmente utile alla modellazione ed alla rappresentazione di connessioni e/o relazioni fra coppie di oggetti. La struttura dati grafo nasce come notazione matematica di supporto alla rappresentazione di proprietà e relazioni matematiche (ad esempio la biunivocità di una funzione) ma trova un ampio utilizzo in campo informatico. Un grafo viene denotato con la seguente notazione:

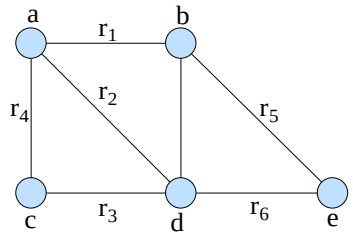
$$G = (V, E)$$

Esso consiste in un insieme V di vertici o nodi e di un insieme E di coppie di vertici detti archi.
Un arco, se presente nel grafo, simboleggia una relazione fra i nodi del grafo. Ad ogni nodo può essere data una informazione a seconda del problema che si sta modellando. Pertanto, un nodo (oppure anche un vertice) rappresenterà un oggetto qualunque e gli archi saranno le relazioni tra questi oggetti (ad esempio un insieme di nodi può rappresentare delle persone, gli archi invece potrebbero simboleggiare le strette di mano avvenute fra due o più persone). L'insieme degli archi di un grafo è contenuto nel prodotto cartesiano dell'insieme V con se stesso:

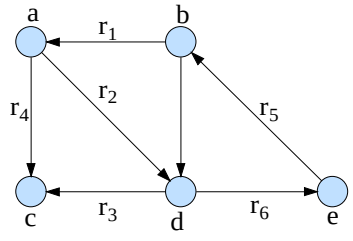
$$E \subseteq V \times V$$

Qualora la relazione fra due nodi implica un ruolo o comunque lascia intendere un verso di percorrenza per la relazione stessa, si dirà che il grafo è di tipo orientato. In un grafo orientato ogni arco è orientato da uno dei due vertici verso l'altro e si disegna con una freccia che ne indica la direzione. Nel corso di questo note chiameremo nodi tutti i vertici di un grafo ed indicheremo con n il numero di nodi mentre con m indicheremo il numero di archi. Di seguito sono mostrate delle figure che mostrano l'attuale notazione usata per disegnare un grafo. Altre figure invece stimolano lo studio nei confronti dei grafi mostrando alcuni dei casi in cui essi vengono utilizzati. Per la figura che segue:

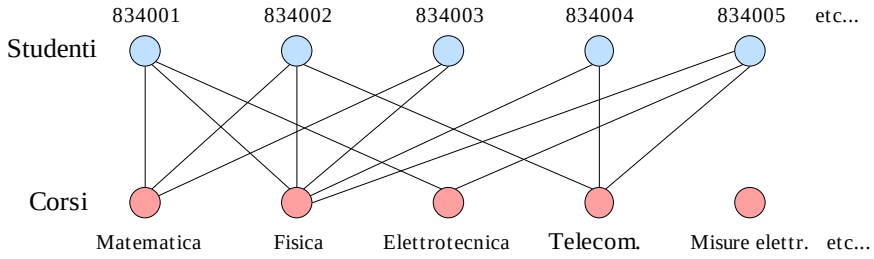
$$V = \{a, b, c, d, e\}$$
$$E = \{r_1, r_2, r_3, r_4, r_5, r_6\}$$



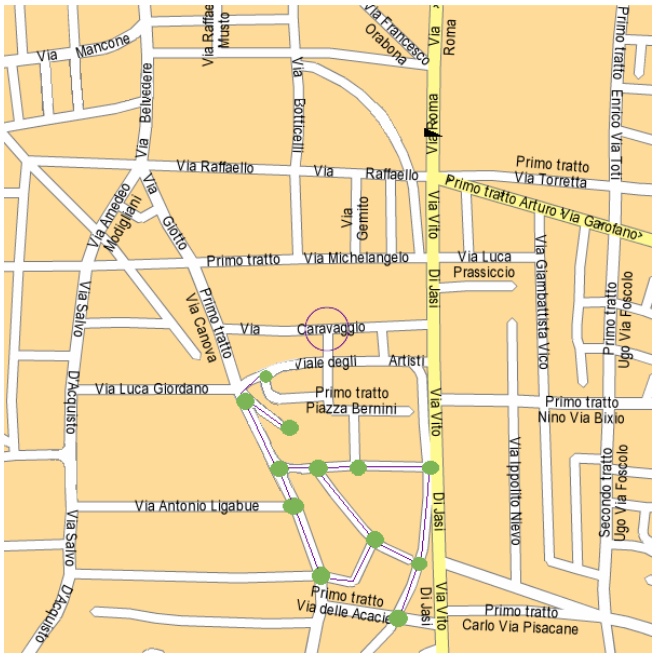
Mentre un esempio di grafo orientato è il seguente:



In quest'ultimo esempio gli archi sono orientati e le frecce ne simboleggiano il verso. Un esempio di grafo può ad esempio essere quello in cui vengono riassunte le relazioni fra gli studenti di un ateneo e gli esami di un corso di laurea:



Un esempio di grafo è ad esempio quello in cui i nodi rappresentano un incrocio e gli archi le vie che collegano gli incroci come viene mostrato nella figura che segue:



Le bolle di colore rappresentano un incrocio, gli di colore viola cono le strade che collegano gli incroci. Nel corso di queste note vedremo in seguito come determinare il cammino minimo che ci conduce da un nodo ad un altro. Problemi di questo tipo sono risolti in maniera agile ed elegante se si decide di adottare un grafo come struttura dati per modellare la realtà del problema in esame.

10.2 Alcune definizioni

Adiacenza ed incidenza

Il concetto di adiacenza indica quei nodi immediatamente vicini ad un nodo considerato. Dato poi un arco (x, y) di un grafo G non orientato, si dice che esso incide sui nodi x ed y che quindi sono tra loro adiacenti. Mentre, se il grafo è orientato si dirà che l'arco esce dal nodo x ed entra nel nodo y. Preso dunque un nodo qualunque v del grafo, chiameremo vertici adiacenti al nodo v quelli che gli sono immediatamente vicini e quindi ad esso collegati.

Grado di un vertice

Il grado di un vertice è la somma degli archi che ad esso confluiscono. Il grado di un vertice si denota come $\delta(v)$. In un grafo non orientato, se sommiamo tutti gradi dei vertici conteremo un arco esattamente due volte: una volta quando consideriamo il grado di x ed una volta quando consideriamo il grado di y.

$$\sum_{v \in G} \delta(v) = 2m$$

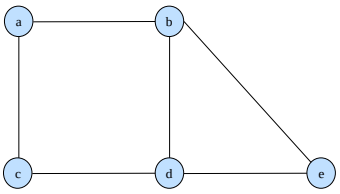
Se invece il grafo è orientato, si può allora parlare di grado in entrata e di grado in uscita per un nodo qualunque v del grafo. La notazione usata per denotare i gradi in entrate ed in uscita è la seguente: $\delta_{in}(v)$ e $\delta_{out}(v)$.

Cammini e cicli

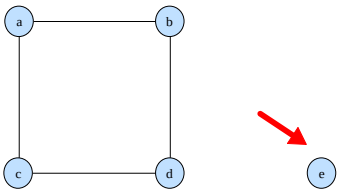
Per cammino si intende una N-pla di vertici tale che per ogni coppia vertice esiste un arco che li connette. Più in particolare, un cammino in un grafo G da un vertice x ad un vertice y è dato da una sequenza di vertici $(v_0, v_1, ..., v_k)$ con $v_0 = x$ e $v_k = y$, tale che per $1 \le i \le k$ l'arco (v_{i-1}, v_i) appartiene a G. Se esiste un cammino da x ad y diremo anche che y è raggiungibile da x. La lunghezza di un cammino è la distanza da percorrere affinché si arrivo al nodo di destinazione. Si dice poi ciclo quel cammino che ha come nodo di origine e nodo di destinazione lo stesso nodo (altra definizione di ciclo: un ciclo è un cammino per il quale il primo nodo è anche l'ultimo e la lunghezza del cammino sia necessariamente maggiore di 0). Il ciclo può essere semplice se il cammino non passa per i nodi per più di una volta, in caso contrario il cammino si dirà non semplice. Un cammino che non sia semplice e ciclico si dice aciclico.

Connettività

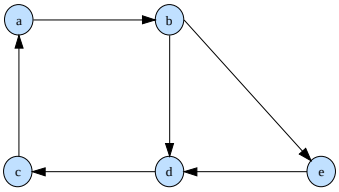
Un grafo non orientato si dice connesso se esiste un cammino tra ogni coppia di vertici in G. Ogni grafo per essere connesso deve avere almeno (n-1) archi. Un grafo orientato G si dice fortemente connesso se esiste un cammino (orientato) tra ogni coppia di vertici in G.



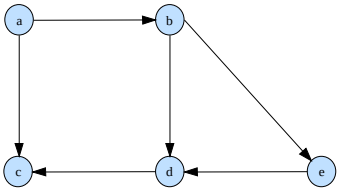
Grafo connesso



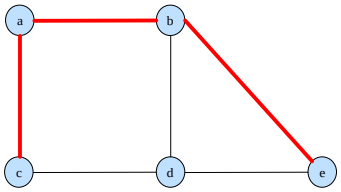
Grafo non connesso



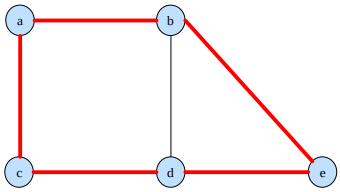
Grafo orientato connesso



Grafo orientato non connesso



Cammino={(c,a),(a,b),(b,e)}
Lunghezza percorso=3
Cammino semplice
δ(a)=2
δ(b)=3
δ(c)=2
δ(d)=3
δ(e)=2



Ciclo={(c,a),(a,b),(b,e)}
Lunghezza percorso=5
Cammino semplice

Riassumendo:
Se G è un grafo orientato:

- In un grafo non orientato il grado di un vertice è il numero di archi che da esso si dipartono;
- In alcuni casi ogni arco ha un peso (o costo) associato.
- Un percorso nel grafo è una sequenza di vertici;
- La lunghezza del percorso è il numero totale di archi che connettono i vertici nell'ordine della sequenza;
- Un percorso si dice semplice se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza), eccetto al più il primo e l'ultimo che possono coincidere;
- Se esiste un percorso p tra i vertici v e w, si dice che w è raggiungibile da v tramite p;
- Se G è un grafo non orientato, diciamo che G è connesso se esiste un percorso da ogni vertice ad ogni altro vertice.
- Un ciclo in un grafo è un percorso di lunghezza almeno pari ad;
- Un grafo senza cicli è detto aciclico;

Se G è un grafo non orientato:

- In un grafo orientato, un arco (w,v) si dice incidente da w in v;
- In un grafo non orientato la relazione di adiacenza tra vertici è simmetrica;
- In un grafo orientato il grado entrante (uscente) di un vertice è il numero di archi incidenti in (da) esso;

- In un grafo orientato il grado di un vertice è la somma del suo grado entrante e del suo grado uscente;
- In alcuni casi ogni arco ha un peso (o costo) associato.
- Un percorso nel grafo è una sequenza di vertici;
- La lunghezza del percorso è il numero totale di archi che connettono i vertici nell'ordine della sequenza;
- Un percorso si dice semplice se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza), eccetto al più il primo e l'ultimo che possono coincidere;
- Se esiste un percorso p tra i vertici v e w, si dice che w è raggiungibile da v tramite p;
- Se G è un grafo orientato, diciamo che G è fortemente connesso se esiste un percorso da ogni vertice ad ogni altro vertice;
- Un ciclo in un grafo è un percorso di lunghezza almeno pari ad 1;
- Un grafo senza cicli è detto aciclico;

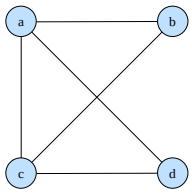
10.3 Come rappresentare un grafo

Non esiste un'unica rappresentazione per un grafo ed a seconda del problema si può preferire una rappresentazione piuttosto che un'altra. Ogni tipo di rappresentazione offre dei svantaggi. Ciascuna delle rappresentazioni che vedremo viene confrontata con le altre in base ai tempi di esecuzione delle operazioni su grafi. Per poter fare ciò è allora necessario introdurre le possibili operazioni che una struttura dati rappresentante un grafo deve implementare, ecco un elenco dei metodi più utilizzati:

```
tipo dato Grafo
dati:
   • un insieme di vertici V;
   • un insieme di archi E;
operazioni:
   • numVertici()
     Restituisce il numero intero di vertici presenti nel grafo;
   • numArchi()
     Restituisce il numero di archi presenti nel grafo;
   • grado(vertice v)
     Restituisce il numero di archi incidenti sul vertice v;
   • archiIncidenti(vertice v)
     Restituisce una lista di archi incidenti su v;
   • estremi(arco e)
     Restituisce gli estremi x ed y dell'arco e;
   • opposto(vertice v, arco e)
     Restituisce y, l'estremo dell'arco e=(x,y) diverso da x;
   • sonoAdiacenti(vertice x, vertice y)
     Restituisce true se esiste l'arco e(x,y), false altrimenti;
   • aggiungiVertice(vertice v)
     Inserisce un nuovo vertice v nel grafo;
   • aggiungiArco(vertice x, vertice y)
     Inserisce un nuovo arco tra i vertici x ed y;
   • rimuoviVertice(vertice y)
     Cancella il vertice e tutti gli archi ad esso incidenti;
   • rimuoviArco(arco e)
     Cancella l'arco e;
```

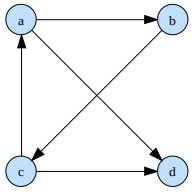
Rappresentazione di grafi con lista di archi

La rappresentazione di un grafo per lista di arco si basa sulla definizione di grafo. Adottando una struttura dati per modellare i vertici di un grafo ed un'altra struttura dati per modellare gli archi di ques'ultimo si può rappresentare un grafo mediante una lista di archi. Lo spazio totale usato da questa rappresentazione è $O(m+n)$ visto che serve una quantità di spazio per ogni vertice e per ogni arco del grafo. Ad esempio:



(a,b)
(c,a)
(b,c)
(c,d)
(a,d)

Grafo non orientato G Lista di archi di G



(a,b)
(c,a)
(b,c)
(c,d)
(a,d)

Grafo orientato G Lista di archi di G

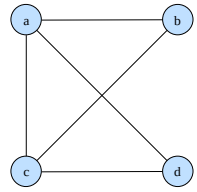
Questo tipo di rappresentazione implica un certo tipo di prestazioni che riassumiamo nella seguente tabella:

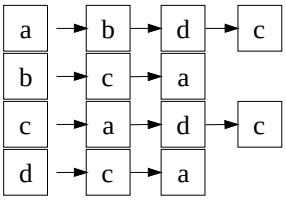
TABELLA: Tempi di esecuzione delle operazioni su grafi non orientati rappresentati con lista di archi

Operazione	Tempo di esecuzione
grado(v)	*O(m)*
archiIncidenti(v)	*O(m)*
sonoAdiacenti(x,y)	*O(m)*
aggiungiVertice(v)	*O(1)*
aggiungiArco(x,y)	*O(1)*
rimuoviVertice(v)	*O(m)*
rimuoviArco(e)	*O(m)*

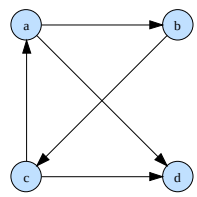
Molte delle operazioni richiedono l'esame dell'intera lista di archi e ciò richiede un tempo che nel peggiore dei casi è *O(m)*. Anche l'accesso ad un arco avviene con lentezza.

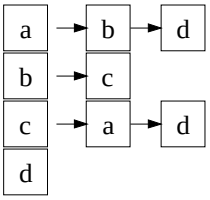
Rappresentazione di grafi con liste di adiacenza
In questa rappresentazione ogni vertice *v* del grafo ha una propria lista di adiacenza in cui sono elencati tutti i vertici *u* per cui esiste un arco *(v,u)*. Ad esempio:





Grafo non orientato G Lista di adiacenza di G





Grafo orientato G Lista di adiacenza di G

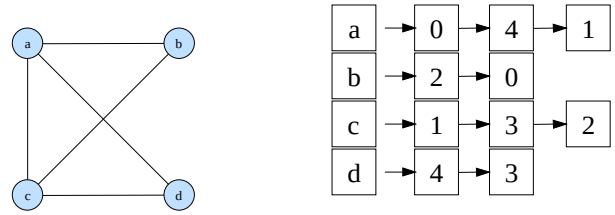
Se n è il numero di vertici allora n sarà anche il numero di liste di adiacenza. Con questo metodo di rappresentazione diventa molto più semplice trovare gli archi connessi ad un particolare vertice v (è sufficiente infatti scorrere la lista di adiacenza del nodo v). Tuttavia, se da un lato questa rappresentazione risulta essere efficiente (lo è infatti nell'applicazioni in cui si richiede di visitare un grafo esaminando gli archi incidenti sui suoi vertici), dall'altro mostra i suoi limiti nella verifica su di una particolare coppia di vertici x ed y come possibile arco. In tal caso infatti è opportuno scandire due liste di adiacenza in parallelo, ad esempio quella del nodo x e quella del nodo y, e vedere un passo per volta se è possibile avere un arco $e=(x_n,y_n)$. Un operazione del genere richiede un tempo di esecuzione particolare che è indicato in tabella. Altra debolezza di una rappresentazione siffatta è la ridondanza di un arco, presente nelle liste di adiacenza per ben due volte (sarà infatti presente nella lista di x ed in quella di y).

TABELLA: Tempi di esecuzione delle operazioni su grafi non orientati rappresentati con liste di adiacenza

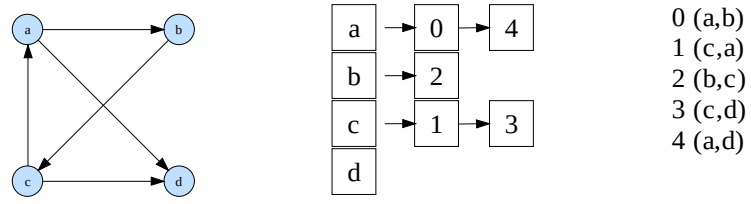
Operazione	Tempo di esecuzione
grado(v)	$O(\delta(v))$
archiIncidenti(v)	$O(\delta(v))$
sonoAdiacenti(x,y)	$O(min\{\delta(x),\delta(y)\})$
aggiungiVertice(v)	$O(1)$
aggiungiArco(x,y)	$O(1)$
rimuoviVertice(v)	$O(m)$
rimuoviArco(e)	$O(\delta(x)+\delta(y))$

Rappresentazione di grafi con liste di incidenza

Questo tipo di rappresentazione combina le caratteristiche delle prime due rappresentazioni finora viste. In aggiunta alla rappresentazione con lista di archi, ogni vertice *v* del grafo ha una lista di puntatori agli archi incidenti a *v*. Ad esempio:



Grafo non orientato G Lista di adiacenza di G

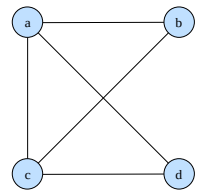


Grafo orientato G Lista di adiacenza di G

TABELLA: Tempi di esecuzione delle operazioni su grafi non orientati rappresentati con liste di incidenza

Operazione	Tempo di esecuzione
grado(v)	$O(\delta(v))$
archiIncidenti(v)	$O(\delta(v))$
sonoAdiacenti(x,y)	$O(min\{\delta(x),\delta(y)\})$
aggiungiVertice(v)	$O(1)$
aggiungiArco(x,y)	$O(1)$
rimuoviVertice(v)	$O(m)$
rimuoviArco(e)	$O(\delta(x)+ \delta(y))$

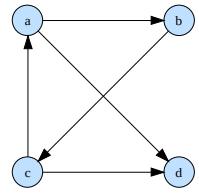
Rappresentazione di grafi con matrici di adiacenza
Si tratta di una rappresentazione che richiede più spazio poiché prevede la dichiarazione di una matrice *nxn* con n numero di vertici del grafo. In tal caso però la verifica della presenza di un arco può essere effettuata più velocemente. La matrice di adiacenza colloca un 1 in corrispondenza dell'arco (a,b) se esso appartiene al grafo, o altrimenti. Nel caso di grafi orientati la matrice di adiacenza è simmetrica. Possiamo quindi verificare, in questa rappresentazione, la presenza di un arco *(x,y)* con un tempo costante (basta verificare il valore in M[x,y] che se 1 indica appunto la presenza di un arco). Un operaziona alquanto scomodo è la verifica dei vertici adiacenti ad un nodo v e dei suoi vicini.



Grafo non orientato G

	a	b	c	b
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
b	1	0	1	0

Matrice di adiacenza di G



Grafo orientato G

a	0	1	0	1
b	0	0	1	0
c	1	0	0	1
b	0	0	0	0

Matrice di adiacenza di G

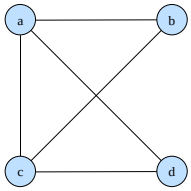
TABELLA: Tempi di esecuzione delle operazioni su grafi non orientati rappresentati con matrici di adiacenza

Operazione	Tempo di esecuzione
grado(v)	$O(n)$
archiIncidenti(v)	$O(n)$
sonoAdiacenti(x,y)	$O(1)$
aggiungiVertice(v)	$O(n^2)$
aggiungiArco(x,y)	$O(1)$
rimuoviVertice(v)	$O(n^2)$
rimuoviArco(e)	$O(1)$

In tale rappresentazione, l'aggiunta o la rimozione di un vertice implica una riallocazione delle matrice di adiacenza.

Rappresentazioni di grafi con matrici di incidenza
La matrice di incidenza (come quella usata nel corso di Elettrotecnica 1) è una matrice di dimensione *nxm* con n numero di vertici ed m numero di colonne. Le righe sono indicizzate appunto dai vertici del

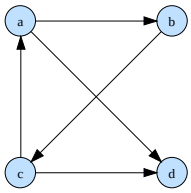
grafo mentre le colonne sono indicizzate sui possibili archi del grafo. Tale matrice avrà quindi un valore uguale ad 1 in corrispondenza del vertice e dell'arco a cui esso appartiene. Nel caso di grafi orientati i valori della matrice possono essere ±1 ed indicare quindi la direzione dell'arco. Ogni colonna avrà allora due soli valori in corrispondenza degli estremi di un arco. Una riga può invece avere due o più valori a seconda del vertice che può quindi afferire ad uno o più vertici. Ad esempio:



Grafo non orientato G

	a,b	c,a	b,c	c,d	a,d
a	1	1	0	0	1
b	1	0	1	0	0
c	0	1	1	1	0
d	0	0	0	1	1

Matrice di incidenza di G



Grafo orientato G

	a,b	c,a	b,c	c,d	a,d
a	1	-1	0	0	1
b	-1	0	1	0	0
c	0	1	-1	1	0
d	0	0	0	-1	-1

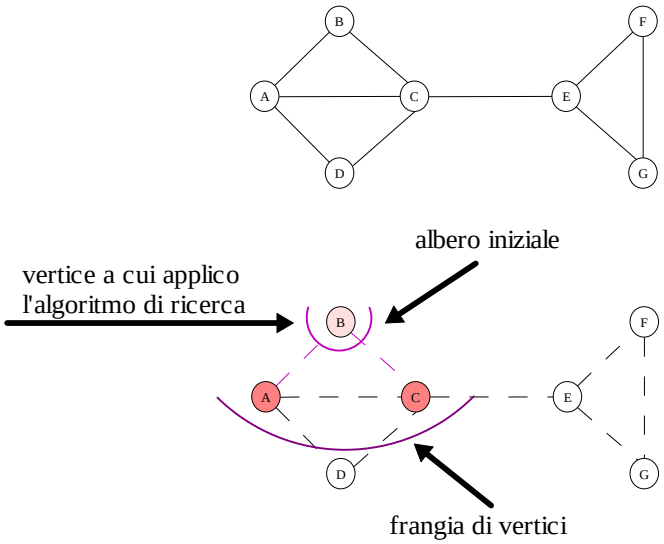
Matrice di incidenza di G

11 Visite di grafi

Nel corso di questo note ci occuperemo per prima di un algoritmo di visita "generico", successivamente andremo a particolarizzare l'algoritmo trovato specializzandolo in base al tipo di visita che effettua sul grafo. Una prima domanda a cui dobbiamo rispondere è quella che ci porta a chiedere il perché di una tale necessità. La risposta a questo motivo non è adesso immediata ma l'utilità di un buon algoritmo di visita per un grafo risiede nelle applicazioni e nell'uso che spesso si fa di un grafo come modello della realtà di riferimento in esame. I grafi possono modellare in maniera efficace moltissimi problemi, spesso l'attraversamento di un grafo si presenta come la base di un problema da risolvere. La visita di un grafo deve essenzialmente esaminare tutti gli archi e tutti i nodi di un grafo in modo sistematico.

Quando si visita un grafo occorre memorizzare i vertici già visitati, ciò è importante ed evita di visitare un vertice più volte. Per risolvere questo problema, che è certamente più complesso in grafi di grosse dimensioni, si usa associare al vertice un colore che ne rappresenta la marcatura, oppure un bit. Quando l'algoritmo visiterà un vertice per la prima volta cambierà il valore della marcatura. Se il grafo è poi connesso, il risultato di una visita su un grafo è un albero che tocca tutti vertici del grafo in un determinato ordine. E' proprio l'ordine ed il modo con cui sono attraversati i vertici che differenzia i vari algoritmi di visita.

Durante l'esecuzione dell'algoritmo di visita esiste un albero contenenti i vertici visitati fino a quel punto, pertanto, quando l'algoritmo viene richiamato su un vertice U del grafo l'albero iniziale è costituito dal solo vertice considerato. L'algoritmo procede esaminando tutti i vertici collegati ad U, il modo di esaminare tali vertici è un elemento discriminante per gli algoritmi di visita. L'algoritmo di visita "generico" sceglie in maniera casuale i vertici collegati ad U. L'insieme di vertici che si collegano direttamente all'albero (parziale) di visita costituiscono la "frangia". Nella figura che segue viene mostrato un esempio di frangia.



L'algoritmo di visita quindi procede analizzando i vertici trovati nella frangia. Per ciascuno di essi infatti, qualora il vertice non fosse stato già visitato si procede come segue: si marca per primo il vertice (colorandolo come prima detto), quindi si aggiunge il vertice all'albero di visita. Il risultato dell'algoritmo di visita sarà infine l'albero generato. Pseudo-codice dell'algoritmo di visita generico:

```
Algoritmo visitaGenerica(vertice s) → albero
      Rendi tutti i vertici non marcati
      T ← albero formato da un solo nodo s
      F ← insieme vuoto di vertici
      Marca il vertice s ed aggiungi s ad F
      While(F!=0)do
            Estrai un qualsiasi vertice u da F
            Visita il vertice u
            For each(arco(u,v) in G) do
                  If(v non è ancora stato marcato) then
                        Marca il vertice v e aggiunge v ad F
                        Rendi u padre di v in T
                  Else
                        Eventualmente rendi u nuovo padre di v in T
      Return T
```

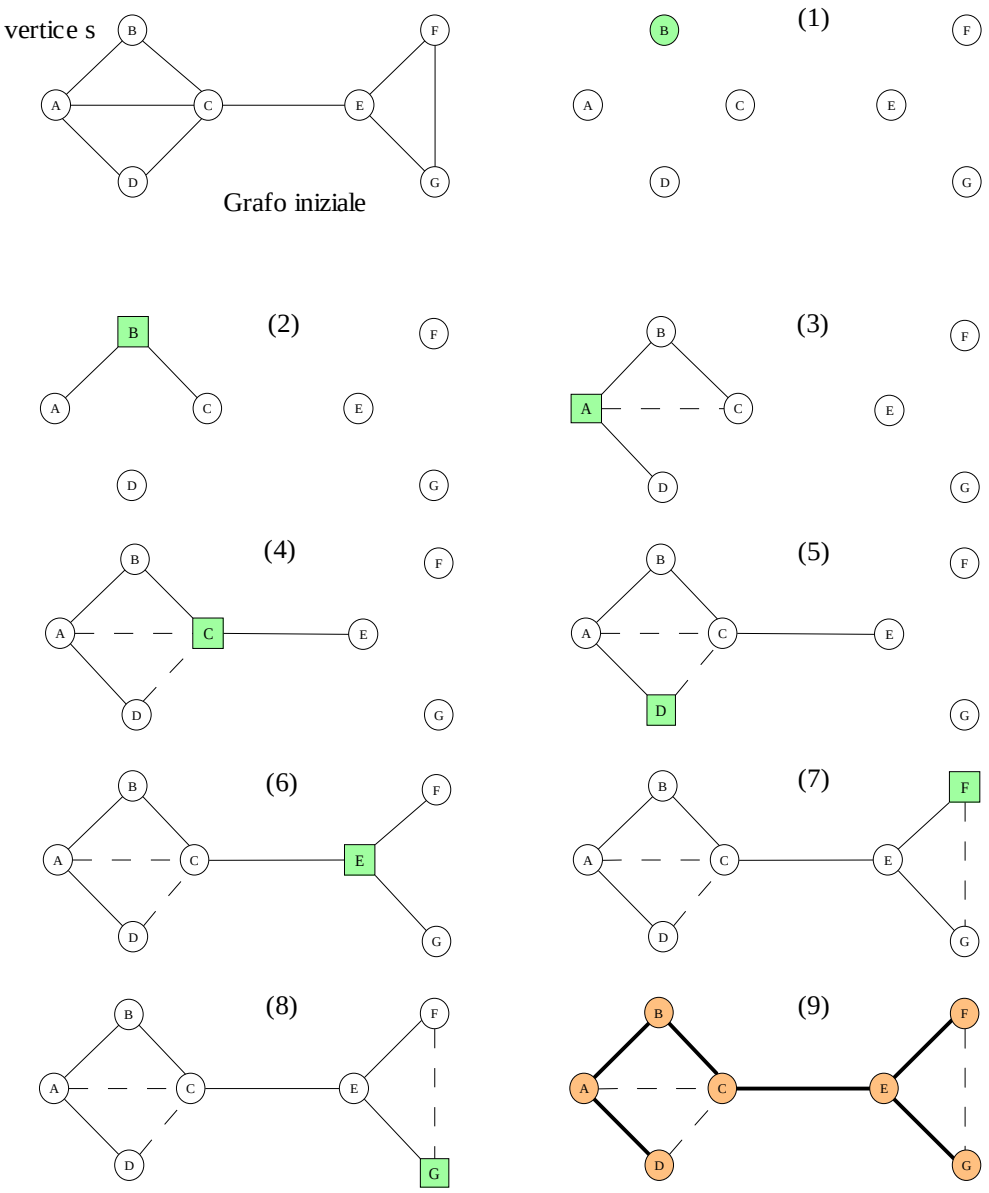
A partire dall'algoritmo di visita generico è possibile ottenere due varianti dell'algoritmo di visita. Infatti, l'ordine con cui abbiamo inserito i vertici e cancellato gli archi incidenti su T nell'insieme F è stato del tutto casuale (estrai un qualsiasi vertice u da F).
Se l'insieme F dei vertici appartenenti alla frangia è gestito con una struttura dati coda si ottiene una visita in ampiezza, detta breadth first search. Qualora si decidesse di utilizzare questa metodologia si otterrà allora una visita del grafo che a partire dal vertice s esamina per prima i vertici vicini ad s e genera un cammino minimo in termini di archi. Al contrario invece, se i vertici della frangia F che incidono sul vertice s (o sul all'albero T) sono gestiti da una struttura dati pila si ottiene una visita del grafo detta depth first search o visita in profondità.

Sia G=(V,E) un grafo non orientato. Una visita in ampiezza genera a partire da un vertice s un albero T che è detto albero BFS. I vertici della frangia sono inseriti in una coda, essi pertanto sono inseriti da un estremo e prelevati dall'altro estremo con operazioni tipiche della struttura dati utilizzata (enqueue e dequeue). Quando a partire dal vertice s si esamina la frangia F degli archi incidenti su T (che inizialmente è composto dal solo vertice s) essi sono inseriti nella coda nell'ordine con cui sono trovati. Al passo successivo pertanto si esamineranno prima i vertici uscenti dalla coda, potrebbe allora capitare che i successivi vertici in uscita non necessitano di una marcatura se risultano essere stati già

marcati in precedenza. Nell'esempio che segue i vertici della frangia F sono inseriti nella coda nell'ordine che tiene conto dell'etichette associate al vertice e quindi secondo l'ordine alfabetico.
Per ottenere una visita in ampiezza si modifica l'algoritmo di visita generico. In tal caso non è necessario rendere un vertice u padre di un vertice v (v è vertice della frangia) nell'albero T ad ogni passo, si tratta di un eventualità che può verificarsi solo in fase di marcatura. Infine, per l'implementazione si potrebbe poi fare riferimento ad un eventuale matrice di adiacenza per stabilire l'ordine con cui inserire i vertici v della frangia nella coda.
Prima di vedere graficamente i passi con cui un algoritmo di visita in ampiezza genera l'albero T accenniamo alcune importanti proprietà. Come nell'algoritmo di visita generico anche per quello in ampiezza si verifica la marcatura di un nodo solo una volta. Inoltre, gestendo i vertici v della frangia F con una coda ed indicando con $\ell(v)$ il livello di profondità di un vertice si può osservare che se:

$$F = \{v_1, v_2, ..., v_n\} \text{ allora } \ell(v_i) \leq \ell(v_{i+1})$$

Con i che varia da 1 ad n. In altre parole, man mano che l'algoritmo procede si analizzeranno vertici della frangia con livelli di profondità al più uguali o maggiori di quelli analizzato al passo precedente.



```
Algoritmo visitaBFS(vertice s) → albero
      Rendi tutti i vertici non marcati
      T ← albero formato da un solo nodo s
      Coda F
      Marca il vertice s
      F.enqueue(s)
      While(not F.isempty())do
            u ← F.dequeue()
            For each(arco(u,v) in G) do
                  If(v non è ancora stato marcato) then
                        F.dequeue(v)
                        Marca il vertice v
                        Rendi u padre di v in T
      Return T
```

Glia archi del grafo G possono essere poi classificati in tre gruppi rispetto all'albero BFS:

1. archi dell'albero BFS;
2. archi tra vertici allo stesso livello dell'albero BFS;
3. archi tra livelli adiacenti dell'alberi BFS;

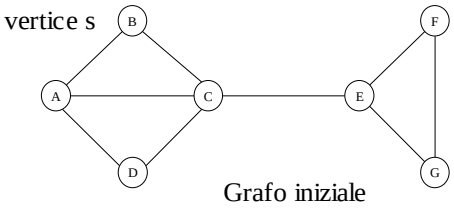
Una visita in profondità di G invece esamina i vertici del grafo generando ancora una volta un albero che stavolta chiameremo DFS (da depth fisrt search). La differenza in questo caso consiste nella gestione dei vertici v della frangia F secondo una struttura dati pila:

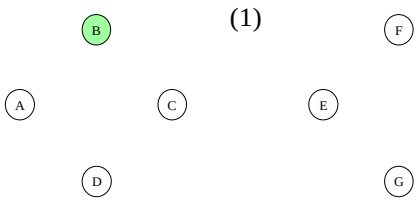
```
procedura visitaDFSRicorsiva(vertice v, albero T)
      marca e visita il vertice v
      for each(arco(u,w))do
            if(w non è marcato) then
                  aggiungi l'alrco (v,w) all'albero T
                  visitaDFSRicorsiva(w,T)

algoritmo visitaDFS(vertice s) → albero
      T ← albero vuoto
      visitaDFSRicorsiva(s,T)
      return T
```

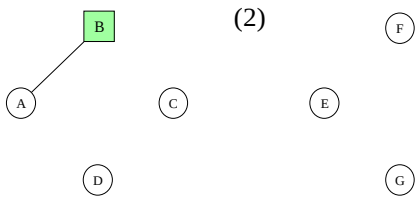
Ancora una volta i vertici vanno marcati una sola volta ed a tal proposito ci si può servire di sistema di marcatura che garantisca una sola marcatura per vertice. Infine, gli archi possono, anche in questo caso, essere classificati in base all'albero DFS:

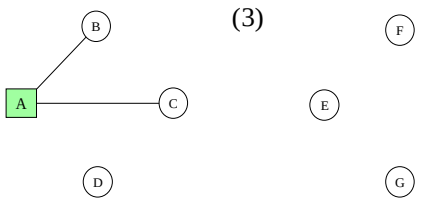
1. (v,w) è un arco dell'albero DFS;
2. v e w sono antenato e discendente nell'albero DFS;

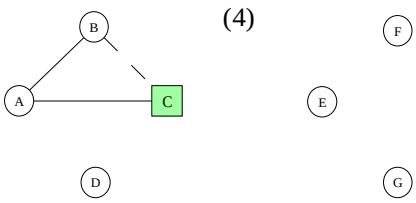
Nella figura che segue sono mostrati i passi che portano alla costruzione dell'albero T secondo l'algoritmo di visita DFS.

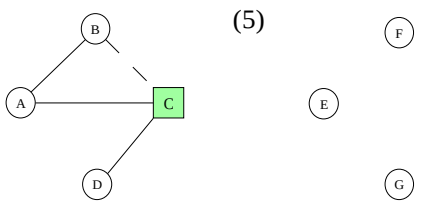

vertice s
B
F
A
C
E
D
G
Grafo iniziale

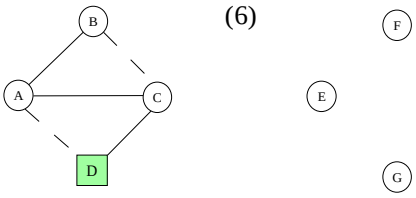

(1)
B
F
A
C
E
D
G

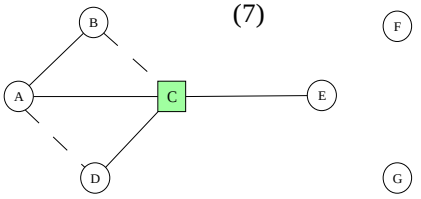

(2)
B
F
A
C
E
D
G

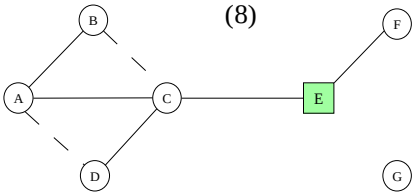

(3)
B
F
A
C
E
D
G

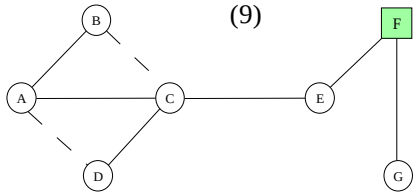

(4)
B
F
A
C
E
D
G

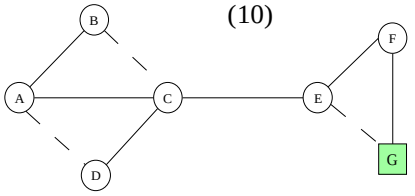

(5)
B
F
A
C
E
D
G

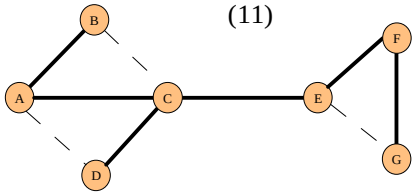

(6)
B
F
A
C
E
D
G


(7)
B
F
A
C
E
D
G


(8)
B
F
A
C
E
D
G


(9)
B
F
A
C
E
D
G


(10)
B
F
A
C
E
D
G


(11)
B
F
A
C
E
D
G

12.1 Minimo albero ricoprente

Si consideri un grafo G=(V,E), con V insieme dei vertici (vertex) ed E insieme di archi (edges). Si definisce albero ricoprente di G quel sottografo T incluso in G e tale che T sia un albero che contenga tutti i vertici di G. Ovviamente un grafo può avere più alberi ricoprenti, uno particolare è però l'albero minimo ricoprente.
Infatti, se associamo un peso o costo ad ogni arco del grafo si può stabilire il costo di ciascun albero ricoprente T che si può individuare nel grafo G. Definiamo allora il costo di un albero ricoprente come la somma dei costi di tutti i suoi archi:

$$w(t) = \sum_{e \in T} w(e)$$

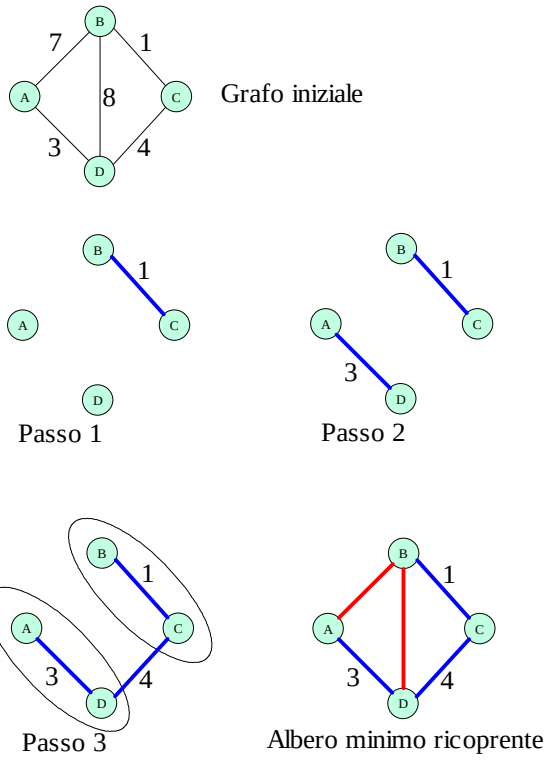
Pertanto, un albero minimo ricoprente è un albero di costo minimo. Non esiste un'unica soluzione al problema appena introdotto tuttavia è bene far osservare che se gli archi del grafo sono tutti distinti allora l'albero ricoprente minimo è unico (lo si può provare con qualche esempio semplice su carta). Se il grafo G considerato all'inizio è un grafo orientato si parlerà allora anziché di albero minimo ricoprente di foresta minima ricoprente.

Una tecnica molto intuitiva e semplice per risolvere il problema del minimo albero ricoprente è la "tecnica golosa". Questa tecnica costruisce l'albero minimo ricoprente un arco per volta effettuando delle scelte "golose" (si scelgono cioè archi di costo piccolo) che escludono gli archi di costo elevato (e quindi pesanti da digerire) dalla soluzione.
Inoltre, per meglio comprendere l'algoritmo si usa colorare solitamente gli archi appartenenti alla soluzione con il colore blu, gli archi invece esclusi dalla soluzione sono colorati rossi. La tecnica "golosa" fa uso essenzialmente di due semplici regole, la regola del taglio e la regola del ciclo che di seguito enunciamo.
La regola del taglio dice: scegli un taglio G che non contiene archi blu. Tra tutti gli archi non colorati del taglio seleziona quindi un arco di costo minimo e coloralo di blu. La regola del ciclo dice: scegli un ciclo semplice in G che non contiene archi rossi. Tra tutti gli archi non colorati del ciclo, seleziona un arco di costo minimo e coloralo di rosso. Il metodo "goloso" applica una delle due regole ad ogni passo e termina quando tutti gli archi sono colorati di un colore.

12.2 Algoritmo di Kruskal

L'algoritmo di Kruskal calcola il minimo albero ricoprente, esso mantiene ad ogni passo la foresta di alberi blu (in altre parola la soluzione provvisoria), quindi per ogni arco applica la seguente regola: se l'arco ha entrambi gli estremi nello stesso albero blu, coloralo di rosso. Altrimenti coloralo di blu.

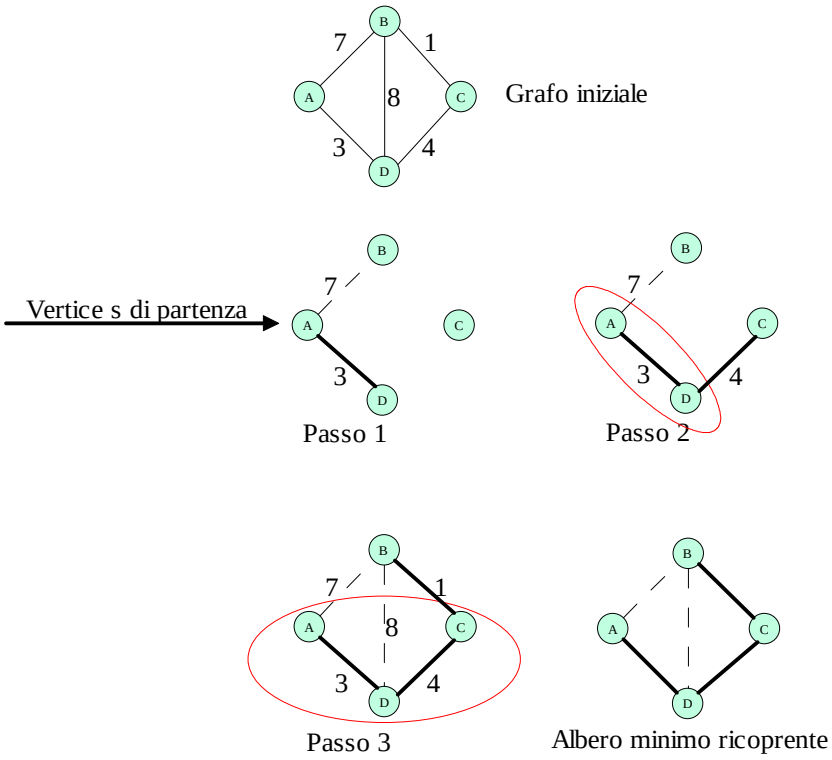
```
algoritmo kruskal(grafo G) → albero
      ordina gli archi di G=(V,E) secondo costi non decrescenti
      T ← albero vuoto
      for each( arco(x,y) di G in ordine non decrescente di costo ) do
            if( x ed y sono connessi in T) then aggiungi l'arco (x,y) a T
      return T
```



12.3 Algoritmo di Prim

Anche l'algoritmo di Prim calcola il minimo albero ricoprente, esso scegli l'arco di costo minimo incidente su T e lo colora di blu. Dopo n-1 passi l'albero blu contiene tutti i vertici (con n numero di nodi o vertici). All'inizio l'albero consiste di un unico vertice che può essere scelto arbitrariamente, tanto alla fine dei passi necessari la soluzione lo includerà nell'albero blu dovendo questo toccare tutti i vertici del grafo G=(V,E).

```
algoritmo Prim(grafo G) → albero
      T ← albero formato da un solo nodo s
      while (T ha meno di n nodi) do
            trova l'arco e di costo minimo incidente su T
            aggiungi l'arco e all'albero T
      return T
```



13.1 Cammini minimi

Il problema della ricerca dei cammini minimi si manifesta in applicazioni telematiche su reti di comunicazioni ma anche nel settore dei trasporti quando si vuole determinare il cammino da intraprendere su strada percorrendo la minima distanza, ciò va a tutto vantaggio dei consumi e del guidatore. In Internet ad esempio l'instradamento dei pacchetti verso una destinazione nota deve possibilmente impiegare il minor numero possibile di passaggi su server (qualora sia possibile).
Consideriamo ancora una volta il grafo G=(V,E) ed associamo ad ogni arco e(A,B) dell'insieme E un costo di percorrenza. Ad esempio, se il nostro grafo G modella in parte l'attuale Internet moderna il costo dell'arco potrebbe essere rappresentativo del livello di congestione di un link verso un server ed indicare quindi il tempo necessario all'attraversamento in base al livello di congestione o saturazione del collegamento. Oppure, se il grafo G modella una rete stradale il costo dell'arco potrebbe allora rappresentare la distanza in chilometri che intercorre fra le due città A e B. In base a quanto detto diciamo che due vertici X ed Y di G sono connessi se esiste un cammino,fatto da archi e(A,B), che collega X ad Y. Si definisce poi il costo di un cammino come la somma dei costi dei suoi archi:

$$w(\pi)=\sum_{i=1}^{k} e(A_{i-1},B_i)$$

Banalmente, quando il grafo è formato da un solo vertice il cammino ha costo zero. Definito il peso o costo di un cammino possiamo adesso introdurre la definizione di cammino minimo. Sia G=(V,E) un grafo orientato con archi e(A,B) dotati di costo. Un cammino minimo fra la coppia di vertici X ed Y del grafo G è quel particolare cammino che individua un percorso di archi da X ad Y il cui costo sia il minimo fra tutti i cammini esistenti:

$$w(\overline{XY})=\min w(\pi)$$

Dato un grafo G, in generale, potrebbe esistere anche più di un cammino fra una coppia di vertici ed Y. Qualora gli archi possano avere anche pesi o costi negativi è necessario fare molta attenzione. Se

infatti nel grafo G sono presenti dei cicli si può allora ottenere un cammino con costo -∞. La ricerca di cammini minimi in presenza di archi negativi e di cicli assume un complessità elevata, pertanto negli algoritmi di ricerca di cammini minimi si assumerà che gli archi siano non negativi e che i cicli (se presenti) non abbiano un costo negativo. Sotto queste opportune condizioni è possibile mettere in atto delle soluzioni efficienti come ad esempio l'algoritmo di Djkstra che in seguito osserveremo, prima però ci occorrono due importanti ma semplici proprietà.
Effettuiamo quindi un ulteriore passo in avanti e definiamo la distanza fra vertici di un grafo. La definizione è semplice ed intuitiva, infatti, dato il grafo orientato G=(V,E) ed una funzione di costo w(e) per gli archi e(A,B) dell'insieme E di G, si definisce distanza tra due vertici X ed Y in G il costo di un cammino minimo che connette X ad Y se presente, oppure +∞ se tale cammino non esiste:

$$d_{xy} = \begin{cases} w(\overline{XY}) \ \textit{se esiste un percorso da } X \textit{ ad } Y \\ +\infty \ \textit{altrimenti} \end{cases}$$

Quindi, l'impossibilità di collegare X ad Y si denota con una distanza di +∞. Banalmente, la distanza di un vertice X da se stesso è zero. Una importante proprietà è la cosiddetta disuguaglianza triangolare. Se G=(V,E) è un grafo orientato con funzione di costo w(e) ed e(A,B) archi dell'insieme E di G, allora dati tre vertici X,Y e Z dell'insieme V:

$$d_{xz} \leq d_{xy} + d_{yz}$$

In altre parole la distanza che intercorre da X a Z è minore oppure uguale alla concatenazione delle singole distanze che vanno da X ad Y e da Y a Z. Da ciò scaturisce la seconda proprietà a noi utile che va sotto il nome di condizione di Bellman. Sia G=(V,E) un grafo orientato con funzione di costo w(e) ed e(A,B) archi dell'insieme E di G. Per ogni arco e(A,B) di E e per ogni vertice s di V, le distanze fra i vertici soddisfano la seguente disuguaglianza:

$$d_{sA} + w(A,B) \geq d_{SB}$$

Ricordiamo che abbiamo precedentemente escluso la possibilità di avere archi di costo o peso negativo, infatti, senza questa ipotesi la proprietà non è più vera.
Se avessi già le distanze oppure i pesi che intercorrono tra ogni coppia di vertice potrei a questo punto calcolare un albero minimo dei cammini. Tuttavia a partire da un vertice v del grafo G=(V,E) non si conoscono a priori le distanze oppure i pesi degli archi che connettono il vertice v agli altri vertici del grafo G. Gli archi allora sono solitamente interrogati sul loro peso, ciò favorisce il calcolo delle distanze fra i vertici. Tutti gli algoritmi di ricerca di cammini minimi calcolano le distanze fra i vertici per poi identificare i cammini minimi. Solitamente si parte da una stima di tale distanza che viene successivamente aggiornata quando si scoprono man mano gli altri vertici del grafo. L'aggiornamento della stima prima citata consiste nel considerare un vertice v del grafo G=(V,E) ed un cammino π_{vy} si procede quindi applicando la tecnica del rilassamento.
Molto brevemente, la tecnica del rilassamento inizia a determinare una stima delle distanze e se trova un cammino che non è minimo lo esclude e calcola nuovamente le distanze. Tutto ciò viene ripetuto iterando il processo descritto più volte:

$$if\left(Dxv + w(\overline{vy}) < D_{xy}\right) then D_{xy} \leftarrow D_{xv} + w(\overline{vy})$$

Molti algoritmi di ricerca dei cammini minimi differiscono essenzialmente per il modo in cui scelgono il vertice v ed il cammino π_{vy}.

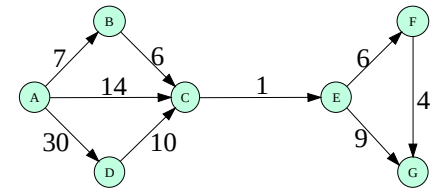
13.2 Algoritmo di Djkstra

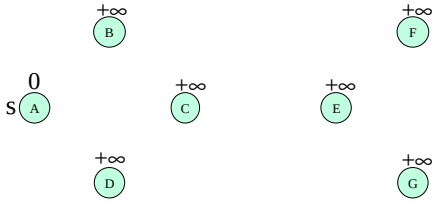
Abbiamo adesso tutti gli strumenti necessari alla formulazione del noto algoritmo di ricerca dei cammini minimi, l'algoritmo di Djkstra implementato alla fine degli anni 50! L'algoritmo fa uso della condizione di rilassamento prima enunciata e prevede l'assenza di archi di peso negativo e cicli di costo negativi. L'algoritmo di Djkstra costruisce l'albero T dei cammini minimi applicando (n-1) volte il seguente passo:

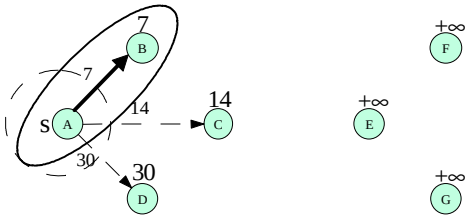
Scegli un arco(u,v) con u appartenente all'insieme dei vertici V e v non appartenente all'insieme dei vertici V che minimizza $D_{su}+w(u,v)$, effettua il passo di rilassamento $D_{sv} \leftarrow D_{su}+w(u,v)$ e aggiungilo a T.

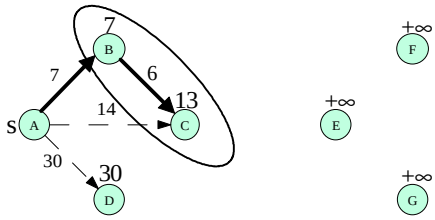
Gli archi incidenti vengono tenuti in una coda con priorità, infatti, la scelta dell'arco incidente è di importanza cruciale per l'efficienza dell'algoritmo di Djkstra. L'algoritmo di Djkstra è molto simile all'algoritmo di Prim per il calcolo di un minimo albero ricoprente. L'algoritmo sfrutta una interessante proprietà delle distanze in un grafo che permette di estendere in modo goloso un albero di cammini minimi aggiungendo progressivamente archi (u,v) estratti da una coda prioritaria in ordine di $D_{su}+w(u,v)$ crescente.

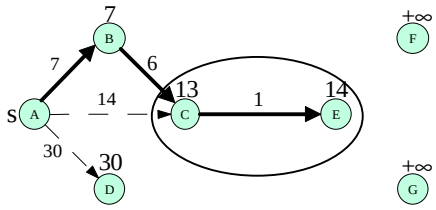
```
algoritmo Dijkstra(grafo G,vertice s) → albero
      inizializza D tale che Dsv=+∞ per v≠s e D(s)=0
      T ← albero formato dal solo nodo s
      while (T ha meno di n nodi) do
            trova l'arco (u,v) incidente su T con Dsu+w(u,v) minimo
            Dsv←Dsu+w(u,v) (Tecnica del rilassamento)
            Rendi u padre di v in T
      return T
```

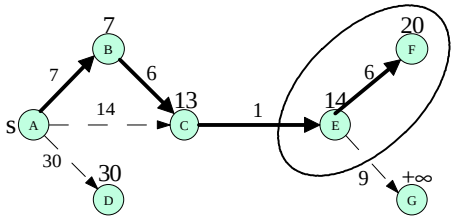

B
7
6
A
14
C
1
E
6
F
4
30
10
9
G
D

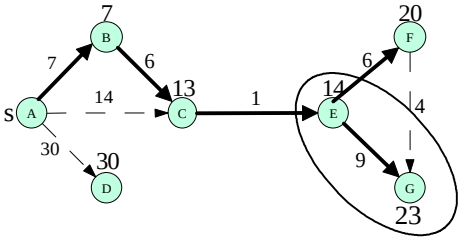

+∞
B
+∞
F
0
s
A
+∞
C
+∞
E
+∞
D
+∞
G

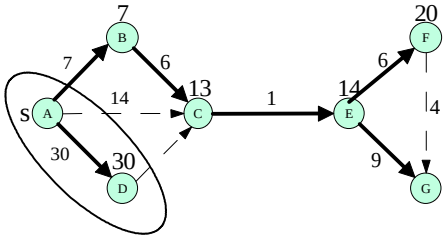

7
B
+∞
F
7
s
A
14
C
+∞
E
30
30
D
+∞
G


7
B
+∞
F
7
6
s
A
14
13
C
+∞
E
30
30
D
+∞
G


7
B
+∞
F
7
6
s
A
14
13
C
1
14
E
30
30
D
+∞
G


7
B
20
F
7
6
s
A
14
13
C
1
14
E
6
30
30
D
9
+∞
G


7
B
20
F
7
6
s
A
14
13
C
1
14
E
6
4
30
30
D
9
G
23


7
B
20
F
7
6
s
A
14
13
C
1
14
E
6
4
30
30
D
9
G