INTRO

La parte algoritmica è l'ex "Strutture dati". Si occupa della definizione di un modello per l'analisi degli algoritmi – ossia la determinazione della velocità di un algoritmo, che sia indipendente dalle specifiche del calcolatore. È uno dei criteri determinanti per la scelta della tipologia di risoluzione di un problema. Si riflette nella domanda: *"Analizza questo algoritmo".* Per la valutazione vengono effettuate analisi asintotiche.

Il primo strumento di progettazione è la ricorsione; gli algoritmi vengono progettati ricorsivi. Esistono diversi schemi ricorsivi, alcuni elementari, altri sofisticati. Implementare un algoritmo ricorsivo in un programma può non essere banale ed è importante saper fare debugging. Alcuni problemi hanno una formulazione intrinsecamente ricorsiva e non possono essere risolti diversamente.

Si useranno **strutture di dati non lineari**. Una lista nodi è una struttura collegata lineare; una lista multiplamente collegata (es. albero) è una struttura collegata non lineare. Fanno parte di questa categoria anche le heap (code con priorità), le tabelle hash, i grafi (generalizzazione degli alberi). Gli algoritmi di base sono invece la ricerca binaria e gli algoritmi di ordinamento.
Uno strumento teorico permette di provare che non esiste algoritmo di ricerca migliore della ricerca binaria, il cui costo è log(n).

I problemi reali dell'informatica sono al più relativi ai modelli. La base della parte di modelli risiede nella logica matematica e in alcuni formalismi quali la prova di un teorema. Si procederà poi con la teoria dei linguaggi e degli automi di Chomsky. La sua sistematizzazione è valida per i linguaggi informatici. Gli automi sono macchine astratte che simulano il ragionamento, e si prestano alla traduzione in algoritmi. I linguaggi possono essere divisi in livelli da 0 a 3, di cui si approfondirà il 3 e si tratterà parte del 2.

La **teoria della calcolabilità** determina quanti problemi possono essere risolti da un computer e quanti non possono esserlo. Esistono problemi la cui inesistenza della soluzione è stata anche dimostrata formalmente. I risolvibili sono inferiori in numero.
La **teoria della complessità** determina quanto tempo di calcolo impiega la risoluzione di un algoritmo. Alcuni algoritmi risolvibili richiedono talmente tanto tempo da non essere molto lontani dagli irrisolvibili.

Algoritmi

RICORSIONE

Costi di un algoritmo: *tempo* (d'esecuzione) e *spazio* (in memoria)
Gli algoritmi ricorsivi non esplicitano lo spazio occupato in memoria; nella valutazione del costo spaziale della ricorsione occorre tenerne presenti i meccanismi. Il principio alla base del funzionamento della ricorsione è il principio d'induzione. Nel calcolo di tutti i record di attivazione generalmente si considera il passo precedente, senza risalire a monte.

La versione iterativa del calcolo del fattoriale (cfr. file .java, primo metodo) permette un impiego migliore della memoria, eliminando numerose chiamate ricorsive (quindi meno costi). La ricorsione ha una formulazione più compatta e lo spazio occupato in memoria è proporzionalmente calato in costo nel corso degli anni, ma la **risorsa tempo** ha ancora il valore originariamente attribuitole, pertanto maggiore di quello associato al tempo.

Le funzioni di costo vengono calcolate in maniera asintotica.
Lo schema della ricorsione è *lineare*. Vengono innanzitutto verificati i casi base, che non utilizzano la ricorsione; la chiamata ricorsiva è unica: possono esistere più tipi di chiamate ricorsive, ma ogni esecuzione del metodo ne dovrà eseguire una sola.

L'errore più comune nella definizione di algoritmi ricorsivi è **non usare il risultato della precedente chiamata** (cfr. file .java, secondo e terzo metodo).
La ricorsione può essere pura o fare side effect. Le specifiche del problema possono richiedere o non richiedere il side effect. Salvo il caso in cui venga richiesto diversamente, è consigliabile ricorrere sempre alla ricorsione pura (in sede d'esame il side effect non richiesto viene penalizzato).
In linea generale: il side effect è IL DIAULO! *{Ogni riferimento a Matteo Montesi è puramente casuale}*

Il calcolo ricorsivo di una potenza (cfr. file .java, quarto metodo) ha costo O(n). Asintoticamente, sono n chiamate dello stesso algoritmo. È tuttavia possibile migliorarne il costo (cfr. file .java, quinto metodo). La seconda formulazione ha costo pari a **O(log(n))**: i valori di n si dimezzano ad ogni chiamata, quindi alla chiamata i corrisponde un valore pari a n/2^(i-1). Tale valore sarà 1 per n = 2^(i-1), quindi log(n) = log(2^(i-1)) = i-1. Quindi **i = log(n)+1**.

*Esercizio: valutare il numero di chiamate ricorsive se, nel quinto metodo, si inserisse pow2*pow2 anziché memorizzare in una variabile d'appoggio il risultato.*

Esercizio: scrivere un metodo iterativo che calcoli la potenza.

Un caso particolare è la **ricorsione di coda**, in cui la chiamata ricorsiva si verifica come ultimo passo del metodo. Ne sono esempi i primi tre metodi del file .java. La ricorsione di coda è particolarmente facile da tradurre in una formulazione iterativa, senza ricorrere ad alcun extra nella pila di attivazione.

Altro caso particolare è la **ricorsione binaria**, che prevede due chiamate ricorsive per caso non-base. Un esempio è il drawTicks, che disegna le tacche di un righello inglese su schermo (cfr. file .java, metodi dal settimo al decimo). Ad ogni chiamata ricorsiva la massima dimensione del record di attivazione è pari al numero di "rami" della ricorsione, ma lo spazio occupato tiene conto anche delle ramificazioni adiacenti. L'altezza dell'albero è la pila dei recordi di attivazione.
Il primo livello dell'albero è il livello zero, dove ci sono 2^0 = 1 nodi. Ad ogni livello ci sono 2^n nodi. La somma di nodi del'albero è Sigma(2^i), ovvero (2^(k+1)-1)/(2-1) quindi 2^(k+1)-1. In tal caso k = log(n), quindi 2^(log(n)+1)-1 = 2*2^(log(n))-1 = **2n – 1**.

Factorial

```
public static int fact(int n)
        {
        if(n == 0 ll n == 1) return 1;
        else return fact(n-1);
        }
```

Input: array of n integers and an integer b >= 1, such that n >= b
Output: sum of the first b integers in the array

```
public static int linearSum(int[] a, int n)
        {
        if(n == 1) return a[0];
        else return linearSum(a, n-1) + a[n-1];
        }
```

Input: array a and two positive integer index numbers, i and j
Output: a reversed array from i to j

```
public static int[] reverse(int[] a, int i, int j)
        {
        if(i >= j) return a;
        else
                {
                int x = a[i];
                a[j] = a[i];
                a[i] = x;
                return reverse(a, i+1, j-1);
                }
        }
```

Integer power

```
public static int pow(int n, int exp)
        {
        if(exp == 0) return 1;
        else return n * pow(n, exp-1);
        }
```

Integer power (optimised)

```
public static int pow2(int n, int exp)
        {
        if(exp == 0) return 1;
        if(exp > 0 && n % 2 != 0)
                {
                int c = pow2(n, (exp-1) / 2);
                return n * c;
                }
        else
                {
                int c = pow2(n, exp / 2);
                return n * c;
                }
        }
```

Array inversion (iterative formulation)

```
public static int[] reverseCyclic(int[] a, int i, int j)
	{
	if(i == j) return a;
	else
		{
		while(i < j)
			{
			int x = a[i];
			a[j] = a[i];
			a[i] = x;
			i++;
			j--;
			}
		}
	return a;
	}
```

Ruler drawer

```
public static void drawRuler(int inches, int majorLength)
	{
	drawOneTick(majorLength, 0);
	for(int i = 1; i < inches; i++)
		{
		drawTicks(majorLength-1);
		drawOneTick(majorLength, i);
		}
	}

public static void drawTicks(int tickLength)
	{
	if(tickLength > 0)
		{
		drawTicks(tickLength-1);
		drawOneTick(tickLength);
		drawTicks(tickLength-1);
		}
	}

public static void drawOneTick(int tickLength)
	{
	return drawOneTick(tickLength, -1);
	}

public static void drawOneTick(int tickLength, int tickLabel)
	{
	for(int i = 0; i < tickLength; i++)
		System.out.print("-");
	if(tickLabel >= 0) System.out.println(" " + tickLabel);
	}
```

La progettazione ricorsiva si basa sull'assunzione che per il caso n sia già pronta la soluzione, che si possa riutilizzare nel caso n+k per qualsiasi k.

Uno stesso metodo ricorsivo può essere utilizzato in diversi modi. Una formulazione ricorsiva del calcolo della serie di Fibonacci (cfr. file .java, primo metodo) può avere un costo esponenziale, richiedendo 2^(k/2) passaggi. Tale formulazione dell'algoritmo ripete molte volte gli stessi calcoli.

Esercizio: formulare Fibonacci ricorsivamente lasciando l'output a int e aggiungendo parametri in input.

La **ricorsione multipla** è, fra le altre cose, alla base del bruteforcing: funziona, ma è molto pesante in memoria, soprattutto per grandi quantità di dati da elaborare.
L'algoritmo *puzzleSolve(int k, sequence S, domain U)* enumera ogni stringa di lunghezza k aggiunte ad S usando senza ripetizioni elementi di U. Tali stringhe vengono poi rimosse da U.
Quindi: se la configurazione risolve il rompicapo, si ritorna una stringa; altrimenti, lo si reinserisce in U e lo si rimuove di nuovo dalla coda di S.

Esercizio: progettare un metodo che calcoli tutte le possibili permutazioni di una stringa. Provare ad assegnare numeri ad ogni nodo.

Data una griglia che è una matrice n*n, scegliere una casella iniziale a proprio piacimento dove inserire il numero 1. Inserire poi tutti i numeri seguenti secondo il criterio: se i è nella casella x, il numero successivo può essere solo in una casella distante 2 caselle da essa (in orizzontale, verticale o diagonale). Non si possono sovrascrivere le caselle piene. Trovare tutte le disposizioni possibili per questa formulazione.

In un albero da n+1 livelli ci sono n! nodi per livello. Nelle chiamate ricorsive ciò equivale a n! + (n-1)! + (n-2)! + [...] chiamate ricorsive.

Il **tempo di output** mediamente cresce proporzionalmente alla dimensione dell'input. Un array, per esempio, ha x*n dati, e dell'x più grande prende il numero di bit e codifica la dimensione massima dell'array sulla base di quel risultato. Valutando il tempo di esecuzione si prende per questo sempre il caso peggiore.

// Fibonacci funziona sia in esponenziale che in lineare (il secondo fa un po' schifo ma vabbè :P), sul puzzleSolver devo ancora lavorarci, portate pazienza e verrà uppato!

Fibonacci numbers, recursive calculation (base formula)

```
public static int fib(int k)
	{
	if(k <= 0)
		return 0;
	if(k <= 2)
		return 1;
	return fib(k-1) + fib(k-2);
	}
```

Fibonacci numbers (linear formula)

```
public static int fibLinear(int k, int cap, int temp1, int temp2)
	{
	if(k <= 0)
		{
		if(k != cap) return fibLinear(k+1, cap, 0, 0);
		return 0;
		}
```

```
	else if(k == 1)
		{
		if(k != cap) return fibLinear(k+1, cap, 0, 1);
		return 1;
		}
	else if(k == 2)
		{
		if(k != cap) return fibLinear(k+1, cap, 1, 1);
		return 1;
		}
	else
		{
		if(k >= cap) return temp1 + temp2;
		return fibLinear(k+1, cap, temp2, temp1 + temp2);
		}
	}
```

PuzzleSolver

```
public static Set<String> puzzleSolver(int c, Set<String> out, Set<String> in, Set<String>
basic)
	{
	if(out.size() >= fact(in.length())) return out;
	else if(!in.equals("") && in != null)
		{

		}
	}

private int fact(int n)
	{
	for(int i = 1; i < n; i++)
		n = n * i;
	return n;
	}

public static void printSet(Set<String> s)
	{
	String[] x = s.toArray();
	for(int i = 0; i < x.length; i++)
		System.out.println(x[i]);
	}
```

COSTO DI UN ALGORITMO

Il tempo di esecuzione, o costo computazionale, è espresso in funzione della dimensione dell'input. A parità di dimensione ci possono essere tempi di esecuzione diversi, distinti in casi medi, casi migliori e casi peggiori. Il caso medio è il valore atteso del costo, ma per quantificarlo occorrerebbe conoscere la distribuzione probabilistica di ciascun caso; per questo, nella valutazione del costo, si tiene sempre presente il **caso peggiore** *(worst case scenario)*.

Alcuni metodi, anche in Java, permettono di calcolare il **tempo di esecuzione** di un dato algoritmo tramite la differenza tra il tempo attuale a inizio metodo e il tempo attuale a fine metodo. Tuttavia, per tempi di esecuzione molto brevi, la differenza si approssima a 0.
Lo svantaggio di questo metodo è il dover implementare ogni algoritmo che si vuole valutare. Inoltre, non si è mai sicuri di avere effettivamente davanti il caso peggiore, né si può provare ogni configurazione, e la misurazione è relativa al tipo di hardware della macchina di testing.

Quindi è chiaro che l'approccio sperimentale non dà garanzie: per questo si ricorre solitamente all'**analisi teorica** di ciascun algoritmo. La fase di testing risulta utile dopo l'analisi teorica, per provare sperimentalmente i risultati ottenuti per via teorica. Le stime sono valide asintoticamente, quindi sarà impossibile scegliere tra due algoritmi asintoticamente equivalenti.
L'analisi teorica viene condotta solitamente sullo pseudocodice.

for(int i = 0; i < a; i++) (c O(a) = O(2^z), come vedremo) è un ciclo che fa *a* iterazioni. Quindi il suo costo è a, ovvero il **numero di bit** usati per rappresentare a: a = 2^z; z = log(a). Distinguere tra caso peggiore e caso migliore non dipende dalla dimensione del seconodo dato, idem per l'input.

Lo pseudocodice è "universale" e può essere tradotto in qualsiasi linguaggio. È fondamentale specificare se l'algoritmo fa o non fa side effect. La struttura dell'algoritmo deve essere evidente, ma non si richiede stretta osservanza della sintassi.
Il modello di computer associato allo pseudocodice è il **modello astratto RAM**, Random Access Machine, in grado di accedere direttamente a ciascuna casella di memoria per leggerla e modificarla. Ha una CPU, infinite caselle di memoria con indice.

L'algoritmo in pseudocodice illustra una serie di azoni, ossia computazioni fondamentali. Qualunque linguaggio formalizzi l'algoritmo, esso sarà descrivibile.
Diverse operazioni hanno costo unitario, ad esempio le chiamate a metodo e i ritorni.
Nello pseudocodice per **calcolare il massimo valore delle n celle di un array**, il costo dell'assegnazione nel ciclo for è 2n − 2, poiché si ripete n volte e ogni volta fa n assegnazioni.
Chiamando a il caso migliore e b il caso peggiore, **a <= T(n) <= b** che sono entrambe 12x − 4. Quindi è limitato da entrambe le parti da due funzioni lineari! In tal caso possiamo affermare che il costo sia lineare. Il discorso è valido anche per le macchine reali, nonostante non sempre si conosca la pendenza (che comunque non è rilevante, ovvero a livello asintotico x e 10x sono equivalenti).

Date f(n) e g(n), si dice che f(n) è **O(g(n))** se esistono c ed n0 tali che: f(n)<= cg(n) per ogni n >= n0. In termini semplici, O(f(n)) è definito dal grado del monomio di grado più alto presente nella funzione. Per esempio, se f(x) = x^2 + 1, O(f(x)) = O(x^2), ma anche O(x^3) e le potenze successive, ma il grado rilevante è il più alto con coefficiente non nullo. Ad esempio per f(x) = x^2, O(f(x)) non è O(x) ma O(x^2)! Il "contenuto" della O è detto *upper bound* (limite superiore) di f(x). In tal caso f(n) cresce *al più* come g(n), mentre per i lower bound f(n) cresce *almeno* come g(n). Il lower bound si indica con omega; la crescita equivalente si indica con theta, e indica l'equivalenza asintotica di due funzioni.
Le costanti additive e moltiplicative, anche qui, si perdono. Nella scelta delle classi di funzioni per le O(f(n)) si sceglie sempre la più piccola.

L'obiettivo dell'analisi asintotica è la stima della funzione di costo. Lo scopo è di rilevare sia la O(g(n)) che la Omega(g(n)). Non sempre il caso migliore e il caso peggiore hanno lo **stesso andamento asintotico**.
In presenza di una funzione dalla complicata stima, si procede per approssimazioni successive; per esempio, se si sa che una funzione non avrà mai un costo superiore a n^3, la si può porre come O(n^3). Similmente, se si sa che non impiegherà mai meno di n operazioni, si può porre Omega(n). Il caso peggiore è sempre quello da tenere in considerazione.

Nell'esempio dell'arrayMax, O(n) = Omega(n) e quindi il costo asintotico dell'algoritmo è pari a n. L'ottimizzazione consente nel far sì che **O e Omega** corrispondano, o quantomeno fornire una stima quanto più possibile precisa di entrambe. Le costanti vengono ignorate, quindi nella valutazione del costo temporale di un algoritmo si considera di solito il numero di esecuzioni di ogni azione ripetuta frequentemente. All'interno della valutazione bisogna considerare anche le chiamate ad altre funzioni o metodi.

Si definisce ***upper bound*** di un algoritmo la funzione f(n) di un algoritmo il cui costo di esecuzione è O(f(n)). Ne esiste uno sul tempo e uno sullo spazio.
Si definisce ***lower bound*** di un algoritmo la funzione f(n) di un algoritmo il cui costo di esecuzione è Omega(f(n)). Ne esiste uno sul tempo e uno sullo spazio.
Si definisce ***upper bound*** di un problema la funzione f(n) di un algoritmo il cui upper bound è O(f(n)) nel caso peggiore dell'implementazione migliore.
Si definisce ***lower bound*** di un problema la funzione f(n) di un algoritmo che consiste nello stimare il lower bound del caso peggiore dell'implementazione migliore, considerando *tutte le varianti implementative dell'algoritmo.*

Supponiamo di avere tre algoritmi che risolvono lo stesso problema: A1, A2 e A3. Di A2 abbiamo una stima tight (in cui O(g(n)) = Omega(g(n))), di A1 e A3 abbiamo una stima con notevoli differenze tra le due funzioni. Di esse **non possiamo delimitare** il lower bound, né l'upper bound: non è possibile testare tutte le funzioni.

Ad esempio, consideriamo un algoritmo che calcola le **medie dei prefissi**, ovvero la media dall'elemento 0 all'elemento i di un array di interi o double (es. voti universitari). La prima soluzione (cfr. slides) ha O(n^2), ma è possibile anche una formulazione in O(n) (cfr. file .java della lezione corrispondente). Verificato che O(n) è anche il lower bound, si conclude che l'algoritmo è **ottimale**.

O(1) indica un algoritmo a costo costante.

Prefix averages (recursive formulation; cost: O(n))

```
public static double prefixAveragesRec(int i, int ct, double[] a, double tot)
        {
        if(ct >= i)
                return tot / i;
        tot += a[ct];
        return prefixAveragesRec(i, ct + 1, a, tot);
        }
```

ALBERI

Gli **alberi** sono la generalizzazione del concetto di lista collegata. Hanno una struttura multilineare con diversi nodi successivi.

I **tipi di dato astratto** (ADT) sono definizioni del tipo di un oggetto o entità e delle operazioni che occorre implementarvi. Per esempio:
ADT Classe; ins_matr; del_matr; ins_subj; del_subj.
L'astrazione è valida in tutti i linguaggi di programmazione a oggetti. Nei linguaggi procedurali, come il C, al posto di definire classi con metodi si definiscono soltanto funzioni.
L'individuazione parte dal dominio dell'entità, sul quale si baseranno le variabili d'istanza, nel caso di Java.

Caratteristica fondamentale degli alberi è che ogni nodo ha **un solo arco entrante** (o zero, nel caso della radice), e un qualsiasi numero di archi uscenti. L'albero è un caso particolare del grafo, struttura che non pone vincoli sul numero di archi entranti. È bene esplicitare graficamente la struttura di un albero.

Un albero è un modello astratto di struttura gerarchica, la cui **definizione ricorsiva** è: ogni nodo è un albero; ogni albero ha un numero qualsiasi di sottoalberi. La progettazione degli alberi è, infatti, ricorsiva. Un **file system** si rappresenta con un albero, i cui sottoalberi sono le sottocartelle. Alcuni nodi possono rappresentare anche i file contenuti nelle sottocartelle, e i file sono nodi senza figli. Vengono chiamati anche foglie o *nodi esterni*, mentre le sottocartelle sono *nodi interni*.

La **profondità** di un nodo è il numero di antenati che ha. L'**altezza** è la profondità massima di un albero, corrispondente alla profondità della foglia più lontana dalla radice.
Riassumendo:

ADT Tree :

- Le *position* estraggono i nodi: Object element();
- *Metodi generici:* int size(); boolean isEmpty(); Iterator elements(); Iterator positions();
- *Metodi di accesso:* position root(); position parent(p); positionIterator children(p);
- *Metodi di query:* boolean isInternal(p); boolean isExternal(p); boolean isRoot(p);
- *Metodi di aggiornamento:* Object replace(p, o);
- Altri metodi si possono definire in strutture che implementano gli ADT Tree.

* La definizione si basa sul linguaggio Java. Una normale definizione di tipi astratti dovrebbe essere indipendente dal linguaggio di programmazione.

Position è un concetto astratto, un contenitore. Gli Object dovrebbero essere sostituiti dai tipi generici. Non c'è metodo di **aggiunta di nodi**; è necessario specificare quale cartella deve contenere il nodo o la foglia da immettere. Se l'ordine è rilevante, potrebbe essere necessario specificare la posizione anche rispetto ai fratelli (nodi dello stesso livello). Quindi non viene incluso nella progettazione "standard" dell'albero. È uno dei tipi astratti la cui implementazione è largamente arbitraria.

È importante anche distinguere tra **metodi** della **classe Nodo** e metodo della **classe Albero**. Per esempio isLeaf() può essere associato al nodo, ma è più corretto definirlo in albero, poiché è l'albero stesso a conferire al nodo la proprietà di essere foglia o meno.

La **visita** (o attraversamento) **in preordine** di un albero è la visita sistematica di tutti i nodi dell'albero, ciascuno visitato una volta sola. Si parte dalla cima e poi si visitano i discendenti. Si può distinguere tra la **visita in ampiezza**, che si focalizza prima sui fratelli e poi sui discendenti, e **visita in profondità**, che si focalizza prima sui discendenti e poi sui fratelli. Si chiamano anche "visita a ventaglio" e "visita a scandaglio".
Visita in profondità: preOrder(v) visit(v); for each child w of v preOrder(w);
La visita in profondità, concettualmente, termina alla fine del ramo. Il ciclo termina una volta finiti i figli di ciascun sottoramo.

La **visita in postordine** ha uno schema opposto: postOrder(v) for each child w of v postOrder(w); visit(v);
Quindi visita l'albero a partire dal fondo. L'algoritmo visit(x) può essere una funzione di stampa o di altro tipo, a seconda delle esigenze.

Supponiamo per semplicità che ogni cartella nell'albero abbia in sé l'informazione di quanti byte occupa in memoria. Conosciamo anche il peso di ogni file. La variante sul problema è assegnare a ogni nodo un numero corrispondente al *numero dei suoi sottonodi*. In entrambi i casi la visita in postordine è la più indicata.

Il **costo** dell'algoritmo postOrder(x): il suo input è O(n), ma le iterazioni del ciclo variano a seconda del numero dei sottonodi, da 0 a n − 1. Ogni nodo viene visitato una volta sola, anche perché visitarlo due volte richiederebbe 2^n visite. Quindi il costo totale, tanto su upper che su lower bound, così pure per preOrder, è **Theta(n)**.

Un caso particolare dell'albero è l'**albero binario**, in cui ogni nodo ha al più due figli. Suo equivalente a più dimensioni è l'albero ennario. Si parla di "figlio destro" e "figlio sinistro". L'albero binario **completo** ha esattamente due figli per nodo, ovvero, al livello n-esimo ha esattamente 2^n nodi (la radice è il nodo 0).
Un albero binario rappresenta bene le operazioni aritmetiche.

ALBERI BINARI

L'**albero binario** ha figli sinistri e figli destri. Gli utilizzi degli alberi binari variano dai processi decisionali agli alberi di ricerca. Anche nelle espressioni aritmetiche si usano spesso gli alberi binari. Gli operatori aritmetici sono tutti binari.
Un **albero "full"** è un albero che ha esattamente n figli per ogni nodo. L'albero completo è sempre full, ma gli alberi full non sono necessariamente completi.

Gli alberi di decisione sono alberi binari full in cui la divisione dei nodi figli avviene secondo la dicotomia Sì/No nelle risposte. Si possono anche ordinare elementi secondo questa struttura:

```
a > b? -> Y -> b > c? -> Y -> a, b, c
                      -> N -> a > c? -> Y -> a, c, b
                                     -> N -> c, a, b
       -> N -> b > c? -> Y -> a > c? -> Y -> b, a, c
                                     -> N -> b, c, a
                      -> N -> c, b, a
```

L'albero binario ha interessanti applicazioni. Ad esempio può indovinare un oggetto al quale sta pensando il giocatore.

Gli alberi binari hanno diverse proprietà associate alle sue caratteristiche: n nodi, e nodi esterni, i nodi interni (n = e + i), h altezza.
Il minimo numero di foglie di un albero binario è 1. Il **massimo numero** si registra in corrispondenza di **alberi completi** (con tutte le foglie sullo stesso livello). Il totale delle foglie è 2^h. Il totale dei nodi è 2^(h + 1) − 1. Quindi h + 1 = (log(n + 1) − 1).

L'addizione di due nodi comporta la perdita di una foglia e l'acquisizione di due foglie nuove, nell'albero full. L'ADT BinaryTree estende Tree e ha i metodi aggiuntivi hasLeft, hasRight, left,

right. Vi si può applicare la **visita inordine**, da sinistra a destra. Le tre modalità di visita vengono dette "anticipata", "simmetrica" e "posticipata".

Il numero nell'ordine di visita è nella posizione alla quale viene visitato l'albero. L'organizzazione inordine è tipica delle espressioni aritmetiche, ove si aggiunge una parentesi per simbolo/operando.
Il costo di inordine è n. Ogni nodo viene controllato una e una sola volta. Non esistono casi migliori, peggiori o medi: il costo è costantemente uguale a n.

Il **cammino euleriano** è una generica visita di un albero binario. Ha le visite prima viste come casi particolari. Ogni nodo interno viene toccato 3 volte, quelli esterni 1. Quindi l'algoritmo è Theta(n) come i precedenti. Giunti al nodo esterno, si visita; in un nodo interno, visita il nodo a sinistra e a destra e chiama ricorsivamente entrambi sul nodo sinistro e destro rispettivamente. Si invoca anche un metodo che visiti uno dei figli.

Nel concreto, un **albero** è rappresentato da **un nodo che memorizza**:
- un *elemento*
- il puntatore al *nodo genitore*
- la lista di *sequenza dei figli*

L'ADT *position* è implementato direttamente nei nodi. Il puntatore al genitore è indispensabile solo se l'accesso NON avviene dalla radice (all'esame occorre motivare la scelta di inserire un puntatore al genitore).
In alternativa, si può implementare attraverso una struttura collegata universale che permette la rappresentazione di un **qualsiasi albero** di una qualsiasi dimensione:
- un *elemento*
- il puntatore al *primo figlio*
- il puntatore al *prossimo fratello*

I nodi si possono **memorizzare in un array**, ipotizzando che parta da 1. I nodi vengono allocati nelle posizioni 2i, 2i + 1. Quindi si può navigare nell'albero usando semplicemente gli indici degli array. Lo svantaggio è che si creeranno dei buchi esponenziali all'interno della struttura, a meno che l'albero non sia completo: in tal caso la rappresentazione è ottimale.

CODA

La **coda** *(queue)* è un tipo astratto che segue la politica FIFO (First In, First Out). Gli inserimenti in coda vengono definiti accodamenti (queuing). La realizzazione più ovvia è attraverso strutture collegate lineari. È d'interesse anche quella basata su un **array circolare**, avente due variabili intere, una per l'inizio e una per la fine della coda.
L'indice viene incrementato con mod(size)|i++|.

L'ADT **coda di priorità** gestisce una collezione di elementi di cui ogni entry è formata da una coppia <key, value>. La chiave indica il "privilegio" d'ingresso in coda; infatti, le code di priorità non sempre seguono la politica FIFO. L'inserimento avviene tramite insert(k, x) e per la rimozione implementiamo removeMin(), che rimuove l'entry col privilegio più alto (cioè con la chiave di valore minimo). Se esistono più elementi di questo tipo, se ne rimuove uno qualsiasi.

Un ovvio esempio di coda di priorità è il **management dei processi attivi** nel sistema. La si può impostare come round robin, ponendo la priorità uguale per tutti i processi in esecuzione, oppure definire una priorità più elevata per alcuni dati processi.

Può essere difficile determinare la priorità degli elementi in coda a partire da chiavi non intere e che non sono stringhe (ordinamento lessicografico); se però due chiavi sono due oggetti, non è altrettanto semplice.
Se U è l'insieme delle chiavi, UxU = U^2 ha in sé l'insieme delle chiavi esistenti. La relazione viene chiamata **relazione d'ordine** se gode delle proprietà riflessiva, antisimmetrica e transitiva:

- x <= x
- x <= y ^ x >= y ---> x == y
- x <= y ^ y <= z ---> x <= z

Una relazione d'ordine di questo tipo è detta **parziale**. Se per ogni coppia è possibile dare un ordinamento, la relazione è **totale**.

L'ADT Comparator e il metodo compare(x, y) permettono di paragonare due chiavi. Abbinato alle code di priorità lo si può usare per ordinare un insieme. In input ha il comparatore, in output gli oggetti modificati. Il costo dell'algoritmo dipende anche e soprattutto dall'implementazione della coda in sé.

Tra la coda con priorità ordinata e quella non ordinata **cambiano i costi**: in quella non ordinata è O(1) per insert e O(n) per remove(), mentre per la coda ordinata è il contrario. Per decidere quale delle due implementare, si può fare un'analisi probabilistica di quale operazione verrò usata di più. La dimensione delle strutture dati spesso crescono, il che significa che le rimozioni sono normalmente meno numerose degli inserimenti.

I due tipi di coda richiamano il **selection sort** e l'**insertion sort**. Nel selection sort la coda viene ordinata in base alla selezione degli elementi; il costo è O(n^2) poiché per cercare la chiave minima si impiega O(n), e per inserire l'elemento serve ancora O(n). Il costo decresce mano a mano che la coda viene ordinata.

Gli algoritmi di ordinamento possono **operare in place** o non in place. In tal caso il consumo di memoria è legato alla dimensione dell'input. Nell'implementazione non in place è pure così, ma ha bisogno di una memoria oggettiva non in place. I dispositivi con poca RAM spesso costringono il progettista ad ordinare in place. Il selection sort è non in place perché copia la coda su di sé (memoria esterna).

L'altro ordinamento chiave è l'insertion sort. Nell'insert si controlla sempre dove vada a finire l'elemento inserito. Il costo è sempre O(n^2), poiché una "passata" serve per il controllo e una per l'inserimento.

Selection sort e insertion sort possono essere modificati per **renderli in place**. L'insertion sort in place viene realizzato spostando via via gli elementi con priorità maggiore. È però vantaggioso il fatto che gli elementi già ordinati vengano saltati in una data implementazione, quindi il costo minimo è O(n).

Esercizio: formulare l'insertion sort in place.

Tuttavia, nel caso peggiore, l'ordinamento per selection sort in place si presta meglio a dei miglioramenti strutturali che possono velocizzarne l'esecuzione.

HEAP

La **heap** è una struttura pensata per le code di priorità e molto ben implementata. Si presenta come soluzione realizzativa di alberi binari ordinati.
Deve soddisfare due condizioni: la chiave in un nodo non può essere maggiore delle chiavi che si trovano nei sottoalberi; e l'albero deve essere completo, fatto salvo eccezionalmente per l'ultimo livello, nel quale è concesso di lasciare alcuni posti a destra (non devono esserci "buchi"). Negli alberi completi propriamente detti ci sono il 50% di foglie.

Il min() ha costo O(1). Inserendo un nuovo minimo, può avere luogo una violazione dell'ordine.

L'operazione **upheap** permette di ripristinare le condizioni originarie. Quando un nodo, per esempio la root, viene cancellato, il minimo dei figli viene preso come nuovo albero padre. Per quello l'operazione di remove() richiede un metodo a parte per la risistemazione della struttura.

L'operazione **downheap** evita i conflitti che potrebbero insorgere nel riordino dell'albero a partire dall'upheap. Entrambe le operazioni hanno costo O(log(n)).

Gli algoritmi di gestione delle heap, l'**upheap** e il **downheap**, ripristinano la priorità corretta. L'estrazione e ripristino dell'heap avviene in entrambi i casi in tempo Theta(log(n)), pari all'altezza dell'albero che rappresenta la heap.

La rappresentazione naturale dell'heap è un **albero binario**. Ogni coppia di valori di un dato nodo è una entry. Anche nel caso delle heap è possibile una rappresentazione tramite array, seguendo la regola numerica in base alla quale ogni child ha posizione 2i e 2i+1 rispettivamente per il figlio sinistro e il figlio destro del nodo i. Nel caso dell'heap, la rappresentazione tramite array è pressoché ottimale, considerata la struttura.

Ordinare un heap secondo il **PQSort (Priority Queue Sort)**, di tipo insertion o selection, richiede un tempo pari a O(n*log(n)). Tecnicamente il logaritmo sarebbe log(n!), considerato che gli inserimenti in una pila vuota hanno costi leggermente variabili, ma asintoticamente il costo è quasi uguale.

Estrazione ed inserimento sono due operazioni opposte: laddove per l'inserimento c'è il caso migliore, di costo O(1), per l'estrazione si avrebbe il caso peggiore, O(n*log(n)), e viceversa. Si può implementare anche l'HeapSort *in place.*

Due heap della stessa altezza e forma possono essere **fusi** con un nodo aggiuntivo. Quest'ultimo diviene la radice e ha per figli le radici degli altri due alberi; dopo l'inserimento, di tempo costante, si esegue un sort sugli elementi. Se l'albero di sinistra è più piccolo di quello di destra, li si scambia prima di fonderli.

La **costruzione bottom-up** parte da una rappresentazione in array di un heap, probabilmente disordinato. L'ultimo livello del caso in esame ha, supponiamo, 4 heap. Ogni sub-heap viene fuso con un nodo aggiuntivo che è il parent, e si prosegue a riordinarli. Il processo si ripete fino a risalire all'heap ordinato.

Prendiamo in considerazione un algoritmo che visita l'heap da destra a sinistra, in cui ogni nodo viene attraversato due volte nel caso peggiore, Theta(n). Ogni nodo non-foglia funge da head almeno una volta nel ramo.
L'ordinamento crescente e decrescente hanno lo stesso costo.

Con O(n) si può ordinare l'array tramite minHeap, maxHeap e altro algoritmo in place.
Si può estendere la classe minHeap quando ne importa, senza modificarli, i comandi. Inserimento ed estrazione costano al più O(log(n)).

L'implementazione tramite array ha il difetto di **limitare a priori** la taglia della heap. Ogni volta che occorre ampliare la dimensione dell'array bisogna allocarne uno nuovo e riempirlo con tutti gli elementi già inseriti. Anche la classe ArrayList ha questo problema implementativo; si può ovviare decidendo a priori la dimensione del nuovo array, ad esempio 2n. Viene definita **analisi ammortizzata** poiché raddoppiando la dimensione dell'array ci si assicura che l'ampliamento della memoria avvenga molto raramente, permettendo di trascurarne i costi rispetto all'ottimizzazione dei metodi implementati.

MAPPA

Le **mappe** sono un tipo di dato astratto che rappresenta coppie <key, value> per le quali è possibile fare inserimento, cancellazione e ricerca. Non sono ammesse key uguali per due elementi distinti: è un dato univoco che identifica la struttura dati. Hanno vaste applicazioni nei database.

IL TDA mappa ha get(k), put(k, v), remove(k); keys(), values() ed entries() che restituiscono rispettivamente una collezione di chiavi, una di valori e una di coppie <k, v>.
Nel caso particolare della **remove(k)**, si può scegliere se lanciare un'eccezione, restituire null, o non fare nulla quando viene invocata su una chiave non esistente. Il **lancio dell'eccezione** avviene normalmente quando la situazione presente è ingestibile; in questo caso, quindi, essendoci un *workaround* per il problema, è preferibile utilizzarlo. In sostanza, le eccezioni non dovrebbero essere un metodo di programmazione.

I metodi principali delle **mappe** sono get(k), put(k, v), e remove(k). Per preservare la proprietà di unicità della chiave, il metodo put modifica chiavi eventualmente già esistenti. In quel caso restituisce l'elemento vecchio dopo averlo modificato. Si implementano anche gli iteratori del tipo keys(), values(), ed entries().

Le mappe si possono implementare attraverso **liste concatenate bidirezionali** non ordinate. Usa due variabili di riferimento ai nodi (previous e next) e due per chiave e valore. Anche se non c'è ordine, sussiste comunque il problema del controllo della presenza della chiave inserita: non si può inserire in testa o in coda senza verificare che la chiave non ci sia già. Questo comporta che le liste non ordinate abbiano un **costo d'inserimento** minimo pari a **Theta(n)**, come le operazioni di ricerca e remove. Lo si può evitare soltanto se si conosce in anticipo l'elenco delle chiavi.

Alcune delle possibili realizzazioni e i loro relativi costi:

	get(k)	**put(k, v)**	**remove(k)**
Lista non ordinata	O(n)	O(n)	O(n)
Lista ordinata	O(n)	O(n)	O(n)
Array non ordinato	O(n)	O(n)	O(n)
Array ordinato	O(log(n)) ***	O(n)	O(n)

*** L'algoritmo usato è la ricerca binaria (cfr. file .java allegato agli appunti).

Esercizio: scrivere la versione iterativa di binarySearch.

Il problema dell'implementazione via **array ordinato** consiste nel **riempimento**: ogni volta deve essere riallocato con il doppio dello spazio in memoria, costo che va ammortizzato linearmente, compromettendo così il risparmio nel costo sublineare del metodo get(k).

Le **tabelle hash** sono strutture di dati tramite le quali è possibile implementare mappe. Ogni chiave k nelle tabelle è, idealmente, associata **biunivocamente** alla posizione della casella corrispettiva (la struttura è analoga agli array). Questo principio, detto dell'*associazione diretta*, ha alcuni difetti: il primo è la **dimensione dell'insieme** delle possibili chiavi, troppo proibitiva per essere conveniente e per comprendere tutte le possibili chiavi esistenti; in principio si tentò di ovviare a questo problema dimensionando l'insieme sulla base del numero di chiavi attese.

Organizzare le chiavi nell'insieme K che le contiene richiede una regola che determini l'associazione univoca delle chiavi a una posizione precisa dell'array/insieme che le contiene. Questo processo di shuffling deve dare, a parità di input, gli stessi risultati. Tale funzione si chiama **funzione di hashing**. In linea ideale dovrebbe essere possibile avere tutte le

operazioni in tempo costante Theta(1), ma occorre prima risolvere i **problemi di collisione**: è possibile che una tabella hash associ una chiave a una posizione in cui c'è già una chiave.

Le chiavi (stringhe di bit) subiscono un duplice trattamento: la funzione di hashing la trasforma in un numero, dopodiché lo "comprime", ad esempio attraverso l'operazione h(x) = x*mod(N(x)). La parte più complicata è nella trasformazione della chiave in un numero.

La funzione di hashing ha il compito di fare una buona operazione di **shuffling**. Quella di Java, dato un Object, ne restituisce un indice intero che corrisponde all'incirca alla collocazione in memoria. Si può anche partizionare gli elementi in dimensioni fissate e sommarne poi i componenti.
Il costo della funzione di hashing non deve dipendere dalla dimensione della tabella, o scadrebbe nel lineare.

Dati a0, a1, a2, ..., indici di un array, li si può usare come coefficienti di un polinomio di grado n. La variabile x può essere fissata "a piacere" e generalmente si usano numeri dispari. Considerato che x^k = x*x^(k-1), è possibile annidare i calcoli **(algoritmo di Horner)**:
(...(((a9*x) + a8)*x + a7)*x...)*x + a0
Essendo un calcolo dimensionato sulla dimensione delle chiavi, ed essendo la dimensione delle chiavi solitamente costante, l'algoritmo ha un costo pari a Theta(1).

Esercizio: implementare l'algoritmo di Horner in maniera iterativa e ricorsiva.

Le collisioni sono frequenti, come mostra il paradosso del compleanno (in un gruppo di 23 persone la probabilità che due di esse abbiano lo stesso compleanno è maggiore di 0.5).

\\\\\\\\\\\\\\\\\\\\

```
public static int binarySearch(int[] a, int k, int first, int last)
	{
	if(first >= last)
		return -1;
	int center = (first + last) / 2;
	if(a[center] == k)
		return center;
	else if(a[center] > k)
		return binarySearch(a, k, first, center - 1);
	else
		return binarySearch(a, k, center + 1, last);
	}
```

\\\\\\\\\\\\\\\\\\\\

Considerato il *birthday paradox*, si sa che i rischi di collisione delle tabelle hash sono di circa **1.774*n*|R|^(1/2)**. Quindi è fondamentale gestire le collisioni. Un modo per farlo è progettare una struttura con **bucket array**, facendo sì che ad una chiave siano associate massimo k entry; comporterebbe di poter gestire k-1 collisioni al massimo. Si rende quindi necessaria una strategia diversa. Ce ne sono due: *separate checking* (concatenazione separata) e *open addressing* (indirizzamento aperto).

Il **separate checking** consiste nell'associare ad ogni cella di array una **lista collegata** di entry mappate in quella stessa cella. Non bisogna comunque inserire la stessa volta due chiavi: ad ogni operazione di inserimento segue un'operazione di controllo mirata a individuare se l'elemento associato a una data chiave è già presente.

Questa soluzione è ottimale se solo alcune caselle sono abitate da liste, quindi nel caso peggiore le prestazioni sono pari a **O(n)** in inserimento; non c'è grande vantaggio rispetto alla lista unica. La soluzione diventa più conveniente se **la tabella si sovradimensiona**, riducendo così la probabilità che si formino liste lunghe e minimizzando di conseguenza la casistica peggiore.

L'**open addressing** si fonda sulla filosofia in base alla quale un elemento che andrebbe in una casella occupata viene invece posizionato in un'altra casella. È una famiglia di algoritmi. Il più semplice di essi è il **linear probing**, che consiste nel sistemare il dato nella prima casella libera successiva a quella data, ripartendo dalla prima se si raggiunge l'ultima.
Anche in questo caso il costo, nel caso peggiore, è O(n), ma anche qui sovradimensionare la tabella aiuta a evitare il più possibile i casi di collisione. Si presenta inoltre il problema del **clustering primario** (agglomerazione): se due caselle adiacenti sono occupate, la **probabilità di collisione** aumenta; infatti, la probabilità di collisione della seconda casella è pari alla sua probabilità di collisione intrinseca, più la probabilità di collisione della casella precedente. Questo comporta l'inserimento di più elementi e la conseguente creazione di zone molto "popolate".

Per ovviare al clustering primario, si può imporre che lo step di distanziamento tra la casella occupata e la prossima non sia 1. Occorre che lo step e la size della lista siano primi fra loro; ma a livello logico, non c'è differenza: non si ovvia al problema creato. Per questo si introduce il **load factor**, che indica la percentuale della pienezza della struttura; le operazioni sono molto meno efficienti per un LF > 60%.

Il problema sopraggiunge quando bisogna eliminare un elemento: usando l'open addrerssing bisogna scandire l'intero array per notare se l'elemento c'è. Un modo per evitare che succeda è usare un array di booleani che registri la presenza o meno di collisioni. Per far tornare a false la variabile è necessario sapere che non ci sarà più rimozione.
In programmazione viene spesso **ridefinita la mappa**.

Altro algoritmo è il **quadratic probing**. Tramite quadratic probing la distanza tra due chiavi compete, perché aumento con la distanza dal pimo punto. Non risolve il clustering secondario (dato da f = f).

Nel caso di **hashing doppio** il metodo di sfasamento è **(i + jd(k))*mod(N)**. Risolve sia il clustering primario che quello secondario. Il calcolo della funzione hashing, di per sé l'unica componente costosa dell'algoritmo. Quando la tabella è molto piccola, la soluzione primaria, il linear probing; quando è più grande (centinaia di entry) si usa il quadratic probing, ma non è ottimizzato, perché per esempio a load factor >= 0.7.

DIZIONARIO

La generalizzazione dell'idea di mappa è un **dizionario**. I dizionari violano il vincolo dell'unicità della chiave. La loro risoluzione è limitata, poiché vengono da uno studio "discretizzato" del mondo (non si "aggregano" nel senso tradizionale del termini.
L'algoritmo find(k) presenta un problema: come fare a scegliere tra **due chiavi uguali**? Facile, implementare anche findAll(), che li tova tutti(!). Quindi l'inserimento non comporta più una ricerca: O(1).
Il remove fa riferimento alle entry, non alle chiavi. Alcune moderne implementazioni di HashMp Dicitionary usano un metodo a parte, removeAll(), che toglie i duplicati.

	get(x)	**put(x)**	**remove(x)**	**findAll()**
Lista non ordinata	O(n)	O(1)	O(n)	O(n)
Lista ordinata	O(n)	O(n)	O(n)	O(n)
Array non ordinato	O(n)	O(1)	O(n)	O(n)
Array ordinato	O(log(n))	O(n)	O(n)	O(log(n) + k) ***

*** Dipende dalla dimensione in output del dizionario, fino a un massimo di O(n).

I **dizionari ordinati** sono dizionari in cui ogni elemento è legato agli altri da una **relazione d'ordine totale**. L'operazione definita come il successore di una certa chiave è applicabile anche da chiavi che non sono presenti nella struttura, e ne restituiscono comunque i successori; idem per i predecessori. L'ordinamento della tabella comporta il taglio del costo di ricerca da O(n) a **O(log(n))**.

La struttura preferita per l'ordinamento è il **binary search tree**. La proprietà di ordinamento segue la logica: i **sottoalberi sinistri** hanno **solo nodi di key minori**; i sottoalberi destri hanno solo nodi di key maggiori. I nodi con key uguali possono essere distribuite indifferentemente dall'una o dall'altra parte, purché si faccia sistematicamente allo stesso modo per ogni key uguale.

È cruciale per la corretta costruzione dell'albero binario di ricerca che la proprietà sia verificata, per ogni nodo, su **tutti i suoi discendenti sia sinistri che destri**, e per ogni figlio sinistro e destro occorre ripetere la verifica.

Il percorso più lungo possibile durante un'operazione di ricerca è pari all'**altezza dell'albero** (varia da **log(n)**, nel caso di alberi binari completi, a n). Ogni volta si fa il compare tra la key ricercata e la key del nodo corrente; se è **maggiore** quella ricercata si va a destra, se è minore si va a sinistra, se è uguale ci si ferma.
L'algoritmo funziona nel seguente modo: controlla se il nodo ha figli; se non ne ha ritorna null; se ne ha li controlla tutti, sottoalberi compresi.

La **posizione di inserimento** è obbligata nell'albero. Per effettuare l'inserimento è necessaria un'operazione di ricerca, volta a individuarne la posizione. L'operazione richiede anch'essa **O(log(n))**.

Anche la **cancellazione** è preceduta da una ricerca e presenta tre casi distinti: 0 figli, 1 figlio, 2 figli. Se ne ha 0, si rimuove banalmente. Se ne ha 1, il suo riferimento viene cambiato col riferimento al figlio. Infine, se ne ha 2, occorre trovare nell'albero binario di ricerca *la più piccola chiave maggiore di quella da rimuovere*, ossia la key più a sinistra del sottoalbero destro. Va notato che la chiave massima **non ha successore**. Il successore, una volta trovato, viene sostuito al puntatore del nodo da chiamare.

Per verificare se un albero binario è di ricerca o meno, è sufficiente fare una **visita inorder** e restituire le chiavi; se le **chiavi** sono **ordinate**, l'albero binario è di ricerca.

Le prestazioni degli alberi binari di ricerca non sono garantite e potrebbero arrivare fino a O(n^2), ma il **caso medio asintotico** conferma che anche delle chiavi inserite disordinatamente hanno una buona probabilità di raggiungere un ordinamento che comporti un costo logaritmico.

ALBERI BINARI DI RICERCA (BINARI, AVL, FIBONACCI)

Gli **alberi binari** di ricerca sono alberi che memorizzano chiavi (o entry **<key, value>**) nei nodi interni secondo la proprietà: dato un nodo e i suoi due figli, quello **sinistro** contiene solo **key inferiori** e quello destro contiene solo key superiori. Le operazioni supportate sono find e **findAll**; la prima trova una qualsiasi entry associata a una data key, la seconda le trova tutte. La chiamata ricorsiva (o aggiornamento del puntatore, nel caso di cicli) avviene a seconda del risultato della compare a destra o a sinistra, oppure da entrambe le parti se la regola è <=, >=.

Output corrente * findAll(nodo corrente.left()) * findAll(nodo corrente.right())
Dove * è il simbolo della concatenazione.
Il **costo** dell'algoritmo findAll è **O(h + k)**, ove h è l'altezza e k è la numerosità delle key presenti per il valore v. Il caso in cui la regola sia **<=, >=** ha un costo leggermente più alto per via della potenziale ricorsione binaria, ma ci sono casistiche – che esamineremo più avanti – per le quali l'implementazione di questa regola è un requisito.

A tempo d'inserimento, con la regola <=, >= è possibile decidere arbitrariamente dove aggiungere una chiave uguale a un'altra. Le operazioni di **cancellazione** sono immediate nel caso di zero figli e un figlio, ma nel caso di **due figli** occorre definire il successore come la minima tra le chiavi >=, e il predecessore come la massima tra le chiavi <=.

Calcolando la **media delle altezze** di tutti i nodi viene restituita una media **logaritmica**. Questo implica che il costo atteso di ciascuna operazione è logaritmico, ma non è una garanzia.

Per ovviare a questo inconveniente si fa una **range query**, o interrogazione d'intervallo: dati in input [a, b] vengono restituiti i nodi contenenti una key appartenente all'intervallo. Le range query si basano sulle **visite inorder**; non ha senso, tuttavia, scendere a sinistra di chiavi minori di a né scendere a destra di chiavi maggiori di b. Perciò:

```
if(value == null) return;
else if(key < a) inorder(node.right());
else if(key > b) inorder(node.left());
else inorder(node.left()); visit(key); inorder(node.right());
```

Massimizzare la velocità significa minimizzare l'altezza h, il che si traduce nell'utilizzo di **alberi bilanciati**, di cui tratteremo gli **alberi AVL**, il tipo originario di albero bilanciato. Esistono tanti alberi bilanciati, tra cui i red-black. Il principio degli alberi bilanciati è che data l'altezza p di uno qualsiasi dei suoi sottoalberi, gli altri **sottoalberi** possono avere un'**altezza** pari a **[p−1, p+1]**. Ogni nodo memorizza quindi anche l'altezza. Questa proprietà vale nell'intero albero.

Il fattore di (s)bilanciamento è l'altezza del sottoalbero destro meno quella del sinistro, compresa in [-1, +1]. Un fattore diverso implica che l'albero non è bilanciato.

Gli **alberi di Fibonacci** sono molto vicini allo sbilanciamento: "potando" le sue foglie si ottiene il corrispettivo sottoalbero, il prossimo più grande. Il risultato è che ciascun albero ha un sottoalbero che ha un numero di nodi pari al numero precedente nella serie di Fibonacci, diminuito di uno (es. 0, 1, 2, 4, 7, 12, 20, 33...). Nel totale dell'albero ci sono quindi (Fib(i)-1)+(Fib(i+1)-1) + 1 nodi. Quindi il **totale dei nodi** è **Fib(i+2)-1**. Ne deriva che l'altezza di un albero di Fibonacci è l'upper bound dell'altezza di un albero bilanciato. Tale upper bound coincide col lower bound: O(log(n)), dimostrabile formalmente.

Gli AVL sono alberi di ricerca bilanciati. Nelle operazioni d'inserimento a volte si rende necessario il **ribilanciamento**, che avviene a opera di **rotazioni**. Ne esistono di quattro tipi: **sx-sx**, **sx-dx**, **dx-dx**, **dx-sx**. Occorre tenere conto del fatto che con ciascuna operazione, potenzialmente, si potrebbe sbilanciare l'intera struttura che lo collega alla radice.
Ad esempio:

```
|--4
   |--8
      |--11
```

Essendo un albero sbilanciato a destra, effettuiamo una rotazione dx-dx che lo modifica in:

```
|--8
   |--4
   |--11
```

Occorre controllare ora se è un albero binario di ricerca. Assumiamo che quest'albero sia un sottoalbero destro di un albero binario di ricerca: la riorganizzazione non comporta danni, perché i figli destri sono tutti *maggiori della radice*.

Esistono anche casi più complessi:

```
|--R
   |--C
      |--A
         |--Alpha (sottoalbero di altezza h ove avviene l'aggiunta)
      |--Gamma (sottoalbero di altezza non specificata, ma regolare)
   |--B
      |--Beta (sottoalbero di altezza h)
```

Questa struttura richiede che **C diventi root**, **R diventi sottoalbero sinistro di C**, e **B diventi sottoalbero destro di R**. Ciò non comporta problemi perché Gamma, essendo sottoalbero di C che è sottoalbero destro di R, ha tutti elementi più grandi di R e più piccoli di C. Il caso sx-sx è speculare a questo.
La massima altezza di Gamma è h+1.

La **rotazione DD** avviene in corrispondenza dell'aggiunta di un nodo a un sottoalbero AVL dal lato destro, così come la **rotazione SS** avviene per il sinistro. Nel caso del destro, ogni nodo di fattore di bilanciamento pari a +1 dello stesso ramo del nodo inserito viene sbilanciato, poiché tale fattore diventa +2.
Gli sbilanciamenti vanno corretti **al più al nodo "grandparent"**, poiché la rotazione così effettuata preserva l'altezza originaria dell'albero, bilanciando automaticamente anche gli altri nodi.

Ad esempio:

|--49

|--49
    |--76 (right)

|--49
    |--22 (left)
    |--76 (right)

|--49
    |--22 (left)
        |--16 (left)
    |--76 (right)

|--49
    |--16 (left)
        |--8 (left)
        |--22 (right)
    |--76 (right)

Dopodiché da qui:

|--49
    |--16 (left)
        |--8 (left)
            |--3 (left)
        |--22 (right)
    |--76 (right)

Si giunge qui:

|--16
    |--8 (left)
        |--3 (left)
    |--49 (right)
        |--22 (left)
        |--76 (right)

Rotazione DS:

|--19

|--19
    |--82 (right)

|--19
    |--82 (right)
        |--37 (left)

|--19
    |--37 (right)
        |--82 (right)

|--37
    |--19 (left)
    |--82 (right)

Altro esempio:

|--37
    |--19 (left)
        |--5 (left)
            |--10 (right)
    |--82 (right)
        |--90 (right)

|--37
    |--19 (left)
        |--10 (left)
            |--5 (left)
    |--82 (right)
        |--90 (right)

[...]

|--37
    |--10 (left)
        |--5 (left)
            |--2 (left)
                |--0 (left)
            |--8 (right)
        |--19 (right)
            |--25 (right)
    |--82 (right)
        |--50 (left)
        |--90 (right)

Dopodiché il 10 diventa la radice; in sostanza **il ramo "in mezzo" viene staccato** dalla nuova radice e attaccato al sottoalbero con un solo figlio a cui appartiene.
Anche nelle **rotazioni doppie** la proprietà di **preservazione dell'altezza** si conserva.
Se l'inserimento è a sinistra di un nodo, la rotazione avviene prima verso destra *(il genitore del nodo inserito diventa il suo figlio destro, e il progenitore del nodo inserito diventa il suo figlio sinistro)* e poi verso sinistra *(il nodo più in basso e più esterno viene sostituito al suo genitore, che diventa suo figlio)*, e viceversa.

Il **test dell'altezza** si può effettuare in **Theta(log(n))**: per controllare l'altezza o il fattore di bilanciamento di un dato albero, è sufficiente controllare il suo albero genitore con tutti i relativi progenitori, che la memorizzano come variabile d'istanza.

Il caso della **cancellazione** è più complesso da ristrutturare. La cancellazione del nodo esterno più in alto è il caso più complesso, soprattutto nel caso di alberi di Fibonacci. L'operazione di rotazione seguente riporta l'albero in una situazione il più vicina possibile alla completezza. Anche il ribilanciamento a seguito di una cancellazione è pari a **Theta(log(n))**.

Un algoritmo per determinare se un albero è bilanciato o meno è incluso nel file .java allegato.

\\\\\\\\\\\\\\\\\\\\\\\\\\\\

```
boolean isAVL()
	{
	return isBST() && isBalanced();
	}

boolean isBST()
	{

	}

boolean isBalanced()
	{
	return !(isBalanced(root) <= -1 || isBalanced(root) > 1);
	}

int isBalanced(Node n)
	{
	if(n == null) return 0;
	if(n.isLeaf()) return 1;
	int leftHeight = isBalanced(n.left);
	int rightHeight = isBalanced(n.right);
	if((leftHeight <= -1) || (rightHeight <= -1)) return -1;
	int b = Math.abs(rightHeight - leftHeight);
	if(b > 1) return -1;
	return Math.max(leftHeight, rightHeight)+ 1;
	}
```

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

ALGORITMI DI ORDINAMENTO

MERGE SORT

L'algoritmo di ordinamento **merge sort** fa parte della categoria dei *divide et impera*, la cui idea di base è la suddivisione in regioni più piccole per poi organizzare localmente. **Non** è un algoritmo **in-place**.

Ci sono problemi e **classi di problemi**: un esempio di classe di problema è la risoluzione di un'equazione di secondo grado, mentre un analogo esempio di problema può essere una particolare equazione di secondo grado, ossia un'*istanza* della classe di problemi.

Il merge sort divide quindi il problema in **due o più sottoproblemi** equamente ripartiti, itera il procedimento fino a risolverli, e poi unifica i risultati ottenuti. Ad esempio, un problema di ordinamento su array si riduce in ultima analisi a un albero di chiamate ricorsive che giunge a ordinare la singola casella dell'array e unificare le caselle così ordinate.

Il costo del merge sort è **O(n*log(n))**, il che lo profila come uno dei migliori algoritmi di ordinamento esistenti. In caso di elementi dispari in numero, l'albero binario delle ricorsioni ovviamente non è completo, ma risulta in ogni caso bilanciato. Lo pseudocodice è:

```
Algorithm mergeSort(Sequence, Comparator)
Sequence has n elements
if(S.size > 1) partition(S, n/2);
                mergeSort(S1); mergeSort(S2);
                merge(S1, S2);
```

L'idea fondamentale del merge si basa sul continuo confronto tra le due sequenze ordinate: ogni volta viene preso l'**elemento minore tra i due alla testa** delle sequenze ordinate, e viene anche rimosso dalla sequenza da cui è stato estratto e messo nell'insieme unione, il cui *indice di scorrimento* viene aumentato (non è ovviamente aumentato l'indice del sottoinsieme da cui non è stato estratto alcun elemento nella corrente chiamata al metodo).
Una volta **esauriti gli elementi** di uno dei due insiemi, un **ciclo while** completa le operazioni di inserimento prendendo tutti gli elementi rimanenti dell'altro insieme.

La suddivisione degli elementi avviene in **preordine**, mentre la fusione avviene in **postordine**. I costi del merge sort derivano dalla suddivisione in albero ricorsivo per la parte logaritmica, più l'ordinamento delle due metà per la parte lineare, risultando nell'atteso O(n*log(n)).

L'**equazione di ricorrenza** (controparte discreta delle equazioni differenziali) del merge sort stabilisce che la sua funzione di costo è:

```
f(n) = { Theta(1)              if(n <= 2)
       { 2*Theta(n/2) + c*n    else
```

Un altro esempio può essere l'equazione di ricorrenza della ricerca binaria:

```
f(n) = { Theta(1)              if(n == 1)
       { Theta(n/2) + c        else
```

Si noti che **le equazioni di ricorrenza sono modellate su algoritmi ricorsivi**, quindi non esistono per algoritmi ciclici. Inoltre, **non sono definite per gli alberi** se non per alberi binari completi o altrimenti omogenei, perché le due ripartizioni devono essere omogenee.

Passi per calcolare la funzione relativa al merge sort:
f(2) = 4*T(n/4) + 2*c*n; f(3) = 8*T(n/8) + 3*c*n; ...; f(k) = 2^k*T(n/2^k) + k*c*n =
= c1*n*log(n) + c2

Questa tecnica viene detta tecnica dello **srotolamento**.
Riepilogando:

Algoritmo	Tempo	Proprietà
Selection sort	O(n^2)	• Lento • In-place • Per insiemi piccoli di dati
Insertion sort	O(n^2) Caso parzialmente ordinato: O(n)	• Lento • In-place • Per insiemi piccoli di dati
Heap-sort	O(n*log(n))	• Veloce • In-place • Per insiemi grandi di dati
Merge sort	O(n*log(n))	• Veloce • Non è in-place • Per insiemi grandi di dati

Il merge sort può essere realizzato anche **iterativamente**: immaginando di vedere l'insieme già suddiviso, si effettua l'ordinamento su **blocchi da due elementi** e successivamente sulle coppie di blocchi ordinati. L'ottimizzazione così comportata permette di evitare l'utilizzo della stack.

Una bozza di mergeSort (incompleto) è inclusa nel file .java allegato.

\\\\\\\\\\\\\\\\\\\\\\\\\

```
void mergeSort(int[] a, int i, int j)
	{
	if(i < j)
		{
		int m = (i + j) / 2;
		mergeSort(a, i, m);
		mergeSort(a, m + 1, j);
		}
	}

int[] merge(int[] a, int[] b, int[] c, int i, int j, int k)
	{
	if(i >= a.length)
		{
		while(j < b.length)
			{
			c[k] = b[j];
			j++;
			k++;
			}
		return c;
		}
	if(j >= b.length)
		{
		while(i < a.length)
			{
			c[k] = a[i];
			i++;
			k++;
			}
		return c;
		}
```

```
if(a[i] > b[j])
        {
        c[k] = a[i];
        i++;
        k++;
        }
else
        {
        c[k] = a[i];
        i++;
        k++;
        }
return merge(a, b, c, i, j, k);
}
```

\\

QUICKSORT

L'algoritmo di ricerca **quick sort** è implementato sempre secondo il *divide et impera:* sceglie un **elemento casuale**, chiamato *pivot*, e partiziona l'insieme in elementi maggiori, minori e uguali. Dopo di questo, ordina ricorsivamente e fonde le sequenze ordinate. In tal modo, l'unico insieme totalmente ordinato è quello al centro, il gruppo di elementi uguali al pivot.

L'efficienza di questo algoritmo dipende dalla **scelta del pivot**: se si riesce a partizionare l'insieme in due parti pressoché uguali, si minimizzano le chiamate ricorsive, altrimenti massimizzate se il pivot scelto è il minimo o il massimo dell'insieme. Il caso peggiore è **O(n^2)**, quello migliore è **O(n*log(n))**.

Il primo passo è il **partizionamento** dell'insieme. Tutti gli elementi maggiori vengono accodati all'insieme, mentre tutti gli elementi minori vengono aggiunti in testa. Anche il quick sort può essere rappresentato come albero binario.

Un modo per evitare il caso peggiore è scegliere tre elementi a caso della sequenza, confrontarli e scegliere il **valore *mediano*** dei tre. In tal modo si è sicuri che il pivot scelto non sarà né il minore né il maggiore. Sebbene in informatica si sia portati a fare analisi di **caso peggiore**, nel caso del quick sort è pessimistico pensarlo: le probabilità del caso peggiore sono ridottissime.

Il pivot può essere il risultato di una buona scelta o di una cattiva scelta: nel primo caso, la dimensione di ciascun insieme non supera i 3/4 degli elementi del sottoinsieme d'origine. Quindi, a ogni passo, la probabilità di fare una **buona scelta** è **1/2**.

Il valore atteso di quick sort è O(n*log(n)); questo caso si verifica molto spesso e le costanti moltiplicative sono inferiori al merge sort, il che lo rende generalmente preferibile.

È possibile effettuare il quick sort **in-place**: al posto di usare strutture d'appoggo, si usano indici per scandire ad esempio l'array dall'iinzio e dalla fine. Appena scelto il pivot, lo si scambia col primo elemento per semplicità. Anche gli elementi delle **due metà** che dovrebbero trovarsi nell'altra metà si scambiano. Successivamente il pivot viene reinserito a metà tra le due sequenze.

Per il passo base, da pochi elementi, viene solitamente usato l'**insertion sort**.

Il **lower bound** non è sempre semplice da determinare: una delle tecniche di base è l'albero binario di ricerca. Si basa sul confronto: x1^2 > x2 => y1 < y2^2 e così via. Il numero di foglie dell'albero è **n!**, tutti i possibili ordinamenti di tutti gli elementi.

Dato che l'altezza dell'albero è almeno pari al logaritmo delle foglie, che sono la metà dei nodi, log(n/2) <= h, il che significa che l'altezza minima dell'albero risulta essere **O(n*log(n))** che è il **lower bound** degli **algoritmi di ordinamento**. Lo stesso procedimento si può ripetere per gli algoritmi di ricerca, provando che il loro lower bound è **O(log(n)).**

Non è necessario disegnare l'albero, basta **indicarne le foglie** e il **lower bound** (che si indica con Omega).

Il problema della **selezione** riguarda l'individuazione dell'elemento k-esimo di una sequenza. Ha un *upper bound* di Theta(n*log(n)), perché ordinando l'insieme l'elemento viene ovviamente rintracciato; tuttavia, è stato provato che è possibile giungere a un costo **O(n)** ossia trovare l'elemento senza ordinare l'insieme.

QUICKSELECT

L'algoritmo **quickSelect** si basa sullo stesso principio del quickSort: prende un **elemento a caso** nell'insieme (*pruning* del pivot) e divide il resto degli elementi in maggiori e minori. Dopodiché l'analisi prosegue nel **sottoinsieme** dato: il pivot ha una sua posizione nell'insieme, quindi si sa se, a partire da esso, occorre andare a sinistra (insieme dei minori) o a destra (insieme dei maggiori).
Proseguendo nei minori, si cerca sempre l'elemento k; proseguendo nei maggiori, si sottraggono a k la cardinalità degli elementi minori e uguali, e si cerca l'elemento k'.

Nel caso peggiore l'algoritmo richiede **O(n^2)**, ma il caso medio è lontano dal caso peggiore. Le **buone chiamate** del metodo sono quelle che, come nel quickSort, suddividono l'insieme in porzioni entrambi grandi almeno 1/4 del totale, il che avviene con probabilità 1/2.

Il calcolo del tempo medio di esecuzione è:

*Tn <= T(3n/4) + bn * (chiamate attese prima di una chiamata buona)*

Quindi in tal caso: Tn <= T(3n/4) + 2bn
Applicando il procedimento dato: T(3n/4) + 2bn <= T((3n/4)^2) + 2*3n/4*b + 2bn =
= T((3n/4)^2) + 2bn(1 + 3/4) <=
<= [...] = T((3n/4)^k) + 2bn(Sum k=0, k->+inf((3n/4)^k))

Essendo una **serie geometrica**, converge. Quindi il lower bound è **O(n)**.

Un caso interessante è fare il mediano di una sequenza, oppure sia il primo che il secondo elemento. Nel secondo caso, è utile ricostruire l'**albero** per trovare l'elemento più "piccolo" (il primo) in modo tale da spendere O(n) per trovarlo, dopodiché confrontare tra loro tutti gli elementi che hanno "perso" al confronto col primo, in numero e costo pari a O(log(n)).

"Bob" Tarjan fu il primo a scrivere un algoritmo lineare per il quickSelect: divideva l'insieme in n/5 sottoinsiemi di 5 elementi ciascuno, poi trovava il mediano e il mediano dei sottoinsiemi, e così via.

INSIEMI

Gli **insiemi** esistono anche come tipo di dato astratto. Le operazioni più importanti da essi supportate sono l'**unione** e l'**intersezione**. Hanno un criterio di ordinamento arbitrario (che può variare dalla locazione occupata in memoria all'hash dell'oggetto).
L'operazione di unione si basa sullo stesso principio del merge di ordinamento; l'unica differenza è che all'incontro di due elementi uguali ne aggiunge **uno solo**. L'algoritmo di intersezione fa esattamentente il contrario. I costi di intersezione e unione sono O(n).

Esiste anche una rappresentazione degli insiemi composti da **vettore caratteristico**, un array mono-dimensionale booleano che determina, per ogni oggetto, se quell'oggetto appartiene all'insieme o no.
Una particolare modalità, molto efficiente, di gestire gli insiemi è la **union find**. Prende in ingresso una collezione di insiemi disgiunti e ne crea un insieme composto, dopodiché li unisce e dato un elemento tova anche a quale set appartiene. Prima di usare makeSet è quindi opportuno il find().

Rappresentando gli **insiemi disgiunti** come **liste**, l'unione richiede comunque O(n) perché il puntatore di ciascuno degli elementi della seconda lista deve ora puntare alla prima. Per questo viene preso l'insieme col minor numero di elementi, e i puntatori di tutti i suoi elementi vengono aggiornati.

Gli elementi non possono essere **uniti** un numero indefinito di volte: il costo di aggiornare il suo puntatore e l'arrivo in un insieme grande il doppio di quello precedente determinano il fatto che, dopo il decimo reinserimento, l'elemento giunga a n. Giungervi costa al più O(log(n)).

Gli **alberi generici** possomo rappresentare gli insiemi. In tal caso l'albero viene implementato "al contrario", col **riferimento al genitore** anziché al figlio, e la radice (che fa riferimento a se stessa) sorregge la struttura. Inoltre, non rispettare le specifiche sopra riportate permette di velocizzare le operazioni: basta attaccare la radice alla nuova radice per effettuare l'unione.
Secondo **union-by-size**, l'insieme con più elementi viene attaccato; è più comune **union-by-rank**, in cui la radice con rank minore viene attaccata a quella con rank maggiore, così da mantenere l'altezza O(log(n)).

Una volta collegate due radici, può essere necessario *aggiornare il rank* (altezza) dell'albero a seguito di una fusione. Il rank deve essere **mantenuto basso** per evitare costi eccessivi di find(). Una volta eseguito find() per una struttura a rank alto si può utilizzare la **path compression**, volta a spostare i riferimenti di alcuni nodi verso la radice, per ridurre il rank. Tuttavia, visto che aggiornarlo richiederebbe una visita di tutti i nodi, non viene aggiornato.

Il calcolo delle prestazioni per gli insiemi è complesso e fa riferimento alla **funzione di Ackerman** che, in ultima analisi, riconduce i costi di ciascuna operazione a costi lineari.

In **makeSet** il rank viene inizializzato a 0 e viene dato il nome all'insieme secondo il suo elemento. L'operazione **union** si effettua trovando il riferimento agli insiemi x e y (la loro radice) e successivamente collegandoli.
L'algoritmo **findSet** ritorna x se x è uguale al suo parent, altrimenti ritorna il findSet invocato sul parent di x, al quale assegna anche il ritorno del metodo *(path compression)*.
Infine, **link** confronta i rank della radice per legare l'albero più basso alla radice del più alto, aggiornando il suo rank.

L'implementazione di questi algoritmi è **flessibile**: come si possono realizzare con alberi, si possono realizzare con array, oppure alberi basati su array. Non è necessaria alcuna struttura specifica.

GRAFI

I **grafi** sono coppie <V, E> dove V è un insieme di nodi (o **vertici**) ed E è una collezione di **archi**. Entrambi sono posizioni e contengono elementi: ad esempio si può pensare ai vertici come a delle città e agli archi come ai percorsi dall'una all'altra, definiti dalla loro lunghezza.
Sono spesso associati ai **multi-insiemi**, ossia insiemi che ammettono duplicati. È ammissibile anche un arco che punta al suo stesso vertice *(cappio)*.

La rappresentazione grafica dei grafi è arbitraria: sia vertici che archi possono essere disegnati in molti modi diversi. È comunque un problema importante nell'informatica il *graph drawing*, ossia la rappresentazione più conveniente di un grafo. Ci si concentra in modo particolare su come **evitare sovrapposizioni di archi** nel disegno. Famoso è il problema del grafo K(3, 3), con tre case e tre centrali: non ammette soluzioni senza sovrapposizioni di archi.

Il **grafo orientato** o **diretto** ha frecce su almeno un arco; viceversa, non è orientato. Col termine **spigolo** si indica un nodo che non è puntato da una freccia, mentre vertice indica un nodo puntato da freccia (la distinzione tra i due non sussiste ai fini del corso).

Un grafo è essenzialmente una **relazione binaria R = V x V** su V. Ogni coppia è ordinata, ovvero (Vi, Vj) != (Vj, Vi). Quindi, dal punto di vista matematico, esiste *solo il grafo orientato*.
Viene tuttavia soddisfatta la **proprietà simmetrica**: se (a, b) appartiene alla relazione, ossia esiste un arco tra a e b, allora anche (b, a) vi appartiene, ovvero c'è anche l'arco di ritorno. Il grafo non è orientato, essenzialmente, se l'arco di andata coincide con l'arco di ritorno.

La coppia ordinata viene indicata con (a, b) oppure <a, b>. Quella non ordinata viene indicata come {a, b}, ovvero come insieme.

Due vertici collegati sono detti **adiacenti**. Il **grado** dei vertici è il numero dei suoi archi. I **cappi** sono archi che puntano allo stesso vertice. Si usa il termine **percorso** o **cammino** per indicare una sequenza vertice-spigolo-vertice... mentre i **cicli** indicano la ricorrenza di un dato percorso. I vertici si indicano con **n** e gli spigoli con **m**.

In un grafo non orientato, senza self-loop e spigoli multipli, m <= n(n-1)/2
Questo significa che il numero massimo degli spigoli è quadratico. Un grafo di ordine di n^2 archi è detto **denso**, quello di ordine n è detto **sparso**.
Se un grafo si può disegnare in maniera planare, è sparso.

Il grafo diretto ha m + (n(n+1)). Metodi principali:

- insert (vertex ed edge)
- remove (vertex ed ed edge)
- vertices () ed edges():
- endVertices
- replace

Ci sono tre implementazioni fondamentali per i grafi. Una si basa su tre array: quello contenente gli elementi, quello per gli oggetti e i riferimenti agli oggetti collegati, e quello per gli edge. Quella con gli array ha però la limitazione di **non saper trovare tutti gli spigoli** adiacenti a un vertice in **tempo veloce**.

Si può quindi cambiare a favore della rappresentazione in **lista delle adiacenze**, ove ogni vertice ha un puntatore aggiuntivo costituito dalla lista collegata di tutte le sue incidenze. Questo comporta che, per trovare n nodi adiacenti, è necessario un tempo pari a **O(n)**, con O(1) per nodo.

Ancora più famosa è la **matrice delle adiacenze**, in cui esistono ancora la lista delle adiacenze e la lista degli archi, ma si aggiunge una matrice nxn di boolean in cui il valore è true se l'arco tra Vi e Vj esiste, e false se non esiste tra Vj e Vi. La proprietà di simmetria si

riflette anche nella matrice, **simmetrica** rispetto alla sua diagonale, che può rappresentare anche autoanelli e può essere allocata solo per metà se il grafo è simmetrico.

L'operazione "trova tutte le adiacenze" si svolge in Theta(n), perché bisogna controllare tutte le adiacenze di un dato vertice. Nel complesso l'implementazione via matrice delle adiacenze è sconveniente: esegue la **verifica di adiacenza** in O(1), gli altri tempi però sono almeno pari agli altri due. L'implementazione è quindi adeguata a un utilizzo che prevede tante verifiche di adiacenza; altrimenti è preferibile la rappresentazione in **lista delle adiacenze**.

	Lista spigoli	**Lista adiacenze**	**Matrice adiacenze**
Spazio	m + n	m	n^2
incidentEdges(v)			
areAdjacent(v, w)	m	min(m, n)	1
insertVertex(o)			n^2
insertEdge(v, w, o)	1	1	n^2
removeVertex(o)			

Il **sottografo** G' è un grafo tale che i suoi vertici e spigoli sono un sottoinsieme di quelli di G.

Un **sottografo ricoprente *(spanning graph)*** è un sottografo che include tutti i vertici di G.

Un **sottografo indotto** o **sui vertici** ha un sottoinsieme dei vertici e tutti gli archi di G.

Un grafo è **connesso** se esiste un percorso tra ogni coppia di vertici, altrimenti può avere delle componenti connesse.

Un **albero libero** è un grafo non orientato connesso e aciclico, *non-rooted*, ossia senza un'orientazione (come gli alberi studiati con radice). Una **foresta** è un grafo non orientato aciclico, le cui componenti connesse sono alberi. Un **albero ricoprente** è un sottografo ricoprente che è anche un albero, e non sempre esiste; la foresta ricoprente è l'estensione alle foreste del medesimo concetto.

Costruire un albero ricoprente è il primo passo per la *visita in profondità* di un grafo.

DFS -- *depth-first search* -- *(algoitmo per spanning trees)*

L'**algoritmo DFS** visita i grafi e consulta le etichette assegnate a vertici e spigoli. Per evitare di rivisitare lo stesso spigolo o lo stesso vertice, si usa una variabile **boolean explored** inizializzata a *false*. L'esplorazione viene lanciata ciclicamente dal metodo principale a partire da un **vertice casuale**, esplorando tutti gli spigoli e i vertici possibili; poi si ricontrolla se il grafo è stato totalmente esplorato e, in caso contrario, l'esplorazione viene rilanciata a partire dal primo vertice inesplorato.

L'algoritmo che visita effettivamente il grafo è ricorsivo. La prima azione compiuta è il cambiamento della variabile explored. Successivamente il metodo viene chiamato nel caso dei vertici per ogni spigolo incidente inesplorato, nel caso degli spigoli per ogni vertice incidente inesplorato.

Supponiamo di avere il seguente grafo:

Si parte da 1, chiamando ricorsivamente per **tutti gli archi incidenti** a partire da uno a caso, il cui label è posto su *DISCOVERY* se sono precedentemente *UNEXPLORED*. Se uno spigolo UNEXPLORED porta a un vertice visitato, lo si marca come *BACK*. **Finite le marcature**, si controlla ciclicamente se ci sono altri vertici inesplorati; se ci sono, l'esplorazione riparte da lì.
Le componenti connesse sono **spanning forests**. Se il grafo è connesso, si parla di **spanning trees**.

Il **costo dell'algoritmo** si calcola approssimativamente considerando che: il ciclo sui vertici costa n; quello sugli spigoli costa m; ogni vertice o spigolo genera al massimo **deg(x)** chiamate ricorsive *(con l'opportuna implementazione dell'algoritmo che trova le adiacenze, quello basato sulle liste di adiacenza)*. Si tratta quindi di un O(m). Sommando i costi si ottiene **O(n + m)** o, con maggiore precisione, **Theta(n + m)**, considerato che nessun vertice o spigolo viene visitato due volte.

GRAFI E LABIRINTI

L'algoritmo DFS può essere usato per ***esplorare labirinti***. A ogni incrocio si va a sinistra, e a ogni passaggio si marca come *attraversato* il percorso. Il metodo ricorsivo richiede un parametro aggiuntivo, z, che rappresenta la **configurazione finale**, in modo tale da confrontarla con la configurazione attuale. Una struttura a pila memorizza il **percorso effettuato** per giungere al nodo corrente.

Per sapere se **un grafo è ciclico** o meno basta aggiungere un controllo sulla visita: se viene posto anche un solo label BACK, il grafo contiene cicli. Si può anche memorizzare in una **pila** i vertici e gli spigoli attraversati; poi si effettuano operazioni di **pop** appena viene posto un BACK fino a che non si ritrova il vertice a cui conduce lo spigolo BACK, restituendo così tutti gli elementi del ciclo. Questa tecnica trova un solo ciclo; trovarli tutti comporterebbe costi molto maggiori.

I **grafi diretti** sono grafi orientati; ogni loro spigolo va in una sola direzione. In questo caso si distingue tra indeg(x) e outdeg(x).
I digrafi sono utili nella **pianificazione** *(scheduling)*, per decidere quale attività deve essere svolta prima che ne inizi una nuova. L'inizio avviene da un nodo tale che indeg(x) = 0, la cui presenza non è garantita nel digrafo; inoltre è necessario che **non siano presenti cicli**.
La DFS è applicabile anche ai digrafi, con la differenza che viene utilizzata ovviamente solo per gli spigoli orientati all'esterno (outdeg(x)).

Ci sono due tipi di **connessione** nei grafi: la connessione **forte** e la connessione **debole**. La connessione forte prevede che *ogni vertice conduca a un qualsiasi altro vertice*; la connessione debole prevede che *il corrispettivo grafo non orientato sia fortemente connesso* (ovvero che ci sia almeno un collegamento, sia esso in ingresso o in uscita, per ogni vertice).

Nei digrafi, le DFS attraversano tutti i vertici raggiungibili da un vertice dato. Se **due spigoli portano allo stesso vertice** e hanno lo stesso spettro di esplorabilità del grafo, il secondo di quelli visitati viene etichettato come *CROSS*.

La visita di un grafo costringe prima o poi a tornare al vertice iniziale; di qui il concetto di **abbandono definitivo** di un vertice: avviene quando un vertice non si visita più. Anche qui è possibile memorizzare i vertici abbandonati e restituirli nell'ordine di abbandono. I cicli riportano quindi ai nodi già visitati.

La verifica di **connettività forte** avviene verificando che la visita sia completata dopo aver finito le ricorsioni. Il trasposto di un grafo connesso è connesso. Anche un grafo debolmente connesso ha delle parti fortemente connesse. L'algoritmo per individuarle prevede la DFS del grafo originario più la **DFS del grafo trasposto**, effettuata a partire dai vertici abbandonati. Il costo asintotico è identico a quello delle DFS per grafi non orientati.

La **chiusura transitiva** di un **grafo diretto** è una proprietà per cui, se esiste un cammino dal nodo a al nodo b, allora esiste un arco tra a e b. È un problema d'interesse la verifica della chiusura transitiva di un grafo, così come la trasformazione di un grafo non transitivo in un grafo transitivo e viceversa *(riduzione transitiva)*.

Un primo approccio può essere la **DFS da ogni vertice**, di costo O(n(n + m)), ricollegando a ogni nodo un arco che lo connetta a ogni altro suo nodo non adiacente. Nel **caso peggiore**, ovvero se siamo in presenza di un **grafo denso** (che ha un numero di nodi pari a O(n^2)) il costo è **Theta(n^3)**, mentre nel caso migliore, con un grafo sparso (numero di nodi pari a O(n)) impiega **Theta(n^2)**.

PROGRAMMAZIONE DINAMICA --> *(Floyd-Warshall, DAG, topological sort)*

L'alternativa è usare un approccio basato sulla **programmazione dinamica**. L'osservazione basilare è una semplice proprietà transitiva: *se il nodo i è collegato a j e j è collegato a k, allora i è collegato a k*. Adottando un particolare ordine di controllo dei nodi è possibile ottimizzare l'algoritmo: si controlla ogni nodo con questo metodo "triadico", costruendo un **grafo "parallelo"** realizzato in chiusura transitiva.

L'algoritmo più famoso in questo senso è quello di **Floyd-Warshall**: dà un ordine casuale al grafo, mettendo etichette numeriche su tutti i nodi, e combina in ogni modo possibile i nodi presi, costruendo su una struttura transitivamente/direttamente collegata. Ciclando su i, j e k, si ottiene un costo pari a **O(n^3)**. L'input normalmente è di dimensione O(n^2), il che significa un costo complessivo in funzione dell'input pari a **O(z*sqrt(z))**. Tale costo vale per i grafi densi; per i grafi sparsi si mantiene su O(n^3).
Si tenga in conto che ogni successiva modifica del grafo deve essere aggiornata in tutti i riferimenti.

Un **DAG (directed acyclic graph)** descrive un *partially ordered set*. In esso è definita una **relazione d'ordine parziale**, che soddisfa la proprietà transitiva. È possibile quindi effettuare un **ordinamento topologico** sull'insieme, basato proprio su quella relazione d'ordine.

Il **topological sort** si basa sull'osservazione dell'ordine nei nodi (generalmente dato dalla loro key()). L'algoritmo prosegue cercando un **nodo-pozzo**, ovvero un nodo senza nodi entranti; poi lo si rimuove assieme a tutti gli archi che lo collegano all'albero, assegnandogli il valore n-k con k numero corrente di iterazioni, finché non si giunge a 0 o non sono finiti i nodi-pozzo (nel DAG ce ne sono sempre alcuni).

Queste operazioni si svolgono su una **copia del grafo** per ovvie ragioni. È possibile eseguire il topological sort tramite DFS, mettendo etichette numeriche a ogni nodo visitato. L'algoritmo è inoltre resistente all'input sbagliato, considerando che alla prima etichettatura con BACK segnala l'impossibilità di effettuare topological sort.

BFS

La visita filosoficamente opposta alla DFS è la **BFS, breadth-first search.** La marcatura con UNEXPLORED avviene come prima, ma i nodi visitati sono tutti quelli **a distanza uno** da quello corrente, poi a distanza due, e così via fino a distanza n. Non si implementa in modo ricorsivo: nella ricorsione la struttura di supporto è una stack., e per la breadth-first è più comoda una **queue**.

Passando per un arco, si usano DISCOVERY e BACK come nei normali grafi. Terminati gli archi adiacenti di un nodo, si testano quelli dell'altro. Il costo totale è di nuovo O(n + m) per le giuste configurazioni e rappresentazioni (non matrice delle adiacenze). Ogni volta che si trova un nuovo adiacente pozzo, lo si **mette da parte** nella queue.

Tramite **BFS** è possibile rintracciare un dato nodo t. Il percorso non sempre è unico: quello selezionato dalla visita è il primo percorso trovato che porti a t. Nel calcolo della **lunghezza minima** del cammino occorre anche considerare che un singolo arco può avere un costo di percorrenza maggiore di un gruppo di archi; solo in uno scenario in cui il costo di percorrenza è costante questa considerazione non è valida.

I grafi che riportano **valori numerici** associati agli **archi** sono detti **grafi pesati**. In alcuni grafi sono definite anche funzioni di peso sui nodi, ma il caso più interessante presenta associazioni di valori solo agli archi. Dato un grafo, trovare un cammino corrisponde alla soluzione di un problema; trovare il **cammino minimo** corrisponde alla **soluzione ottimale**.
Ogni porzione di cammino minimo è essa stessa un cammino minimo.

La sovrapposizione di **tutti** i cammini minimi costituisce un **albero dei cammini minimi**. Tuttavia, è possibile che la sovrapposizione di due cammini crei un ciclo: questo significherebbe che, per due nodi dati, almeno uno dei due ha più di un cammino minimo.
Ci sono tre varianti per l'algoritmo che trova i cammini minimi:

1) s, t $\underline{c}$ V *(shortest path)*
2) s $\underline{c}$ V, for each t $\underline{c}$ V *(single source shortest path)*
3) for each s $\underline{c}$ V, for each t $\underline{c}$ V *(all pairs shortest path)*

ALGORITMO DI DIJKSTRA

L'algoritmo più famoso è l'**algoritmo di Dijkstra**, per il single source shortest path. Si basa sul presupposto che i costi di percorrenza degli archi non siano mai negativi. Si parte con una *nuvola* che contiene solo il vertice di partenza; poi si espande visitando altri nodi e tenendo memoria della **distanza dalla sorgente**. Tale distanza viene stimata per eccesso, poiché per il calcolo sono considerati unicamente gli archi attraversati nel corso della visita. Per tutti i nodi già contenuti nella nuvola, il problema è già stato risolto. Mano a mano che il grafo viene percorso, se si trova una path più conveniente la distanza minima viene aggiornata e gli archi non più percorsi vengono eliminati.

Il processo che porta all'abbassamento della distanza minima di un dato nodo viene definito **rilassamento**. Ogni nodo viene inserito in una **coda di priorità** (heap) ordinata secondo la distanza minima di un dato nodo dall'origine, inizializzata a +inf prima di visitare il grafo. A ogni iterazione viene estratto un nodo e, per **tutti i suoi archi incidenti**, si esegue il rilassamento. Finita la visita, a ogni nodo sarà associato il valore minimo della distanza dalla sorgente.

Implementando anche un **localizzatore** che indichi la posizione del nodo nella coda di priorità è possibile anche rintracciare nodi specifici. La chiave associata al nodo deve essere anche aggiornata se necessario, ovvero quando viene aggiornata la distanza minima.

In termini di **costi**: l'inizializzazione, che realizza la costruzione dell'heap, inserisce n nodi in tempo log(n), per un costo totale di O(n*log(n)). Nella seconda parte dell'algoritmo si estrae un nodo in tempo log(n) e ne trova tutti gli archi adiacenti in tempo deg(m) = m. Stimata la distanza la si potrebbe dover aggiornare, il che avviene sempre in tempo log(n). Il tempo totale risulta quindi essere **Theta((n + m)*log(n))**. Al termine delle operazioni si aggiunge anche il genitore di ogni nodo, per ricostruire l'**albero de cammini minimi**.

In presenza di costi di percorrenza negativi l'algoritmo non aggiornerebbe appropriatamente tutti i nodi. In particolare, qualora l'**arco a peso negativo** si trovasse all'interno di un **ciclo**, l'algoritmo potrebbe percorrerlo indefinitamente per abbassare i costi totali di percorrenza dello spanning tree.

Il **minimum spanning tree** è lo spanning tree associato alla rete di cammini minimi. È possibile migliorarne la struttura analizzando arco per arco gli archi non presi, confrontandone i pesi coi pesi degli archi già presi dall'albero. Di tutti gli spanning tree ne esiste almeno uno che usa un dato arco m.

L'**albero dei cammini minimi** e il **minimum spanning tree** sono entrambi alberi ricoprenti, ma il primo è caratterizzato dalla presenza del minor numero possibile di archi, mentre il secondo ha il minor costo di percorrenza possibile.

Algoritmo di Kruskal

Uno degli algoritmi di calcolo del MST più celebri è l'**algoritmo di Kruskal**. L'algoritmo costruisce l'insieme A degli archi, che costituirà alla fine il minimum spanning tree, e **ordina gli archi** in ordine crescente di peso. Poi lavora con un partizionamento dell'insieme dei nodi in **due sottoinsiemi**, usando la *proprietà del partizionamento*. Poi controlla se, per due nodi dati, essi appartengono allo stesso insieme. L'arco selezionato viene aggiunto all'insieme A se e solo se collega **due insiemi disgiunti**; dopodiché i due insiemi si uniscono. La procedura va avanti finché non rimane un solo insieme, che è il minimum spanning tree.

Il costo dell'algoritmo di Kruskal è O(m*log(m)) = O(m*log(n)) per l'ordinamento degli archi, ma spende solo O(m*a(n)) (inversa di Ackermann). Il ciclo sui lati ha costo asintotico inferiore, per un totale di **O(m*log(n))**. Il suo antenato, l'algoritmo di Boruvka, ha costo asintotico uguale ma non ottimizzato.

Algoritmo di Prim-Jarnik

L'**algoritmo di Prim-Jarnik** è l'equivalente di Dijkstra per incontrare spanning trees. Si parte sempre da un nodo qualunque, il nodo sorgente, e si crea una **nuvola di nodi** alla quale si aggiunga via via un nodo, scegliendolo dal gruppo dei nodi ancora fuori e con la distanza minima dalla nuvola. Anche Prim-Jarnik ha la Heap e il Locator. Il **costo** è **identico a Dijksta.**

Esempio di esercizio d'esame:

Dati n dadi, ciascuno a k facce, e dato un valore z; con quante diverse combinazioni può uscire z lanciando i dadi? (Si costruisca l'algoritmo)
res(k, n) ⊆ {n, kn}

Ricorsivamente si può risolvere immaginando di chiedere all'algoritmo di calcolare le combinazioni di ogni numero da n a kn.

\\\\\\\\\\\\\\

```
conta(int n, int k, int z)
        {
        if(n == 1)
                if(z < n || z > k*n) return 0; else return 1;
        int result = 0;
        for(int i = 0, i < n, i++) result += conta(n – 1, k, z – i);
        return result;
        }
```

\\\\\\\\\\\\\\

Calcolo del costo: T(n, k, z) = {c1 if n = 1; k * T(n - 1, k, z) + c2}
*T(n) = k * T(n – 1) = k^2 * T(n – 2) = ... = k^n * T(n – i) + i * c2*
Totale O(k^n) – che diventa **O(k^2^y)** *nella dimensione dell'input!*

Esame del 12 settembre 2012.

Nel **primo esercizio** si presenta un algoritmo Java e si chiede di calcolare il **costo**. Spesso si chiede anche di definire un **metodo iterativo equivalente**. La versione iterativa richiede normalmente poche righe di codice. Nell'esempio preso in esame la dimensione dell'input è n*m, righe*colonne, quindi anche i costi vanno calcolati di conseguenza.
Quando i1 == i2 e j1 == j2, viene restituito l'elemento trovato; se i2 o j2 eccedono rispettivamente i1 o j1, viene restituito l'elemento [0][0] diminuito di 1. Il passo ricorsivo individua una sottomatrice delimitata dalla media tra i valori limite delle righe e delle colonne; dopodiché la chiamata ricorsiva successiva viene effettuata sui **quattro sottoquadranti** della matrice individuata. Questa operazione va avanti finché i quadranti non degenerano in una singola casella; quindi l'algoritmo trova il **massimo valore** contenuto nella matrice.
Equazione di ricorrenza:

T(m, n) = { c1 m*n <= 1
{ 4*T(m/2, n/2) + c2 m*n > 1
Oppure T(N) = { c1 N <= 1
{ 4*T(N/4) + c2 N > 1
T(N) = 4*T(N/4) + c2 = 4*(4*T(N/4^2) + c2) + c2 = 4^2*(4*T(N/4^3) + c2) + 4*c2 + c2 = ... = 4^k*(T(N/4^k)) + c2*(4^0 + 4^1 + ... + 4^(k – 1))
N/4^k = 1; N = 4^k; k = log_4_(N)

Quindi N*T(1) + c2*((4^(log_4_(N)) – 1)/(4 – 1)) = c1*N + c2*(N – 1)/3 = **Theta(N)**

Il costo complessivo è **lineare**.

Il **secondo esercizio** è un esercizio ricorrente nei compiti d'esame: richiede la definizione delle classi *BinTree* e *BinNode* specificandone le variabili d'istanza e la segnatura dei metodi.
Il nodo ha: int key, BinNode left, BinNode right
Il metodo **isAVL()** da implementare richiede che siano contemporaneamente verificate **isBST()** e **isBalanced()**. Si può fare entrambe le cose con il cammino Euleriano, ma è possibile risolvere il problema anche in maniera più semplice, effettuando due visite separate. Per isBST() è sufficiente una visita inorder; se restituisce valori crescenti, l'albero è un BST. *(Si può implementare come mostrato nel file .java allegato.)*

Il **terzo esercizio** è sui **grafi** e ne richiede le diverse implementazioni (a livello teorico). Poi propone il problema di restituire una lista di studenti che non hanno ricevuto il compito d'esame, che un solo studente è riuscito a svolgere. Si può risolvere il problema facendo una visita DFS (o BFS) e aggiungendo a una lista tutti i nodi UNEXPLORED, rappresentanti gli studenti a cui non è arrivato il compito. Il terzo punto richiede di adattare l'algoritmo al caso di due studenti geniali; per fare ciò è sufficiente fare due visite.

\\\\\\\\\\\\\\\\\\\\\\\\\\

```
boolean isAVL()
	{
	return isBST(root) && ; // oppure
	// return null != isBST(root, Integer.MIN_VALUE) && 0 <= isBalanced(root);
	}

Integer isBST(BinNode n, int prev)
	{
	if(n == null) return prev;
	Integer max = isBST(v.left, prev);
	if(max == null || n.key < max) return null;
	return isBST(v.right, n.key);
	}

boolean isBST(BinNode n)
	{
	if(n == null) return true;
	if((n.right == null || prev < n.right.key) || (n.left == null || prev > n.left.key)) return
false;
	return isBST(n.left) && isBST(n.right);
	}

int isBalanced(BinNode n)
	{
	if(n == null) return 0;
	int l = isBalanced(n.left);
	if(l == -1) return -1;
	int r = isBalanced(n.right);
	if(r == -1) return -1;
	if(Math.abs(l - r) > 1) return -1; // altezza negativa: non esiste
	return 1 + Math.max(l, r);
	}

boolean isBalanced(BinNode n)
	{

	}
```

\\\\\\\\\\\\\\\\\\\\\\\\\\

Modelli

LOGICA

La **logica** è la disciplina che studia la **correttezza del ragionamento**. Aristotele introdusse il concetto di **sillogismo**, una connessione di idee, un ragionamento. Il suo più famoso esempio è:

- Tutti gli uomini sono mortali
- Socrate è un uomo
- Quindi Socrate è mortale

In sostanza, da due idee di base ne deduce una terza. Le prime due sono accettate come vere nel caso preso in esame, ma a volte è meno immediato dimostrarne la consistenza; in matematica all'ipotesi segue la tesi, e il procedimento sillogistico aristotelico segue esattamente questo schema.

Uno dei primi problemi posti in questo senso fu la realizzazione di un algoritmo che risolvesse il **test di Turing**. Il crittografo aveva ideato uno schema secondo il quale un operatore umano, posto di fronte a una macchina e un essere umano, dovrebbe sempre riconoscere quale dei due è una macchina.

Non tutti i sillogismi sono validi. Ad esempio:

- Tutti gli animali sono mortali
- Socrate è un mortale
- Socrate è un animale

Parte dal presupposto (sbagliato) che Socrate appartenga all'insieme "animali", sottoinsieme proprio dei "mortali".

- Tutti gli dei sono immortali
- Gli uomini non sono dei
- Gli uomini sono mortali

Malgrado la validità della terza affermazione, il risultato non è corretto in quanto l'insieme "uomini" non è parte dell'insieme "dei" che è sottoinsieme degli "immortali", il che di per sé non esclude che anche l'insieme "uomini" sia sottoinsieme degli "immortali".

Le dimostrazioni sbagliate possono trarre in inganno. Ad esempio:

1 = sqrt(1) = sqrt((-1)*(-1)) = sqrt(-1)*sqrt(-1)= (sqrt(-1))^2 = -1

Il **teorema dell'incompletezza** di **Godel** ha dimostrato che esistono affermazioni per le quali non è possibile provare la veridicità, né fornirne una controprova.
Il **problema della terminazione** di **Turing** ha invece dimostrato che non esiste un algoritmo in grado di determinare se, a un dato input, un dato programma risulterà in un output finito (quindi terminerà l'esecuzione) oppure no.

Elementi di logica proposizionale: ^ (e), v (o), ! (non)
Ad esempio "piove^fa freddo" è formata da due *proposizioni atomiche*.

- Un insieme non vuoto, finito o numerabile (= in corrispondenza biunivoca coi reali), di simboli proposizionali (es. 'A', 'B', 'C'...)
- Le costanti proporzionali TRUE ('T') e FALSE ('T^(-1)')
- I connettivi proposizionali '^' e 'v', e i simboli separatori '(' e ')'.

Le costanti e i simboli proposizionali sono formule.

Se alpha è una formula, è una formula anche *not(*alpha*)*.
Se alpha e beta sono formule, lo sono anche alpha v beta e alpha ^ beta.

Le formule devono essere anche "ben formate": A v !B -> A v (!B).

L'operatore ! è unario, mentre v e ^ sono binari (hanno effetto su due operandi).

Not: !1 = 0; !0 = 1
And: 1^1 = 1; 1^0, 0^1, 0^0 = 0 (**commutativa** e **associativa**)
Or: 1v1, 1v0, 0v1 = 1; 0v0 = 0 (commutativa e associativa)

Un'assegnazione booleana è una funzione V: A -> {1, 0}

(PROP insieme delle formule ben formate)

Una **valutazione booleana** è invece: **V: PROP -> {1, 0}**

Ovvero è una funzione che assegna a ciascuna proposizione un simbolo di verità o falsità, a seconda dei valori che la proposizione stessa assume.

Per esempio I(V) *(interpretazione)* (T) = 1 e *I(V) (T^(-1)) = 0*.

I(V) (A) = V(A) è definita **ricorsione strutturale**.

Come esempio possiamo porre tre amici che vanno (potenzialmente) in pizzeria.

- Valutazioni: A = 0, B = 1, C = 1
- Affermazione "almeno due sono andati in pizzeria": (A^B) v (B^C) v (A^C) = 1
- Affermazione "tutti sono andati un pizzeria": A^B^C = 0

Una formula è soddisfatta per I(V(a)) = 1, è **soddisfacibile** se è soddisfatta da qualche I(V), è **tautologica** se è soddisfatta per ogni I(V), è una **contraddizione** se non è soddisfatta da alcuna I(V).

Abbiamo una tautologia a se !a è una contraddizione.

Per esempio, nel caso precedente, A^B^C è soddisfacibile, ma non una tautologia.

Implicazione logica: a -> b <=> per ogni I(V), ogni I(V(a)) = 1 produce un I(V(b)) = 1.

Due affermazioni sono *tautologicamente equivalenti* se, per ogni I(V), I(V(a)) = I(V(b)), ovvero <=> a -> b ^ b -> a.

È possibile esprimere qualsiasi proposizione logica con not e or, è un insieme minimo.

- a -> b = !a v b
- a ^ b = (((a -> nT) -> nT) -> b -> nT)

Un dato connettivo logico può essere anche definito in termini di altri connettivi. AND può essere equivalentemente espresso con una combinazione di NOT e OR. Ogni a e b si riferisce a formule ben formate. La scelta di usare AND, OR e NOT è una scelta volta alla semplicità, poiché l'insieme logico definito da tali connettori **non è minimale**.

a -> b = !a v !b (ovvero l'implicazione è falsa solo per a = 1 e b = 0)
a v b = !a -> !b (come sopra, coi simboli invertiti)
a v b = !(!a ^ !b)
a ^ b = !(!a v !b)

La dimostrazione formale dell'equivalenza delle relazioni avviene scrivendone la tabella di verità, e provando che quelle delle due funzioni coincidono.

La **precedenza** degli operatori logici è NOT, AND e OR, ->, <->.

Le **proprietà** associate alle formule logiche sono l'idempotenza, l'associatività e la commutatività. Vale anche, per AND e OR, la distributiva, per esempio:

a ^ (b v c) = (a ^ b) v (a ^ c)

E la proprietà di assorbimento:

a ^ (a v b) = a

Diverse leggi sono **duali**, ovvero restano valide anche se si invertono gli AND e gli OR, così come le leggi di De Morgan
.
Una delle più usate in ambito matematico è la contrapposizione:

a -> b = !b -> !a (dimostrazioni per assurdo)

Tuttavia gli intuizionisti negano che sia valida, considerando che la non verità dell'ipotesi non implichi la verità della tesi.

Data una funzione con n atomi, la funzione che vi associa un valore booleano rappresentante la valutazione dell'intera espressione è detta funzione booleana. Per un dato numero n di atomi, esistono 2^(2^n) casi.

Un dato insieme di connettivi logici viene definito **completo** se i connettori in esso presenti possono essere espressi anche attraverso altri connettori, sempre.
Per capire se un insieme è completo occorre definire la sua funzione. Per fare ciò è sufficiente elencare le assegnazioni per cui vale 1. La funzione viene così ridotta "ai minimi termini".

Es.: 0010 = !a ^ !b ^ c ^ !d

Esercizio: dimostrare che NAND e NOR sono completi. Usare il teorema precedente.

Esempio: tre amici vorrebbero andare in pizzeria. Se Corrado va, va anche Antonio. Se Antonio va, va anche Bruno. Quindi:

(c -> a) ^ (a -> b)

Il che NON implica logicamente a ^ b ^ c, come si può verificare nel caso in cui nessuno degli amici andasse in pizzeria. È soddisfacibile a ^ b ^ c, ma non è una tautologia, né lo è che la prima formula implichi la seconda.

Tuttavia è possibile implicare che *se Carla è andata in pizzeria anche Bruno è andato in pizzeria,* o che *se Carla è andata in pizzeria tutti gli amici sono andati in pizzeria*.
Assomiglia al sillogismo aristotelico che afferma che Socrate è un mortale (poiché è un uomo), ma quest'ultimo è più potente, perché comprende un'intera categoria anziché un solo gruppo di individui.
Non si può affermare che *se Carla è andata in pizzeria anche Bruno è andato in pizzeria* implichi a sua volta le proposizioni iniziali.

MODELLI

Un **modello** è uno strumento che permette di verificare la verità o falsità di una formula, e ne costituisce l'interpretazione.
Sia M $\subseteq$ A un insieme di simboli proposizionali; la relazione tra i due si definisce come segue:

- M |== T ogni insieme di modelli è un modello di TRUE (nessun insieme di modelli è un modello di FALSE)
- M |== a <-> a $\in$ M
- Data una formula a, M |== !a <-> M |=/= a (vale anche per AND e OR, similmente)
- M |== a -> M |=/= a oppure M |== b (vale per la doppia implicazione se a = b)

Se M |== a, a è vera in M, ossia con un'assegnazione booleana a tutti e soli i simboli presenti in M con a dà TRUE. Se ad esempio a = 0, b = 1 e c = 1, definire un modello per A significa costruire l'insieme B = {b, c}. Si dice in tal caso che M **rende vera** a.
Se M |== a per ogni M, a è tautologia.
Se M |== a per qualche M, a è soddisfacibile.
Se M |== a per nessun M, a è contraddizione.

Esercizio: provare le implicazioni logiche presenti sulle slides.

Un modello è un insieme di simboli proposizionali. Per A ^ B, il modello è M = {A, B}; per A v B i modelli possono essere {A}, {B}, oppure {A, B}. A ^ !A non ha modelli, quindi è insoddisfacibile *(contraddizione)*. A -> B ha modelli {}, {B} e {A, B}.

Esercizio: provare formalmente l'esempio 6.

Se a -> b tautologicamente, allora |== a -> b.
Se un insieme di proposizioni G implica logicamente a, tutti i modelli di G sono anche modelli di a, ovvero G |== a. Esempio: G = {g1, g2, g3} ove ciascun g ammette G come modello.

Definiamo **Mod(a)** = **{M | M |== a}**, ovvero l'insieme dei modelli di a. Per definire un modello di una formula è necessario esplicitarne anche l'alfabeto (es. ^ v ! ecc.). L'insieme di riferimento è rilevante ai fini della valutazione degli elementi in esso contenuti; l'insieme negato di {A} può essere {B}, {B, C} ecc. a seconda dell'alfabeto di riferimento. Se a -> b, Mod(g) = !Mod(a) v Mod(b).

L'**insieme delle parti** A è definito come {M | M $\subseteq$ A}, tutti i sottoinsiemi M di A. Viene indicato come P(A) oppure 2^A.

Il legame tra implicazione logica e insoddisfacibilità dimostra che:

se **G |== a** *(G modella a), è insoddisfacibile* **G u {!a}**.

Per dimostrarlo supporremo che non sia vero e che esista un modello M per !a. M |== G per definizione e dovrebbe essere valido anche M |== !a, ma tutti i modelli di M sono modelli di a, e non possono essere contemporaneamente modelli di !a. □

Possiamo anche supporre che G u {!a} sia insoddisfacibile e provarlo come segue: non esiste M | M |== G e M |== !a; quindi ogni modello può al più modellare uno dei due. Se M |== a, M |=/= !a, allora se M |== a allora M |== G e G |== a. □

Un problema famoso è costituito dalla domanda:
data una formula, possiamo stabilire se è soddisfacibile (problema della soddisfacibilità)*?*

Gli algoritmi possono anche essere divisi in **decidibili** e non decidibili, a seconda se esiste un algoritmo che risolve sempre il problema in **tempo finito**.

Nelle tabelle di verità ridotte il risultato delle formule è nelle colonne adiacenti le variabili.

Tra i problemi decidibili ci sono quelli **trattabili** e quelli **intrattabili**. Ad esempio, quello basato sulle tabelle dei numeri di simboli atomici è intrattabile, avendo costo 2^n per n simboli. Se il costo divenisse polinomiale sarebbe più accettabile, ma non ne è stata trovata la formulazione.

Esistono comunque tecniche che permettono di evitare le tabelle di verità, ad esempio la dimostrazione per casi. Supponiamo di voler provare che b sia una tautologia: è sufficiente trovare una formula tale che a -> b ^ !a -> b.

Ad esempio, poniamo che ci siano tre uomini A, B e C implicati in un furto. Sappiamo che C non lavora mai senza la complicità di A, B non sa guidare, e sono tutti fuggiti con un furgone. Almeno uno di loro è colpevole. Questo implica che Antonio sia colpevole?
L'informazione su C può essere esplicitata con C -> A.
L'informazione su B può essere esplicitata con B -> (A v C) perché non sa guidare.
Abbiamo anche posto che A v B v C.
Tecniche: *tabelle di verità, dimostrazione per casi, dimostrazione per assurdo.*

Dimostrazione per casi: se A è colpevole, banalmente A è colpevole.
Se B è colpevole, A o C sono colpevoli. Se A è colpevole, banalmente A è colpevole, mentre se C è colpevole, ciò implica che A sia colpevole.
Se C è colpevole, ciò implica che anche A sia colpevole.
In tutti i casi A è colpevole. □

Esercizio: verificare la compatibilità di tale metodo dimostrativo col concetto di dimostrazione per casi. Ricondurre le affermazioni a due casi.

HILBERT, TEOREMI

Esiste la possibilità di **dimostrare** che **a sia una tautologia** attraverso un insieme fissato di regole, senza tabelle di verità né regole prefissate. Il sistema deduttivo utilizzato a questo fine è il **sistema assiomatico di Hilbert**. Esso utilizza soltanto gli operatori ! e ->, con i quali definisce anche gli altri.
Data a e b(a/p), in cui ogni occorrenza di p è sostituita da a, poniamo b = (A ^ B) e a = C -> D e abbiamo b(a/B) = (A ^ (C -> D)). Per I(g) = I(d), I(a(g/p)) = I(a(d/p)).

Il sistema assiomatico di ***Hilbert*** prevede che:

- **a -> (b -> a);**
- **(a -> (b -> g)) -> ((a -> b) -> (a -> g));**
- **((b -> !a) -> ((b -> a) -> !b).**

Ad esempio se a -> b e b è vero, a -> b è vero. *Modus ponens (MP)*
Se b è vera e p è un simbolo vero che non compare in b, anche b(a/p) è vera. *Sostituzione uniforme (SU)*

Un **teorema** è una proposizione ottenuta a partire dalle stesse regole di inferenza. I suoi membri possono provenire dall'insieme degli assiomi AXIOMS, contenente tutte le proposizioni ottenibili dalle tre appena elencate sopra, dalla dimostrazione di un insieme di proposizioni anch'esse assiomatiche o derivate da modus ponens.

AXIOMS è l'insieme delle formule assiomatiche derivabili dai tre assiomi hilbertiani. In tale sistema si chiama teorema una relazione ottenuta a partire dagli assiomi applicando le regole di inferenza. Ovvero: |--H(a) è un **teorema del calcolo proposizionale** se esiste una sequenza a(1), a(2)... a(n) tali che:

- a(i) appartiene ad AXIOMS
- a(i) è ottenuto da a(j) e a(k) tramite *modus ponens* (a(k) = a(j) -> a(i))

Diciamo in questo caso che la sequenza a(1), a(2)... a(n) è la **dimostrazione** del teorema.

Esiste una **dimostrazione** per a(i), indicata con **G(|--H(a))** se esiste una derivazione a(1), a(2)... a(n) tale che le condizioni sopra sono verificate, e in più:

- a(i) appartiene a G
- j < k < i

Uno dei teoremi fondamentali è il ***teorema di deduzione***, che caratterizza il sistema hilbertiano: dato un insieme di formule G, siano a e g due formule proposizionali. Allora:
G, g |--H(a) se e solo se **G |--H(g) -> a**
Ovvero una formula può essere "estratta" e, tramite estrazioni successive, è possibile risalire a un teorema.

Il ***teorema di consistenza (coerenza)*** afferma che non esiste una proposizione a tale che **G(|--H(a))** e **G(|--H(!a))** siano entrambi teoremi.

Se assumiamo {a, !a} |--H(b), a e !a sono le ipotesi; possiamo scrivere allora:

- a -> (!b -> a)
- !a -> (!b -> !a)
- !b -> a (a è ipotesi)
- !b -> !a (!a è ipotesi)
- (!b -> !a) -> ((!b -> !a) -> !!b)
- !!b
- b

Questo dimostra che, prendendo come presupposto un **assurdo**, si può arrivare a qualsiasi conclusione.

L'insieme delle tautologie coincide con l'insieme dei teoremi:

|--H(a) se e solo se |==(a).

Ciò che si può derivare da un insieme di ipotesi G è valido per tutti i modelli di G:
G(|--H(a)) se e solo se G |== a.

Le due **forme normali** più usate sono la congiuntiva e la disgiuntiva. Chiamiamo **clausola** una disgiunzione di letterali (atomi), in OR.
Una formula è in **forma normale congiuntiva** se è in forma normale negativa (le negazioni sono solo davanti agli atomi) ed è della forma C(1) ^ C(2) ^ ... ^ C(n) oppure {C(1), C(2)... C(n)} ovvero C(i) è una clausola, e sono in AND.
Invertito nella definizione il ruolo ricoperto da AND e OR, la forma normale si chiama **disgiuntiva**. La clausola vuota si scrive {}.

E' sempre possibile ricondurre una formula a una forma normale.

- !(a ^ b) = !a v !b (da De Morgan)
- !(a v b) = !a ^ !b (da De Morgan)

Con De Morgan si possono semplificare anche formule molto lunghe, ma applicando la proprietà distributiva si allunga notevolmente una formula complicata.
Convert2 è più efficiente. I primi due passi sono gli stessi, ma per ogni AND vengono create nuove clausole. Ogni AND vede l'aggiunta di due clausole. Funziona così anche per OR.
Prendiamo per esempio: (a(1) ^ a(2)) v (b(1) ^ b(2)) v (c(1) ^ c(2)))
Sostituiamo a(1) ^ a(2) con d(1) e b(1) ^ b(2) con d(2), e così via.
a = Convert1(a) sfrutta l'equivalenza, mentre Convert2(a) sfrutta la soddisfacibilità e crea una nuova formula che ne preservi la veridicità. (Convert1 si vede più spesso agli esami.)

ALGORITMO DPLL

Data una formula in forma normale congiuntiva (tutti AND), l'**idea di DPLL** è un algoritmo di ricorsione binaria che tenta di assegnare tutti i valori proposizionali a una proposizione. Ogni atomo isolato viene tradotto col valore che gli si deve affidare *(unit propagation)*, oppure i letterali sempre veri e atomici sono sempre presi come tali *(pure literal elimination)*. Tutte le proposizioni atomiche di quel tipo devono essere o affermate o negate, per esempoi non ci sono a e !a insieme in tal caso.

Le clausole che hanno alcune parti vuote, {}, sono sempre false. La chiamata dell'ipotetico metodo ricorsivo avviene scegliendo un letterale. Il risultato è la presenza del valore nell'insieme: a e !a come due separate chiamate ricorsive (!a si esegue solo quando necessario, per via della *lazy evaluation*).
Se si vuole sapere se una data formula a è una **tautologia** o meno, e in una formula è presente solo !a, basta negare la fomula e verificare che sia soddisfacibile.

(c v !a v b) = (!c ^ a ^ !b)

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

```
public static boolean uscitaRaggiungibileDa(int r, int c)
        {
        if((lab[r + 1][c].crossed || !(percorribile(r + 1, c))) && (lab[r][c + 1].crossed || !
(percorribile(r, c + 1))) && (lab[r - 1][c].crossed || !(percorribile(r - 1, c))) && (lab[r][c -
1].crossed || !(percorribile(r, c - 1))))
                return false;
        if(uscita(r, c) && percorribile(r, c))
                {
                lab[r][c].crossed = true;
                return true;
                }
        if((percorribile(r + 1, c) && uscitaRaggiungibileDa(r + 1, c)) || (percorribile(r, c + 1)
&& uscitaRaggiungibileDa(r, c + 1)) || (percorribile(r - 1, c) && uscitaRaggiungibileDa(r - 1,
c)) || (percorribile(r, c - 1) && uscitaRaggiungibileDa(r, c - 1)))
                {
                lab[r][c].crossed = true;
                return true;
                }
        }
```

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

Il concetto di **linguaggio** ha come caso particolare i linguaggi di programmazione, così come lo sono i linguaggi di scripting, di specifica, di configurazione, di marcatura (es. HTML e XML), i compilatori e i protocolli. Ciascuno permette di dire che alcuni problemi sono risolvibili e altri no, e tra i risolvibili alcuni sono semplici e altri complessi.

Un **alfabeto** è un insieme finito non vuoto di simboli (caratteri). Per esempio:
Alfabeto binario = {0, 1}
Alfabeto decimale = {0, 1, ..., 9}
Dato un alfabeto Sigma, si dice **stringa** o **parola** una sequenza finita di caratteri appartenenti a Sigma. Data una stringa x, la sua lunghezza si indica con |x|. La stringa di lunghezza zero (eps) è detta vuota o nulla. L'insieme di tutte le stringhe componibili con un dato alfabeto Sigma è **Sigma*** (Sigma-star).

La **concatenazione** è la giustapposizione di due stringhe. È associativa ma non commutativa:

salva°gente = salvagente != gente°salva

Per indicare la ripetizione di simboli all'interno di una stringa si usa il simbolo di **potenza**:

w^3 = www
x^2 (con x = cous) = couscous

La convenzione è che le ultime lettere dell'alfabeto indichino stringhe.

Dato un alfabeto, la terna **<Sigma*, °, eps>** è un **monoide**, un insieme chiuso rispetto alla concatenazione. È chiamato monoide sintattico ed è alla base della definizione dei linguaggi.

Un linguaggio è un *sottoinsieme* di Sigma*. Esistono quindi infiniti linguaggi.

Ad esempio, dato {0, 1}, può essere un linguaggio l'insieme di stringhe che contengono almeno un numero pari a 1: {1, 10, 11...}

Il linguaggio {eps} non è considerato un **linguaggio vuoto**. {} lo è.

Sigma e Sigma* sono linguaggi. {} = Lambda.

Non tutti i linguaggi hanno significato; a noi interessano solo quelli che hanno un preciso e definibile criterio di appartenenza. Per esempio il linguaggio costituito da parentesi bilanciate (una aperta e una chiusa, inizia con l'apri parentesi e chiude col chiudi parentesi): (()()))()

Un **linguaggio di programmazione** può essere definito come il linguaggio le cui *stringhe* rappresentano *programmi* sintatticamente corretti. Il compilatore, quindi, risolve un problema di appartenenza della stringa a un linguaggio.

Si può definire un linguaggio tramite:

- *Approccio algebrico*, mostrando come il linguaggio è costruito a partire dagli altri;
- *Approccio generativo*, definendo una grammatica da rispettare per le stringhe dell'alfabeto;
- *Approccio riconoscitivo*, che data una stringa in ingresso determina se appartiene o meno al linguaggio.

Non tutti possono essere definite mediante grammatiche o riconosciute da automi. Nel secondo caso è possibile provare formalmente che l'alfabeto non ammette soluzione di tipo riconoscitivo.

L'insieme delle parti dei naturali ha la stessa **cardinalità** dei reali. Esistono più linguaggi che algoritmi. Le operazioni tra linguaggi sono come le operazioni tra insiemi. La **complementazione** avviene invece così: L = {x app. Sigma* | x non app. L}

Dati due linguaggi L1 e L2, definiamo come segue il prodotto di linguaggi:

L1°L2 = {x app. Sigma* | esistono x1 app. L1 e x2 app. L2 | x1 + x2 = stringa).

Come quella tra stringhe, è associativa ma non commutativa.

Esempio:

L1 = {0, 1} = L2; L1°L2 = {00, 01, 10, 11}
L°L = L^2

L'**iterazione** di un linguaggio L* è l'unione di tutti i suoi possibili Li a partire da i = 0. L+ parte da i = 1. Per esempio dato {a, b} L* contiene tutte le stringhe costituite da coppie di a e di b, senza la stringa vuota.

Nell'**approccio generativo** alla descrizione dei linguaggi si mira alla **riscrittura**. L'approccio di Chomsky portò alla **grammatica formale G = <Vt, Vn, P, S>** in cui Vt è un alfabeto di simboli terminali (quelli dell'alfabeto del linguaggio), Vn è un alfabeto di simboli non terminali, P è l'insieme di produzioni (sequenze di terminali e non terminali, ma anche vuota) e S è l'assioma.

Ciascuna regola di produzione è una regola di riscrittura. Ogni regola a -> b indica che la stringa a **può essere rimpiazzata da b.** In tal modo si costruisce una nuova stringa. Il linguaggio generato da una grammatica è l'insieme di stringhe ottenibili con riscritture finite. Ciascuna di esse è l'applicazione di una regola di produzione.
Esempio: G = <{a, b}, {S}, P, S> ove P è l'insieme delle produzioni:

S -> aSb
S -> ab

In forma compatta: S -> aSb | ab
V indica Vt + Vn.

Data una grammatica **G = <Vt, Vn, P, S>** una stringa b si ottiene da una stringa a per **derivazione diretta** solo se esistono le stringhe c e d (anche vuote) tali che a = cxd, b = cyd. In tal caso si scrive a -> b (se P comprende la produzione x -> y).
La **derivazione** semplice è l'estensione della derivazione diretta, data per una sequenza di stringhe ognuna delle quali ottenuta dalla precedente attraverso derivazione diretta. In tal caso a -> *b. Per convenzione, a -> *a, anche se tale * è nullo.

Data una grammatica **G = <{a, b, c}, {S, B, C}, P, S>** ove P ha le seguenti produzioni:
S -> aS | B
B -> bB | bC
C -> cC | c
Operazione lecita: aaaB -> aaabC -> aaabcC

Una forma di frase è una qualunque stringa x t. c. x app. V* e S -> *x. Si definisce **linguaggio generato** l'insieme di soli terminali derivabili dall'assioma:
L(G) = {x | x app. Vt* ^ S -> *x}

Esempio di linguaggio generato:

G = <{a, b}, {S, A}, P, S> con S -> aB | b e B -> aS produce L = {a^n b, n >= 0}.

GRAMMATICHE

Ogni linguaggio è un sottoinsieme di Sigma*. Il linguaggio generato da una grammatica è l'insieme di stringhe derivabili dall'assioma: S =>* x | x c Vt*.
Due grammatiche si dicono **equivalenti** se generano gli stessi linguaggi. Alcune sono più efficaci di altre nella descrizione di linguaggi specifici; il problema fondamentale di ogni linguaggio è il problema del riconoscimento, ossia il determinare se una data stringa appartiene o no al linguaggio.

È possibile **passare** da una data grammatica a una **grammatica equivalente**: hanno lo stesso alfabeto, sia di terminali che non, e lo stesso assioma, ma le loro produzioni sono diverse.
G1 e G2 = <{a, b}, {S, A}, P, S> con:

S -> Ab | b	S -> Ab
A -> aAa | aa	A -> Aaa | *epsilon*

L'epsilon-produzione non sempre è una caratteristica positiva. Spesso si passa a una grammatica equivalente per evitare di gestire il caso epsilon.

Esempio di generazione del linguaggio {a^n*b^n*c^n | n >= 1}
Dalla grammatica G = <{a, b, c}, {S, B, C}, P, S>
Produzioni:

- S -> aSBC
- S -> aBC
- CB -> BC
- aB -> ab
- bB -> bb
- bC -> bc
- cC -> cc

S -> aSc | B
B -> bB | b

Ad esempio la forma di frase S -> aSBC -> aaSBCBC -> aaaBCBCBC (ottenuta applicando le prime due regole) viene ordinata con la terza regola. È opportuno ricordare che un'applicazione prematura delle regole dalla quarta in poi possono rendere impossibile la composizione di una stringa **priva di non terminali**.

Per arrivare a una stessa stringa si può anche passare attraverso due diverse sequenze di produzioni. Ciò pone un problema: il compilatore di un programma deve ricostruire la sequenza di produzione di un programma, il che può essere più complicato in caso di mancata univocità.

Il **problema del parsing** (analisi sintattica), in cui due sequenze generano la stessa stringa, causa ambiguità di interpretazione dei programmi.
Ad esempio, se in un programma gli if non fossero indentati, ammettendo che il compilatore non associ il primo else all'ultimo if, si presenterebbe un problema di parsing.

GRAMMATICHE CHOMSKY

Chomsky ha classificato le grammatiche in:

- *tipo 0:* produzioni non limitate
- *tipo 1:* produzioni context-sensitive (contestuali)
- *tipo 2:* produzioni context-free
- *tipo 3:* produzioni regolari

Aumentando i livelli aumentano anche le restrizioni.

Il **tipo 0** non presenta vincoli:
alpha -> beta, alpha c V* ° Vn ° V*, beta c V*
Durante la riscrittura, la forma di frase si compatta; avere **produzioni che si accorciano** non è permesso nelle grammatiche di altro tipo.
Il **tipo 1 (context-sensitive)** ha la seguente regola:
alpha -> beta, alpha c V* ° Vn ° V*, beta c V+, con |alpha| <= |beta|
La parte destra non può essere più corta della parte sinistra. La lunghezza delle forme di frase resta invariata.
a^n*b^n*c^n è un linguaggio di tipo 1: b e c non hanno esponenti inferiori ad a. Vedremo che è *strettamente di tipo 1*, perché non è generabile con una grammatica di tipo 2.

Il **tipo 2 (context-free)** ha la seguente regola:
A -> beta, A c Vn, beta c V+
Ovvero a sinistra può esistere solo un non terminale.

Il **tipo 3** ha la seguente regola:
A -> aB | a con A, B c Vn, a c Vt
Impone condizioni anche sull'output delle produzioni e sulla formattazione dello stesso.

Ogni linguaggio è incluso in Sigma*. L'insieme di *tutti i possibili linguaggi* è **2^(Sigma*)**, che NON coincide con l'insieme dei linguaggi di tipo 0. I linguaggi al di fuori dell'insieme dei linguaggi di tipo 0 non sono descrivibili tramite grammatiche di Chomsky.

Un esempio di linguaggio di tipo 2 è il linguaggio con le **espressioni**.

- E -> E+T | T
- T -> T*F | F
- F -> a | (E)

Esempio: E -> E+T -> E+T*F -> T+T*F -> F+T*F -> F+F*F -> a+a*a

Lo si può graficare anche con un albero, aggiungendo rami anche per i simboli delle operazioni aritmetiche da svolgere. La **struttura ad albero** aiuta a preservare la corretta **precedenza** tra le operazioni.

Alternativa all'impostazione sopra:

E -> (E) | E+E | E*E | a

Derivando però (a+a)*a ci si rende conto che la grammatica è ambigua: ci sono due interpretazioni possibili.

Data una stringa w su un alfabeto Sigma, la **stringa simmetrica w(r)** o **w~** è tale che:

- se w = epsilon, w~ = epsilon
- se w = sigma*x, w~ = x~*sigma
- w = (w~)~

Il linguaggio delle stringhe palindrome si definisce come L = {x ⊆ Sigma+ | x = w*w~ | x = w*sigma*w~, w ⊆ Sigma+, sigma c T}.

Esercizio: provare a scriverne il linguaggio. Scrivere anche una serie di regole per le stringhe palindrome.

L'attuale formulazione dei linguaggi di tipo 1 non è interamente fedele a quella proposta al tempo da Chomsky; lui descrisse le produzioni di tipo 1 in questo modo:
alpha -> gamma con alpha -> phi1*A*phi2 e gamma = phi1*beta*phi2, A ⊆ Vn
Tuttavia è applicabile solo se alpha è effettivamente preceduto da qualcosa. **Quindi la definizione di Chomsky è inclusa nella definizione attuale**.

Notiamo come è possibile trasformare BC -> CB in grammatica di Chomsky:
- CB -> CX
- CX -> BX
- BX -> BC

Le **epsilon-produzioni** sono produzioni che generano una stringa vuota. Non sono ammesse nei linguaggi più specifici del linguaggio 0, quindi ogni linguaggio che la include è un linguaggio di tipo 0. Se un linguaggio contiene la parola vuota, deve esistere anche la rispettiva epsilon-produzione.
Ammettiamo anche epsilon-produzioni in linguaggi diversi dallo 0, ma di fatto così facendo non aderiamo appieno alle grammatiche di Chomsky.

Supponiamo di avere un linguaggio L che non contiene la **stringa vuota**. Ora vogliamo ottenere un linguaggio L' | L' = {L ∪ epsilon}. Oppure: G' | L(G') = L(G) ∪ {epsilon}
G = <Vt, Vn, P, S> e G' = <Vt, Vn ∪ {S'}, P', S'}
P' = P ∪ {S' -> S | epsilon}
Si rende necessario un **nuovo assioma** perché qualora comparisse in una delle produzioni, si rischierebbe di modificare il linguaggio mantenendo le produzioni originarie.

Se la epsilon-produzione compare in una posizione generale, trasforma grammatiche di tipo 1 in grammatiche di tipo 0 e non amplia le produzioni di grammatiche di tipo 2 e 3.
Per esempio per a -> b, se |a| > |b|, avviene una violazione delle regole del linguaggio (stringa destra più corta).

L'epsilon-produzione è ammessa in una grammatica di tipo 1 purché compaia alla destra di un letterale **non terminale** che non compare a destra. In tal caso, per ogni non terminale che non compare a destra, la produzione si cancella e si trova una nuova produzione sostituendo i precedenti non-terminali. Non si può fare in presenza di **produzioni unitarie**, ossia non terminali che producono altri non terminali.
I simboli *nullable*, che si annullano, derivano da altri non terminali annullabili, ad esempio nei cicli di produzione.

A -> x1*x2*x3*x4, x1 e x2 si **annullano**:

- A = x1*x3
- A = x1*x2*x3
- A = x1*x3*x4
- ! = 1 x1*x2*x3*xd4

0 Contiene le eps.-prod.
1 Sì eps-prod. se l'assioma non compare a destra
2 Sì eps-prod.
3 Sì eps.-prod.

Il **problema del riconoscimento** è il problema decisionale dei linguaggi: decidere se una stringa x appartiene a un linguaggio o meno. Una delle macchine classiche è una macchina astratta, la *macchina di Turing*, il cui algoritmo risolve il problema. Alcuni linguaggi di tipo 0, però, lasciano la macchina "perplessa" sull'attribuzione. Il **riconoscimento** di una stringa appartenente al linguaggio è garantito, ma quello del non riconoscimento è un problema diverso; per una stringa che non appartiene effettivamente al linguaggio, il computer può bloccarsi o dare la stringa per buona. Questo genera problemi semidecidibili: a volte il programma va in loop cercando una soluzione.
Riconoscere = decidere; accettare = semidecidere

I **problemi decidibili** sono quelli proveninenti da linguaggi decidibili. Similmente, i linguaggi **semidecidibili** generano problemi semidecidibili. Per provare che un linguaggio è decidibile, occorre provarne la semidecidibilità e la semidecidibilità del suo complementare.

I **linguaggi regolari** (di tipo 3) hanno propri algoritmi di riconoscimento, eseguiti dagli automi a stati finiti. L'esame avviene in maniera sequenziale; ad ogni passo la "testina" passa avanti e modifica il suo stato, registrandone l'avanzamento.
L'input è un nastro unidirezionale di controllo. Arrivato all'ultimo carattere, si trova in stato di accettazione o in stato di rifiuto.

Definizione di **automa a stati finiti**: **A = <Sigma, K, delta, q0, F>**
Sigma è l'alfabeto di input, K è l'insieme di stati (alcuni di essi sono finali, quindi F c K e q0 $\underline{e}$ K è lo stato iniziale.
Guardando lo stato corrente delta e l'alfabeto Sigma, l'automa determina lo stato successivo.

Funzione estesa alle stringhe:

delta(q, eps) = q
delta(q, ax) = lambda((lambda(q, a),x))

Nel secondo caso si allega come parametro una stringa intera.

Il **linguaggio riconosciuto** è L(A) = {x $\underline{c}$ Sigma* | delta(q0, x) $\underline{c}$ F} composto dalle stringhe che fanno cambiare lo stato dell'automa e lo portano allo stato finale.

La **funzione di transizione** può essere rappresentata tramite matrice di transizione o diagramma degli stati. La matrice è facilmente implementabile algoritmicamente. Il diagramma è un grafo (albero generico). L'ingresso è indicato da una freccia entrante generica.

Ad esempio, possiamo costruire un automa che riconosca il linguaggio delle parole che contengono un numero pari di a o di b.
Sigma = {a, b}
K = {q0, q1, q2, q3}
F = {q0, q1, q2}
Lo stato iniziale può essere anche uno **stato finale**. Possono esistere più stati finali.

Un altro automa può riconoscere gli **identificatori**, ad esempio i nomi assegnati alle variabili o agli oggetti, oppure le parole riservate di un linguaggio di programmazione. Un esempio può essere dato dal fatto che il nome di una classe Java deve iniziare con una maiuscola. Tale parte preliminare dell'analisi sintattica *(parsing)* si chiama **analisi lessicale**.

Esistono anche gli **automi a stati finiti non deterministici**. Si possono simulare con macchine multicore. La filosofia operativa consiste nel propagarsi ogni volta che ci si trova di fronte a un bivio, ossia scegliere tutte le alternative possibili. Tuttavia, per quanto grande, la memoria è finita, e non è possibile proseguire la moltiplicazione dei thread all'infinito. Il non determinismo è essenzialmente uno strumento teorico che permette di analizzare i limiti dell'informatica.
La differenza con l'automa deterministico è nella **funzione di transizione**:
delta(n) : K * Sigma -> P(K) va nell'insieme delle parti di K, ossia tutti i possibili sottoinsiemi dello stato (tutti i possibili stati). Anche l'insieme vuoto è parte di P(K).

La funzione di transizione **estesa alle stringhe** è:
delta(n) : K * Sigma* -> P(K)
L'insieme di stati in cui si arriva attraverso la stringa di Sigma*.
delta(n)(q, eps) = {q}
delta(n)(q, ax) = $\underline{U}$ delta(n)(p, x)
Essendo una funzione non deterministica, va in un insieme di stati.

Per esempio: delta(n)(q0, a) = {q0}
(q0, b) = (q0, q1); (q1, a) = (q0, q1); (q1, b) = (q1)
Applicando delta(n) a (q0, a) il risultato è q0, quindi b varia solo all'interno di q0.
delta(n)(q0, bb) = delta(n)(q0, b) $\underline{u}$ delta(n)(q1, b)

Si definisce **linguaggio riconosciuto da un automa non deterministico** il linguaggio che ha l'insieme di stati di arrivo con un'intersezione non vuota con gli stati finali.

L'introduzione del non determinismo non implica un aumento nella potenza di ricognizione del linguaggio; ogni macchina deterministica può individuare gli **stessi linguaggi** di una macchina non deterministica. Le macchine non deterministiche sono solo più veloci.
Nelle transizioni non deterministiche è possibile che, da uno stesso stato e con uno **stesso input**, si giunga a **due stati diversi**. Ad esempio: delta(q0, a) -> q0 e delta(q0, a) -> q1.

delta(n) : K x Sigma -> K (caso deterministico)
delta(n) : K x Sigma -> 2^K (caso non deterministico)

Affinché un automa sia **deterministico** non è sufficiente che gli archi uscenti non si "dividano" come nell'esempio precedente: è necessario anche che **non manchi nessun arco**, ovvero che ogni carattere nell'alfabeto d'ingresso sia presente sugli archi.

Una **computazione** è un calcolo eseguito da un automa. Se lo stato in cui l'automa si trova dopo la computazione è uno stato finale, la stringa viene accettata; altrimenti, viene rifiutata. L'input viene scritto su una parte di nastro; la porzione non occupata da input è *blank*, indicata con *b* o □. Immaginando di "congelare" la lettura mentre il puntatore sta per leggere la stringa x, la **transizione di stato** viene indicata con "|-":
<q, x> |- <q', x'> <=> x = ax' ^ delta(q, a) = q'
Ovvero possiamo scrivere x come stringa unica dell'alfabeto e, dati q e a (carattere letto), lo stato successivo della stringa è dato da q'.

Una **configurazione** (coppia <q, x> con q $\underline{c}$ K) è *iniziale* se q = q0, *finale* se x = epsilon, *accettante* se x = epsilon ^ q $\underline{c}$ F.

Due configurazioni sono in relazione se e solo se:
<Q, x> |- <Q', x'> <=> x = ax' ^ $\underline{U}$(q $\underline{c}$ Q) delta(n)(q, a) = Q'
Quindi la computazione è una **sequenza di configurazioni** c(0), c(1), ..., c(i) tale che c(i) |- c(i+1).

La **chiusura transitiva** R+ è una relazione binaria che indica la presenza di una relazione binaria tra tutti gli elementi successivi x(i) e x(i+1) di una sequenza, tali che x(0) = x e l'ultimo membro x(n) = y (oppure, banalmente, x = y). Se avviene che gli elementi successivi sono uguali tra loro, la chiusura diviene **transitiva e riflessiva**.
Per esempio: A = N (insieme dei numeri naturali); R = succ.:
a [succ.] b -> a = b || a+1 = b
In grafi O->O->O->O->...->O
Allora *R corrisponde alla relazione* <=.

La chiusura transitiva e riflessiva di una sequenza di configurazioni (computazione) si indica come: **c(k) |-* c(k+i)**.

Gli stati non deterministici non sono ben definiti; sono sequenze di insiemi di stati. L'idea di transizione di stato è spesso rappresentata con un albero (generalizzazione del grafo). Si chiama **grado di non determinismo** il numero massimo di stati successivi. Esso caratterizza la macchina in base a un particolare input, dato che in base all'input può variare. Una definizione alternativa è che ciascuna computazione sia un percorso a parte e le macchine non deterministiche siano le uniche ad avere molte computazioni possibili.

Le **espressioni regolari** è una stringa r sull'alfabeto (Sigma $\cup$ {+, *, (,), ., vuoto). Un'espressione regolare è: r = vuoto, r = a per ogni a $\subseteq$ Sigma, r = (s+t) o r = st. Nella moltiplicazione il simbolo si sottintende.
Esiste per definizione un linguaggio associato a un'espressione regolare: essi sono linguaggi di tipo regolare. Se ,l'espressione è 0, il linguaggio è Lambda; di a è {a}; di s+t, st e s*.

(a+b)* -> (L(a+b))* -> (L(a) $\cup$ L(b))*

Per esempio se r = (((aa)+b)c*) si può scrivere come L(a) $\cup$ L(b) + L(c)* (numero qualsiasi di c).

LINGUAGGI

I linguaggi rappresentabili dalle espressioni regolari sono **linguaggi regolari**. Negli editor di testo sono identificati, solitamente, come *grep* o *regex*. L'espressione regolare a+ corrisponde ad aa* oppure a*a. Alcuni simboli:
^ esclude un'espressione dalla riga, ma a inizio riga impone che l'espressione cominci da lì
$ impone che l'espressione termini a fine riga
[] indicano che i simboli in essa contenuti possono comparire (solo un'istanza per volta)

L(ASFND) = L(ASFD), ovvero la **potenza di ricognizione** di linguaggi degli automi a stati finiti non deterministici è uguale a quella degli automi a stati finiti deterministici.

Dimostrazione:

1) L(ASFD) $\subseteq$ L(ASFND)

Intuitivamente vera considerando che gli ASFD sono lineari.

1) L(ASFND) $\subseteq$ L(ASFD)

Assegnato l'ASFND, costruiamo A' = <Sigma', K', delta', q0', F'> deterministico, ove l'alfabeto Sigma è lo stesso, l'insieme degli stati K' è l'insieme delle parti di 2^K (perché è lineare, quindi alcuni stati si "ripetono") e si mantengono la funzione di transizione e l'ingresso dato.

Per **K = {q0, q1, q2}**, F = {[], [q0], [q1], [q0, q1], [q0, q2], [q1, q2], [q0, q1, q2]} (automa a stati finiti non deterministico).
Supponiamo che lo stato finale sia q2. Allora **tutti gli stati** che comprendono q2 sono **stati finali**.
Rimane da definire la funzione di transizione delta'. Ad esempio: delta'([q0, q1], a) = ?
Lo stato successivo è un insieme di stati; allora è possibile fare: *delta(q0, a)* $\cup$ *delta(q1, a)*.
Dopodiché è vero per costruzione che il risultato sia lo stesso. □

In questo automa a stati finiti non deterministico esiste uno **"stato pozzo"**, B, nel quale conduce l'input 0. Descriverne gli stati tramite una tabella richiederebbe un **costo esponenziale**, pari a 2^n, poiché secondo quanto visto prima K' = 2^K.
Tipici esercizi d'esame sono ad esempio:
Dato (0* + 11)* rappresentarne l'ASFD (solitamente si passa per l'ASFND prima di trovarne la realizzazione).

Classi di **linguaggi equivalenti**: **L(ER)** espressioni regolari, **L(ASF)** definiti da automi a stati finiti, **L(GR)** grammatiche regolari.
Dimostrazione:

1) L(ER) $\subseteq$ L(ASF)

Supponiamo dato Sigma. Dimostriamo che i linguaggi delle ER a, b, Lambda ecc. Sono riconoscibili con degli ASFND. I grafi relativi sono semplici. Ora mostriamo che un ASFND riconosce anche l'unione di linguaggi: dati A1 e A2, l'automa che **riconosce L(A1) $\cup$ L(A2)** è l'automa A in cui Sigma = Sigma1 $\cup$ Sigma2, K = K1 + K2 + {q0} nuovo stato iniziale, F = F1 $\cup$ F2.

Nel processo di fusione di due automi viene preso un **nuovo stato iniziale**. Di conseguenza, gli stati iniziali dei vecchi automi si **perdono**, poiché il nuovo stato iniziale si occupa di tutte le transizioni a loro precedentemente assegnate.

1) Esiste un ASFND che **riconosce L(A1) ° L(A2)**

Dati A1 e A2, che riconoscono rispettivamente L(A1) e L(A2), l'automa che riconosce L(A1) ° L(A2) è un automa le cui transizioni avvengono come in A1 o A2 qualora il rispettivo stato appartenga a uno dei due, ove lo stato finale dell'automa A1 è lo stato iniziale di A2.

1) Dato un ASFND A, esiste un ASFND che **riconosce L(A)***.

L'automa che riconosce l'iterazione del linguaggio ha le stesse transizioni delta' delle delta viste nel primo automa.

L(ASF) $\subseteq$ L(GR)

Dove Vn = {A(i) | per ogni q(i) $\subseteq$ K} dove per ogni transizione delta(sigma, q(i)) -> q(j) costruiamo la produzione A(i) -> sigmaA(j). Può essere reso anche con epsilon-produzioni, nonostante non convenga.
Se q0 appartiene agli stati finali F, creiamo un nuovo non-terminale A0' che per ogni **A0 -> aA(i)** comporta **A0' -> aA(i)** e **A0' -> epsilon**. Questa è una formulazione equivalente, valida se e solo se il linguaggio contiene anche la stringa vuota.

Per verificare questa costruzione occorre verificare che la derivazione A(i) -> *xA(j) se e solo se **delta(q(i), x) = q(j)** e le rispettive produzioni finali portano a **stati finali**.
Per esempio, se |x| = 1, poniamo x = a ed è evidente che delta(q(i), a) -> q(j) e per costruzione A(i) -> aA(j) mentre se q(j) c F allora A(i) -> a.

ASF = ASFND; ER -> ASFND -> GR. Provare che GR -> ER chiude il circolo delle espressioni equivalenti e proverebbe anche che GR -> ASFND.
Un accenno di dimostrazione: si trasforma la serie di produzioni in un sistema di equazioni. Per esempio se abbiamo **A -> a1B1 | a2B2 | ...** diventa **A = a1B1 + a2B2 + ...**
I metodi di risoluzione sono due. Uno è ovviamente quello della soluzione, che però funziona solo se i non-terminali non sono ricorsivi (cioè se non producono anche se stessi). In tal caso bisogna prima **eliminare le ricorsioni**. Per esempio: A -> aA | b diventa A = a*b. In generale:

A = alpha1A + alpha1B ... -> (alpha1 + alpha2 + ...) + (beta1 + beta2 + ...)

A = aA + bB = aA + bb*c
B = bB + c = b*c
Quindi A = a*b*c

Facendo il processo inverso si potrebbe giungere a un'altra delle infinite grammatiche che rappresentano tale linguuaggio.

PUMPING LEMMA

Nei casi di linguaggi regolari possono esistere un numero finito o **infinito di parole**. I casi finiti sono semplici; quelli infiniti richiedono cicli. Il fatto che un automa abbia i cicli NON implica che abbia anche stati finali; se per un dato input riconosce un output via vicli, ne riconoscerà anche n ireraioni dello stesso ciclo. Si chiama **pumping lemma:** per ogni L esiste una costante n tale che z c L e |z| >= n allora z = uvw tali che |uw| <= n e k c N; l'importanza del pumping lemma risiede nel provare che il dato lunguaggio NON appartiene a una data categoria di linguaggi.

Non si possono definire regolari i linguaggi che hanno un **numero arbitrario di non-terminali** (es. a^n b^n). Le n non sono definite, quindi non possono essere riconosciute da automi a stati finiti, né deterministici né non deterministici.

Per esempio le palindrome non sono regolari:
w -> w' | w c (a + b)*
Non si può scomporre w in u, v, w. Non sono regolari nemmeno le "stringhe doppie" ww.

Il pumping lemma porta alle seguenti conclusioni:

- Se un linguaggio è riconosciuto a un dato stato n, esiste x c L di lunghezza |x| < n.
- Se un linguaggio è riconosciuto a un dato stato n, esiste n < |x| < 2n.

Quindi è possibile dire se un linguaggio è **vuoto, finito, infinito.**

La classe dei linguaggi regolari è **chiusa** rispetto alla **complementazione.** Il complemento di un linguaggio regolare è regolare. È possibile provarlo invertendo stati finali e non finali, ma funziona solo con gli automi deterministici: quelli non deterministici possono avere due stati di risultato diverso dato lo stesso input.
Differenza simmetrica tra insiemi: Delta(L(A1), L(A2))
Equivale all'unione meno l'intersezione degli insiemi. Per provare che due linguaggi o automi sono equipotenti (possono rappresentare lo stesso linguaggio, nel caso di automi) la loro differenza simmetrica deve essere l'**insieme vuoto**.

Tipico esercizio d'esame:
(00* + 10)(01*)* -> Automa a stati finiti

LINGUAGGI CF

I **linguaggi context-free** (o *non contestuali*) sono generati da grammatiche di tipo 2. Molti dei problemi decidibili per linguaggi regolari non sono decidibili per linguaggi context-free. Sono riconoscibili da automi non deterministici, con strutture di memoria a **pila**. Una particolare categoria dei linguaggi non contestuali è quella a cui appartengono i **linguaggi di programmazione**, con alcune riserve: ad esempio dichiarare due volte la stessa variabile non è possibile, e ciò indica una certa sensibilità al contesto.
Un altro esempio di linguaggio context-free è il **linguaggio umano**.

L'esigenza di creare un compilatore nacque dal fatto che i primi computer erano programmati in esadecimale. Il primo compilatore fu scritto in Assembler.
Uno dei programmi più famosi che effettuavano il test di Turing, Eliza, simulava uno psicologo.

L'analisi parte dall'**analisi lessicale**, che consiste nella scomposizione di codice sorgente in *token*. I token possono essere identificatori o parole riservate di un linguaggio (for, while, class, boolean, volatile, transient).

La fase successiva è l'**analisi sintattica** o **parsing**. Esso determina la struttura del programma: segnala errori lessicali e costruisce l'**albero di derivazione** della stringa, ossia ricostruisce la sequenza di produzioni che genera la stringa (il programma) in input.

Le **espressioni matematiche** possono essere assunte come linguaggi context-free, a meno di espressioni che possono generare ambiguità (alcuni rami possono essere attraversati indifferentemente prima o dopo di altri).
L'ambiguità matematica delle prime calcolatrici fu risolta dalla HP con la notazione inversa polacca, di fatto più complicata, ma anche più efficiente.

Ai fini del parsing sono più interessanti le **grammatiche non ambigue**, ossia quelle che hanno un solo albero di derivazione. Questo fa porre il problema: dato un linguaggio, esiste una grammatica non ambigua che lo genera? È un problema tuttora irrisolto. Alcuni linguaggi non hanno grammatiche non ambigue che li generano, e per questo vengono detti *inerentemente ambigui*.

Il parsing è un'operazione diffusissima: la svolgono, ad esempio, i compilatori e i **browser web** per i file .html delle pagine visitate. Il parsing web prevede di solito di ignorare le porzioni di codice non interpretabili o contenenti errori.

Esiste un **pumping lemma** anche per i linguaggi context-free, che permette di stabilire alcune proprietà decidibili di tali linguaggi:
Dato L linguaggio context-free infinito, esiste n(L) > 0 tale che per ogni s ⊆ L con |s| > n(L), s = yuvwz per cui:

- **|uw| > 0** *(almeno una delle due non nulla)*
- ***|uvw| < n(L)***
- *per ogni i >= 0,* **y*u^i*v*w^i*z**

Ovvero, la stringa viene suddivisa in 5 sottostringhe (eventualmente alcune vuote). Poi viene effettuato il pompaggio (pumping) sulla seconda e la quarta.

Applicando nel particolare il pumping lemma al linguaggio:
{ S -> E
{ E -> E + E | E * E | id
Si ottiene che *u e w sono pompabili, ma lo è anche v.* È legittimo quindi supporre che sia un linguaggio regolare; per verificare che effettivamente lo sia, occorre applicare il pumping lemma per i linguaggi regolari. Tuttavia, il **pumping lemma** costituisce una **condizione necessaria e non sufficiente** per l'appartenenza a un dato tipo di linguaggio, perciò anche nel caso di corrispondenza col lemma di tipo 3 non si potrebbe concludere nulla sulla regolarità del linguaggio sopra citato.

Si può provare applicando il pumping lemma che il linguaggio L di stringhe del tipo a^n*b^n*c^n per n >= 0 non è context-free, ma contestuale. Infatti, dovrebbe appartenere al linguaggio la stringa y*u^2*v*w^2*z. Ma u e w possono contenere diversi tipi di simboli, il che creerebbe delle inversioni, mentre porli uguali a un solo simbolo ripetuto non verificherebbe l'ipotesi per la quantità.

Rispetto all'**intersezione** l'insieme dei linguaggi context-free **non è chiuso**. Lo si può verificare così:

- L(1) = { 0^n*1^n*2^m | m, n >= 0 è tipo 2 }
- L(2) = { 0^m*1^n*2^n | m, n >= 0 è di tipo 2 }
- L(1) intersecato L(2) è { 0^n*1^n*2^n | n >= 0 } che NON è di tipo 2

Data una grammatica context-free è possibile decidere se essa genera un linguaggio vuoto. Tuttavia sono chiuse l'unione, la concatenazione e l'iterazione, come sono definite nei linguaggi regolari.

Regole di produzione:

- Nuovo simbolo iniziale S
- Eliminazione della parte destra vuota (o unitaria) (es. X -> eps)
- Riduzione a destra
- Normalizzazione

Esempio: A -> bCdE non è in forma di Chomsky.
Allora riscriviamo:
A -> bC'
C -> CdE
C' -> CD'
D' -> dE
Quindi una qualunque grammatica assegnata può essere riscritta come grammatica in **forma normale di Chomsky**.

Grammatiche di Chomsky: S -> epsilon, A -> BC e A -> a. Questa **forma normale** esiste per **ogni grammatica context-free**. Quindi possiamo assumere che ogni grammatica di questo tipo sia in forma normale di Chomsky.
Le regole per la trasformazione in forma normale di Chomsky sono:

- Usare un nuovo assioma **S' -> S**;
- Eliminazione delle epsilon-produzioni del tipo **X -> epsilon** (avvertenze: potrebbe trasformare il linguaggio L in L − {epsilon}); se il linguaggio contiene la epsilon, ricordarsi di **rimetterla sull'assioma** al termine delle trasformazioni;
- Eliminazione delle produzioni unitarie, quelle del tipo **X -> Y**, affinché ogni non-terminale ne produca altri due;
- Riduzione della parte destra: X -> a0X0a1X1...akXk => **X1 -> a0X0**, X2 -> a1X1, ecc.;

Per esempio:

A -> bCdE | CE => A -> bCd | C

Un caso limite:

A -> B | epsilon
B -> C
C -> A

Rimuovere l'epsilon-produzione qui importa **spostarla** su C, rimuoverla da C implica spostarla su B, e il ciclo ricomincia. Un modo per evitare questo è considerare quali non-terminali sono **annulllabili**; si dice che un non-terminale è annullabile se **A ->* epsilon**.
Assumiamo di avere Xk non terminali. Almeno uno di loro verifica la condizione, per definizione di epsilon-produzione. Ammettiamo ora **Yj ->* Xk**. Imponiamo che Y1, Y2... generino gli X, quindi; se l'insieme di simboli viene arricchito ripetiamo il processo finché è possibile ampliare l'insieme. L'analisi avviene solo sulle **produzioni** che producono **non-terminali**.

Per k produzioni di non-terminali, verranno prodotti **2^k** nuovi terminali. Ad esempio:
A -> BCDEF con C, D, E annullabili. Allora: CD, CE, DE, DF, C, D, E, epsilon, CDE.

Per quanto riguarda l'eliminazione delle **produzioni unitarie**, l'idea è la stessa. Per esempio:
A -> bCD | B | aa
B -> bb | C | ab
C -< c | CcA | CcB
Qui si può rimuovere direttamente B nella prima:
A -> bCD | *B* | aa
B -> bb | C | ab
C -< c | CcA | CcB
Rimuoviamo anche C dalla seconda:

A -> bCD | aa
B -> bb | *C* | ab
C -< c | CcA | CcB | *CcC*
Un simile funzionamento lo ha anche la riformulazione di terminali eventualmente tolti.

A -> BCD
[...]
A -> BA'
A' -> CD

A -> BCDE
[...]
A -> BA'
A' -> CA''
A'' -> DF

A -> BCdE
[...]
A -> BA'
A' -> CA''
A'' -> dE => A'' -> ZE, Z -> d

Nell'esempio della grammatica ambigua:
E -> E * E | E + E | (E) | i
Diventa:
E -> E + T | T
T -> T * F | F
F -> (E) | i
Proviamo che eliminando T troviamo:

E -> E + T | T + T
T -> T * F | F
F -> (E) | i

E -> E + T | T + T | T * F | F
F -> (E) | i

E -> E + T | T + T | T * F | F
F -> (E) | i

E -> E + T | T + T | T * F | (E + T) | (F + F) | (F * F)
F -> (E) | i

Così il linguaggio **torna a essere ambiguo**.

Per riconoscere i linguaggi di tipo 2, come detto, si usano **automi a stati finiti non deterministici con strutture a pila**. Ha due nastri, uno per la pila e uno per l'input. Sul "fondo" della pila avrà i riferimento al primo nodo. L'operazione di push può riguardare anche una stringa intera, mentre il pop si riferisce a *un elemento solo*.
Gli ASFND hanno quindi la capacità di **contare**. Con push() si rimuove un elemento. In sostanza, c'è equivalenza tra l'automa che prefrisce la riga vuota e quello che riconosce l'oggetto null.

Uno di tali automi è il push-down automa, costituito da:
- Sigma, l'alfabeto
- Gamma, i simboli
- z0 c R, simbolo iniziale della pila
- Q, insieme degli stati
- q0, stato iniziale
- F c Q, stati finali

La **funzione di transizione** è delta(Q * Sigma{eps}) -> P(Q, Gamma*), P(Q, Sigma*).
Per esempio: delta(q0, 6, e0) -> qd1, qa, qb
-> qk, epsilon
La pila può essere vista in due modi: vuota o che contiene **z0** all'inizio della compilazione, che comunque può essere rimosso con un pop. È però convenzionale non fare pop dell'ultimo simbolo, è il simbolo sentinella; se è necessario, si fa il **pop** di **eplison**.

Gli ASFND a pila hanno definizioni leggermente diverse a seconda delle scuole di pensiero. Il **riconoscimento della stringa** può avvenire al termine della lettura o all'arrivo in uno stato finale.
Definizione della funzione di transizione:
delta : Q x (Sigma $\underline{U}$ {epsilon}) x Gamma --> P(Q x Sigma*)
Dove P è l'insieme delle parti di Q x Sigma*. Lo stato successivo viene raggiunto dopo il push di una stringa, seguito per antonomasia dal pop.

Varianti:
Sigma $\underline{U}$ {epsilon} $\underline{U}$ {$} / {#} / {...}
Gamma $\underline{U}$ {epsilon}
Ce ne sono infinite, ma non cambiano la capacità di riconoscimento dell'automa.

Una **pila vuota**, intuitivamente, significa accettazione della stringa se è finito l'input.
Come esempio, prendiamo il linguaggio: {w*w~ | w $\underline{c}$ (a+b)^(+)}
E le produzioni: S -> aSa | bSb | aa | bb
Sarebbe semplice riconoscere il linguaggio partendo dal **centro stringa**: è sufficiente controllare che i caratteri affioranti a sinistra e a destra siano uguali. È possibile arrivare a un automa a pila partendo da questo concetto e sfruttando il **non-determinismo**: alla lettura di ogni stringa la diramazione dei casi avviene secondo la dicotomia appartenenza-prima-metà e appartenenza-seconda-metà. Nel secondo caso, la pila comincia a essere svuotata, e se è vuota nel momento in cui **finisce l'input** la stringa viene accettata.

I concetti di configurazione, relazione di transizione e computazione sono identici a quelli applicati ai linguaggi regolari: la **configurazione** è una tripla <q, x, gamma> con stato interno, stringa da leggere e stringa in pila; la **relazione di transizione** <q, x, gamma> |-- <q', x', gamma'> esprime la consumazione del carattere in ingresso e l'affioramento di un nuovo carattere nella pila che porta allo stato successivo q' (oppure l'epsilon-transizione dove la stringa x resta uguale anche nel caso in cui lo stato cambia); la **computazione** è la chiusura transitiva e riflessiva della relazione di transizione |--.

Un linguaggio può essere accettato per pila vuota o per stato finale. L'**accettazione per pila vuota** avviene solo se al termine della scansione la pila è vuota:
Null(M) = {x $\underline{c}$ Sigma | <q0, x, gamma0> |--* <q, epsilon, epsilon>}*
Mentre l'**accettazione per stato finale** avviene solo se al termine della scansione si giunge in uno stato finale:
L(M)= {x $\underline{c}$ Sigma | <q0, x, gamma0> |--* <q, epsilon, gamma>, q $\underline{c}$ F, gamma $\underline{c}$ F*}*

L è context-free se e solo se è riconoscibile da una **grammatica di tipo 2**. Questo enunciato è dimostrabile mostrando la reciproca inclusione dei due insiemi; intuitivamente, si mette un carattere speciale in pila, più quello inziale della grammatica, del quale vengono impilati in ordine inverso i **non-terminali** derivati oppure confronta il terminale con l'input. Il linguaggio viene riconosciuto qundo la pila di svuota.
La dimostrazione, ancora una volta intuitiva, del fatto che il linguaggio implichi almeno un automa che lo riconosce. Ad esempio, ad ogni coppia di stati associa solo un simbolo iniziale. *(Non richiesto all'esame)*

Dato un linguaggio **context-free**, non ci sono automi a **stati finiti deterministici** che lo riconocono se è della forma {w1... wnwn... w1). Già l'epsilon-transizione è fronte di non-

determinismo, poiché l'esecuzione si divide tra la lettura di epsilon e il passaggio a nuovo stato col primo carattere.
Tuttavia, una **sottoclasse dei linguaggi context-free** è riconoscibile dall'automa. Non si può riconoscere, per esempio, stringhe come 'abbaabba' che nel linguaggio arriverebbe allo stato finale dopo il primo 'abba', non esaminando il resto della stringa a meno che non sia stata predisposta un'analisi non-deterministica.

I linguaggi context-free non sono chiusi rispetto a **intersezione** e **complementazione** e i problemi dell'**uguaglianza** e della dimensione dell'**intersezione** di due linguaggi sono indecidibili. Il passaggio a linguaggi di tipi superiori aumenta la complessità dei formalismi; i linguaggi oltre il tipo 0 non hanno neanche una grammatica generativa.

AUTOMI

Gli automi a pila deterministici hanno un potere computazionale inferiore a quello degli automi a pila non deterministici. L'**automa a pila** memorizza le informazioni dell'**input**, leggendo per primo l'ultimo carattere inserito; la tabella di transizione per gli automi deterministici è "semplice", univoca, in quanto ogni combinazione va in uno stato solo. Consultando la tabella, l'automa realizza le transizioni.

Il **parsing** è un'operazione equivalente: ci si appoggia a una struttura dati di tipo pila, e si rappresentano le operazioni in tabelle. Quando una stringa non può essere parsata, oppure si verifica un errore a tempo di parsing, il compilatore non riesce a portare a termine l'operazione e mostra un messaggio di errore.

Nella **programmazione** si usano le grammatiche **context-free**. La definizione non è rigorosa, dato che per azioni come dichiarazioni di variabili (che non possono essere ripetute) mostrano dei comportamenti *context-sensitive*. Devono essere **deterministici**, affinché l'algoritmo si possa eseguire, e dalla grammatica non ambigua.

Ci sono due diversi approcci per la ricognizione della stringa: top-down e bottom-up. I **top-down** provano a ricostruire la stringa dalla fine, i **bottom-up** applicano le produzioni al contrario fino ad arrivare a un punto morto o all'assioma; nel secondo caso, la stringa viene riconosciuta, nel primo no.

La prima fase della compilazione è l'**analisi lessicale**, poi c'è l'**analisi sintattica** *(parsing)*, poi l'**analisi semantica** (costruzione dell'albero di derivazione) e infine l'ottimizzazione. Questi quattro passi portano alla formulazione del linguaggio macchina. L'*interprete* funziona come il compilatore, eccetto per il fatto che non ottimizza il codice e lo esegue anche, mostrandosi quindi significativamente più lento del compilatore.

Un **token** è una parte di un input che viene letto e classificato. Gli identificatori riservati non possono essere usati per descrivere variabili dichiarate autonomamente. Per esempio, nell'espressione C:
sum = 3 + 2;
Abbiamo sei token: sum è l'identificatore, 3 e 2 sono interi letterali, = è l'assegnazione, + è l'addizione e ; è il termine del'istruzione.

<Program> := program <ident>
Il token program è un terminale, quelli racchiusi in <> sono non terminali. Tale forma è detta **BNF**, **Backus-Naur Form**.

Un esempio di derivazione:

S -> S ; S
S -> id := E
S -> print(L)
E -> id
E -> num
E -> E + E
E -> (S, E)
L -> E
L -> L, E

Si nota che questa grammatica è ambigua: si può decidere ogni volta su che non terminale derivare. La scelta di cominciare a derivare dalla produzione più a sinistra è detta **left-most**, il contrario **right-most**, poi ci sono le scelte intermedie. In ogni caso l'analisi semantica (traduzione) è la stessa.

Per una grammatica ambigua si può effettuare una **disambiguazione** se il linguaggio non è inerentemente ambiguo. Per esempio:
E -> E + E
E -> E * E
E -> id
E -> (E)

Diventa

E -> E + T
E -> T
T -> T * F
T -> F
F -> id
F -> (E)

Notasi che si potrebbero anche invertire * e +, ma in tal caso il + avrebbe la precedenza sul * nel calcolo.

In principio la compilazione era un'operazione **multipasso**: a ogni lettura analizzava e processava l'input, dopodiché ricominciava, rileggendo a volta l'input diverse volte. Questo ovviamente penalizzava le prestazioni.

La **proprietà dei prefissi** prevede che a ogni sequenza di token *(prefisso)* che l'analisi riconosce come legale deve seguire un nuovo pezzo che mantenga questa proprietà, fino al termine dell'input, al fine di soddisfare le specifiche del linguaggio. Non è valida se il prefisso non può essere riconosciuto né come corretto né come scorretto.

L'analisi sintattica **bottom-up** costruisce l'albero di derivazione dal basso; a ogni passo avanza di un terminale oppure deriva un non terminale – dato che procede per **derivazioni inverse** – e quando l'input è esaurito, ossia arriva all'assioma, decide se accettarlo.
L'analisi sintattica **top-down** costruisce l'albero dall'assioma applicando produzioni per costruirlo. Una volta finiti i non-terminali, se è stata generata la stringa in input, essa viene riconosciuta.

Supponiamo di avere:
S -> wAalpha e di dover riconoscere wxy.
Se A -> sigma(0)alpha(0) e non ci sono altre produzioni per A, la scelta è univoca. Altrimenti, bisogna provare **tutte le scelte** di A finché non si trova quella corretta oppure si esauriscono tutte le possibili computazioni. Questa tecnica si chiama **backtracking**.

Una particolare implementazione di un **parser top-down** è detta **discesa ricorsiva**. L'idea di base è scrivere una funzione per ogni non terminale. Il funzionamento è semplice per non-terminali non ambigui. Ad esempio:

S -> aB
B -> bA
A -> b

Se invece uno o più non-terminali hanno **n produzioni**, vanno testate tutte prima di determinare quale porti alla soluzione. Se una stringa di una di queste combinazioni venisse riconosciuta senza aver completato l'output. È possibile fare una *diagnostica* degli errori inserendo stampe di debugging. C'è un caso in cui quest'idea causa uno **stack overflow**:
A -> Ab
Occorre in tal caso modificare le produzioni in modo tale da soddisfare le specifiche.

Il **parser predittivo** ovvia a questo problema in alcune condizioni: se in input c'è un if, lo utilizza come produzione.

Il **parsing LL(K)** esegue un *look-ahead* sull'input: parte da sinistra (L) ad analizzare la stringa, usa una tecnica *leftmost* (L), dopodiché con i prossimi (K) token decide quale regola utilizzare (questo è il look-ahead). Il parsing LL(K) prevede che le derivazioni siano sempre non ambigue. Il più interessante parsing LL(K) è il parsing **LL(1)**.

Linguaggi CF $\subset$ CF (deterministici) $\subset$ LL(K) $\subset$ LL(1) $\subset$ LL(0).

Data una **grammatica LL(k)**, guardando k token è possibile ricostruire la produzione. Le più comuni grammatiche LL(k) hanno LL(1).

Avendo una produzione A := alpha, definiamo FIRST(alpha) come i terminali che possono trovarsi all'inizio di alpha. Ad esempio:
A => alpha => a(Beta1) | b(Beta2)* => **FIRST(alpha) = {a, b}**
Allora una grammatica è LL(1) se, per ogni A := alpha e A := Beta, allora: **FIRST(alpha) $\cap$ FIRST(Beta) = insieme vuoto**

Nei casi di grammatica LL(1) la stringa viene analizzata da sinistra a destra iterativamente. Un esempio di grammatica LL(1) è:
ex -> digit | "(" ex op ex ")"
op -> "+" | ""*
digit -> "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
Mentre non è LL(1):
S -> Ac | Bc
A -> c
B -> c

Un **parser top-down efficiente** parte dal generatore di parser che analizza la grammatica in input, e la rifiuta se è ambigua; se non lo è, costruisce il parser che prende in ingresso i token e ne ricostruisce l'albero di derivazione.
FIRST(alpha) = {t | (t $\subset$ V(t) ^ alpha =>' B) v (t = epsilon ^ alpha =>' epsilon)}
Il FIRST di un simbolo non terminale è inizializzato a insieme vuoto e, applicando le produzioni, si cerca di aumentarne gli elementi.

Dato A -> B1B2...Bk, Ove Bi $\subset$ V, FIRST(A) = FIRST(A) + FIRST(B1) | epsilon
if(epsilon $\subset$ FIRST(B2)) FIRST(A) = FIRST(A) + FIRST(B2) | epsilon
[...]
if(epsilon $\subset$ FIRST(Bk)) FIRST(A) = FIRST(A) + FIRST(Bk) + epsilon
Il **primo test fallito** interrompe la procedura.
Il calcolo per FIRST(alpha) è analogo.

Una volta costruiti i FIRST, se la grammatica non è LL(1) è un errore; in mancanza di errore si procede con la creazione del parser a discesa ricorsiva. Ad ogni token **t ⊆ FIRST(alpha)**, si ricostruisce la produzione **A -> alpha**. Avviene tuttavia una perdita di produzioni se la **epsilon ⊆ all'insieme FIRST(alpha)**. In tal caso abbiamo bisogno di informazioni aggiuntive rispetto ai FIRST: i FOLLOW(alpha), definiti come segue:

*FOLLOW(alpha) = {t | (t ⊆ V(t) ^ S =>' alpha*A*t*Beta) v (t = $ ^ S =>' alpha*A)}*
$ denota la fine dell'input (come z0 nella stack).
L'inizializzazione di FOLLOW avviene direttamente su { $ }. FOLLOW(A) = empty per A != S.
Nella produzione **X -> alpha*B*Beta**, FOLLOW(B) = **FOLLOW(B) + FIRST(Beta) \ { epsilon }**

Se epsilon non fa parte dei FIRST(alpha), il SELECT(A -> alpha) si definisce come:
FIRST(alpha) ∪ FOLLOW(A), annullabile per epsilon ⊆ FIRST(A), altrimenti ha FIRST(alpha).
Una grammatica è LL(1) se per A -> alpha e A -> Beta, **SELECT(A -> alpha) * SELECT(A, Beta) = empty**.

Le grammatiche non sono più LL(1) se si incorre in **ricorsioni sinistre** oppure le parti destre di produzioni di uno stesso livello e prefisso condividono dei non terminali (es. A -> aBC | aCD). Cambiare la ricorsione in una ricorsione destra non è efficace se il prefisso è lo stesso (esempio: E -> E + T | T che diventa E -> T + E | T, e ha due T), ma vi si può ovviare con:
ex -> term exit
exit -> + term exit | epsilon

Nel caso generale di n ricorsioni sinistre:
A -> A*alpha1 | A*alpha2 | ... | Beta1 | Beta2 | ...
Si risolve con: (Beta1 + Beta2 + ...)(alpha1 + alpha2 + ...)*

Nel caso di prefissi comuni (non LL(1)) si usa la fattorizzazione:
A -> alpha*Beta | alpha*Gamma
Diventa:
A -> alpha*A'
A' -> Beta | Gamma

Si può basare il parser su una tabella di parsing.

Nei **parser top-down** i token presi in ingresso danno origine all'albero di derivazione che porta alla "creazione" della stringa. Quelli di categoria **LL** sono **left-most**, gli LR invece sono right-most, indicando rispettivamente che le produzioni vengono applicate ogni volta dal non-terminale più a sinistra o quello più a destra.
Si ricorda che il parser viene costruito solo se la grammatica **non è ambigua**.

Nell'**approccio LR** l'input viene messo in una pila e si procede attraverso due passi: *SHIFT* e *REDUCE*. Se un terminale fornisce la terminazione di una data produzione, lo si sostituisce con il non-terminale che lo produce.

Per evitare di ripetere dei passaggi o fare passaggi inutili, si usa il **look-ahead**, tecnica propria dei **parser predittivi**: LL(1) / LR(1) è l'ideale, poiché si capisce se un non-terminale fa parte di una produzione guardando solo il carattere successivo, anziché più di uno. Il look-ahead LL(1) è possibile solo per *determinati* linguaggi context-free **deterministici**; LR(1) è possibile per tutti i linguaggi context-free deterministici.

Il linguaggio in esame potrebbe anche non essere in forma LL(1), ma in alcuni casi si può **ridurre**. Per esempio:
S -> aB | aC
Diventa: S -> aS'
S' -> B | C

Per eliminare la **ricorsione sinistra**:
E -> E + T | T
Diventa: da A -> Aa | B
A -> B | BA'
A' -> aA' | epsilon

Il **SELECT** di una produzione definisce una produzione SELECT(A -> a) come:
- FIRST(a) + FOLLOW(A), oppure
- FIRST(a), se non si può annullare, ossia non produce epsilon

L'implementazione nel parser a discesa ricorsiva prevede che, a ogni iterazione, venga riconosciuta **una** delle possibili produzioni.

Una volta depositato l'input nella pila, viene consumato finché la verifica della stringa non è completa.
Secondo l'**approccio imperativo** il parser può essere ricondotto a una tabella di parsing con Vn linee e Vt colonne, rispettivamente associate a terminali e non-terminali (gli errori sintattici corrispondono a caselle vuote):

	$	a	b	c
S				
A				
B				

Nella **grammatica standard** per **espressioni aritmetiche**:
E -> E + T | T
T -> T * F | F
F -> id | (E)
In tal caso:
FIRST(F) = {id, (}
FIRST(T) = FIRST(F) per la produzione T -> F
FIRST(E) = FIRST(T) = FIRST(F)
FIRST(E + T) = FIRST(E)
FIRST(T * F) = FIRST(T)
Avere tutti risultati uguali rende impossibile la semplificazione.

Quindi:
E -> TE'
E' -> TE' | +TE' | epsilon
T -> FT' | *FT' | epsilon
F -> id | (E)
FIRST(T) = FIRST(E) = FIRST(TE') = FIRST(FT') = {id, (}
FIRST(E') = FIRST(+TE') = {+}
FIRST(T') = FIRST(*FT') = {*}

Il trattamento degli **errori sintattici** è più complesso: l'errore deve essere localizzato, diagnosticato e riportato al fine di essere corretto. I parser top-down permettono il trattamento degli errori.

CALCOLABILITA' & COMPLESSITA'

MACCHINE DI TURING

Due dei problemi chiave dell'informatica sono la **calcolabilità** e la **complessità**. Rispondono entrambi a domande fondamentali: cosa è possibile calcolare con un computer, e quanto costa farlo? La **tesi di Church-Turing** afferma che è calcolabile tutto ciò che può essere calcolato da una macchina di Turing.

Le **macchine di Turing** sono macchine astratte (ossia modelli di calcolo) introdotte da Turing con lo scopo di dare una definizione formale per ogni algoritmo e studiare i limiti della computazione. Gli automi sono costituiti da una testina di lettura e scrittura *bidirezionale*. È sufficientemente generalizzata per poter simulare qualunque algoritmo.

L'idea di Turing riguardava scrivere il programma sul nastro, in modo tale che potesse essere modificato dalla macchina stessa successivamente. Per questo Turing è considerato il padre dell'**intelligenza artificiale**.

Anche le macchine di Turing sono dotate di stati iniziali e finali. Trattandosi di coppie ordinate di stati, l'arco da q0 a q1 non è lo stesso che va da q1 a q0. La tripla: **(sigma1, tau1, m1)** rappresenta il *simbolo letto*, *scritto*, e il *movimento della testina*. Tale macchina è **deterministica** e avrebbe comunque lo stesso potere di ricognizione del suo equivalente non deterministico.

Una classica rappresentazione per le macchine di Turing è quella basata sulla sua funzione di transizione, descrivibile mediante **rappresentazione tabellare**. Le tabelle hanno una quintupla *(stato di partenza; simbolo in ingresso; stato di arrivo; simbolo in uscita; movimento della testina)*. Nei simboli sono compresi **0**, **1** e ***blank***, che indica il termine della lettura.

Più difficile è determinare quando l'esecuzione termina. Il cosiddetto "stato finale" si riduce ad una **posizione** per la quale non è definita **nessuna transizione**.
La **configurazione minimale** si rifà al concetto di configurazione istantanea, che descrive lo stato attuale dell'automa e la porzione di stringa ancora da leggere. La differenza è che la macchina di Turing, essendo bidirezionale, non ha una porzione definita di stringa ancora da leggere. Perciò viene definita con la rappresentazione **x^q*y**, ove q è lo stato e la testina si trova sul primo simbolo di y.
La **configurazione di accettazione** prevede che x^q*y abbia q c F.

Le regole di transizione definite per le macchine di Turing comportano che **la terminazione della computazione non è assicurata**. Del resto non esiste neanche un modo per riconoscere se la macchina è in loop.
Dalla configurazione C1 si produce C2 solo se è definito un singolo passo di calcolo tra le due.

Con una macchina di Turing è possibile implementare un **complementatore** di numeri binari (algoritmo: inversione di ciascun bit e somma di 1). La macchina:
- Raggiunge il primo *blank*
- Trova e supera il primo 1
- Si sposta a sinistra fino a tornare a *blank*
- Si sposta a destra

Il costo complessivo dell'algoritmo è **O(n)**.

Altra implementazione possibile attraverso le macchine di Turing è l'**ordinamento lessicografico**, es. 010101101 -> 000011111. Nel caso peggiore, l'algoritmo costa O(n^2), perché parte dall'inizio e, dopo aver incontrato un 1 e uno 0, comincia a scambiarli di posto riscandendo ripetutamente la stringa fino al termine delle sostituzioni.

Possiamo anche considerare una macchina che riconosca una stringa, ad esempio per il linguaggio {0^n*1^n : n >= 0}. Una prima implementazione può consistere nell'usare la porzione *blank* del nastro per **contare gli zeri**, dopodiché **decrementare** il contatore

individuando gli uno. L'*upper bound* dell'algoritmo è O(n^2) poiché necessita di scandire n volte 2n + log(n) celle (il contatore, gli zero, gli uno).

È possibile operare con una **macchina di Turing multi-nastro**. La testina si può spostare a destra, sinistra, in alto, in basso e potenzialmente anche in diagonale o non spostarsi affatto. Nonostante la multidimensionalità e il passaggio al non-determinismo, si è constatato che **la potenza è la stessa della macchina deterministica a nastro singolo**. Tuttavia, ha introdotto alcune migliorie: la macchina multinastro può annotare risultati su nastri aggiuntivi vuoti, e/o scrivere l'output su un nastro separato. In sostanza tiene separati nastri di input, output, e di lavoro.

L'algoritmo precedente, implementato con una macchina multi-nastro, ha una complessità temporale pari a **O(n)**: per ogni zero letto annota un 1 sul nastro di output, dopodiché li cancella mentre legge gli 1. Se al termine della lettura il nastro di output è vuoto, la stringa viene riconosciuta.

Il **tempo utilizzato** non può essere **mai inferiore allo spazio**, il che significa che per attraversare un'unità di spazio è necessaria almeno un'unità di tempo, anche nelle **macchine di Turing**. Il fatto che algoritmi come la ricerca binaria funzionino in tempo sublineare (O(log(n))) per input lineari (O(n)) non è una contraddizione, in quanto nel calcolo del costo temporale si include il costo speso in ogni unità di spazio *effettivamente attraversata*.

Le macchine di Turing **multinastro** non hanno potere computazionale superiore a quello delle macchine a nastro singolo, ma permettono di velocizzare le operazioni. Si può inoltre definire una **"macchina minima"**:
Sigma = {0, 1, *blank*}
Stati: s0 e s1
Nastro semi-infinito (ha un inizio, non ha una fine)
Ha lo *stesso potere computazionale* di qualunque altra macchina di Turing.

Supponiamo di avere una macchina con **k nastri** e di volerla simulare con un'altra macchina. Potremmo ad esempio prendere un nastro diviso in **tracce** (es. corsie orizzontali parallele) oltre che in caselle. Le tracce pari corrispondono ai nastri; le tracce dispari sono tutte blank, eccetto alcune caselle che tengono traccia della posizione della testina.
Nel caso della macchina a nastri, le testine *non sono sincronizzate* e hanno quindi, al tempo *t*, una distanza massima pari a *t*. La macchina a tracce, per risincronizzare le testine, deve **scandire le caselle** una ad una prima verso destra e poi verso sinistra. Risulta quindi evidente che questa complicazione comporta un maggiore costo computazionale, con un margine di circa il 100% (t -> 2t).

Ad ogni istante possibile, la macchina sopra descritta *(che NON è di Turing!)* deve poter leggere 2k informazioni. A questo scopo si può **aumentare** il numero di caratteri in ingresso. La **cardinalità** del primo nastro, Gamma1, ha la stessa cardinalità (= lo **stesso alfabeto**) di Gamma3; idem per Gamma2 e Gamma4. Quindi, si ridefinisce un alfabeto Gamma la cui cardinalità è pari al **prodotto di tutte le cardinalità** dei sottonsiemi Gamma1-Gamma4. Questo significa, a parità di cardinalità dei nastri pari e di quelli dispari, che il tempo necessario a una macchina a nastro singolo è comunque **quadratico** rispetto a quelle a nastro multiplo: la differenza sostanziale la si vede passando proprio da 1 a 2 nastri.

Il concetto di sottomacchina di Turing è un concetto simile al sottoprogramma: prevede che, arrivati in un certo stato "speciale", si affidi la computazione a una sottomacchina.
Invece la **macchina di Turing universale** nasce dall'esigenza di far sì che una macchina non esegua solo un tipo di computazione. Una macchina di Turing si può definire con *stored program*, in modo tale che esegua automaticamente le sue operazioni, oppure **associare** fisicamente **un programma al nastro** a tempo di esecuzione, e cambiarlo di volta in volta.

La MdT universale può anche modificare e *migliorare* il suo programma. Questo è il principio fondatore dell'***intelligenza artificiale***. Turing ha inoltre dimostrato che per costruire una MdT

universale è sufficiente costruirne il programma e un input. In questo contesto, per **"programma"** si intende una macchina di Turing che svolge una singola funzione.

L'alfabeto utilizzato dai computer moderni è lo Unicode a 16bit (precedentemente ASCII a 8bit). Le codifiche devono seguire date regole; ad esempio, se A = 0, B = 1 e C = 00, la stringa 000 non ha codifica univoca. Per renderla univoca occorre **assegnare** a ciascun carattere un **numero di bit** pari al **log(k)**, dove k è il numero di caratteri nell'alfabeto. In tal modo non si creano ambiguità.

È possibile anche ottimizzare attraverso la **codifica di Huffman**: le lettere meno frequenti vengono "unite" per permettere di rappresentare alcuni termini con meno bit.
La codifica si può effettuare via albero binario: ogni carattere viene messo in una foglia. Poi si sommano i valori dei **due più piccoli**. Questa somma, insieme alle altre foglie, concorre all'associazione dei successivi due più piccoli. Giunti alla cima, ossia 1, ci si ferma.
Oltre all'approccio di Huffman esiste anche l'approccio stocastico.

Una MdT con un alfabeto *ad almeno 3 simboli* (di cui uno è il blank) si può cambiare con una macchina Sigma1 che svolga le stesse identiche funzioni e abbia come alfabeto solo {0, 1, *blank*}: stati d'ingresso e d'uscita, terminazione, operazione; a nastro singolo, tuttavia, per **ogni passo** richiede **k passi**.
La MdT universale è alla base dello sviluppo dei calcolatori di **Von Neumann**. Hanno l'alfabeto minimo {0 = Z, 1 = U, *blank* = B} e tre movimenti {L, R, S} (left, right, stop). Per rappresentare gli stati si usano numero binari.
Esempio: 0, q0, 0, 1, R => 0Z0UR

Si può inoltre fare sì che gli **stati** siano solo **0 e 1**, ove 0 è quello iniziale (unico non finale) e 1 è quello finale. La realizzazione più naturale prevede tre nastri: uno per l'input, uno per l'output, e uno di lavoro; prima viene **copiato l'input codificato**, poi inizializzato il nastro di output con lo stato iniziale, cerca le transizioni possibili e le applica (o **rifiuta la stringa** se non esistono più produzioni) e applica la transizione, ripetendo il processo fino a giungere al rigetto o allo stato finale.
Il motivo dell'**indecidibilità** di alcuni problemi è dato che, nonostante queste condizioni, non è garantito che MdT riconosca qualsiasi linguaggio.

Cosa succede dando in input alla **macchina di Turing universale** la stessa codifica della macchina di Turing universale? Un input sensato potrebbe essere la codifica seguita da una **stringa di input** da processare. Questo procedimento può essere ripetuto infinite volte. Tuttavia, il doppio livello di codifica è superfluo: l'output è lo stesso.
Ragionamenti di questo tipo portano a dei paradossi che aiutano a definire i *limiti* delle macchine di Turing.

Una configurazione finale della MT è una configurazione per la quale la macchina si trova in uno stato finale. Una configurazione di *non accettazione* corrisponde invece a uno stato-pozzo non finale.
Prima e dopo una stringa di input, ci sono dei **blank**. Ogni stringa deve quindi essere isolata. La codifica dell'output è la stessa. La differenza tra *input/output* e *codifica di input/output* risiede nel fatto che la seconda utilizza un **linguaggio binario**.

f(x) = y è la **funzione calcolata** della macchina di Turing. Questo assume che la macchina sia giunta in una configurazione finale. La macchina calcola una **funzione parziale**, poiché non esiste una y per ogni x, ossia la funzione è definita solo su una parte dell'input. Una funzione di questo tipo è detta **T-calcolabile**. Si può definire anche semplicemente "calcolabile", perché *ogni funzione calcolabile è calcolabile con una macchina di Turing*.

DECIDIBILITA'

La **funzione caratteristica** del linguaggio è una funzione uguale a 1 se la stringa in input appartiene al linguaggio, 0 altrimenti. Un linguaggio è **decidibile** solo se la sua funzione caratteristica è T-calcolabile; è invece **semidecidibile** se risponde 1 per ogni stringa appartenente al linguaggio, e risponde 0 oppure va in loop per ogni stringa non appartenente al linguaggio.

Esempio: *L* = {1^n: n = 2k} ha uno stato iniziale, uno intermedio (per i k dispari) e uno finale (per i k pari).
Se *L* è decidibile, anche il suo **complementare** *L** è decidibile. È possibile formare una macchina di Turing che lo rappresenti invertendo i caratteri della risposta (0 -> 1, 1 -> 0).
Se *L* è semidecidibile, il suo complementare è non decidibile.
L è **decidibile** <=> esiste una MT con **uno stato finale** che **termina per ogni input** e accetta solo le stringhe che appartengono al linguaggio.

È sempre possibile ridurre una macchina di Turing a una macchina ad un solo stato finale. Si può anche *collegare uno stato intermedio non accettante con uno stato finale* aggiungendo delle transizioni che scrivano sull'output 0 o 1 a seconda se la stringa deve essere accettata o meno, in modo tale da **assicurare la terminazione**.

Due insiemi A e B hanno la **stessa cardinalità** se esiste una funzione biettiva f : A -> B che li collega. Gli insiemi equicardinali a N sono *numerabili*, e hanno la caratteristica di essere equicardinali anche ai loro sottoinsiemi propri.
Prendiamo ad esempio l'insieme delle frazioni (*non è* l'insieme dei razionali!) e dividiamolo in classi di equivalenza. Diciamo che a/b R c/d | ad = bc.
La corrispondenza con Q è data dalla **funzione coppia di Cantor:**
Cantor Pi(i, j) = (i + j)(i + j + 1) / (2 + j)

L'insieme dei reali non è numerabile. Supponendo per assurdo che lo sia, e ponendo q -> 1/q, appare chiaro che l'intervallo tra 0 e 1 contiene una parte sostanziale dei numeri reali. Se dimostriamo che l'intervallo (0, 1) sia numerabile, sarà numerabile anche R. Supponiamo ancora per assurdo che lo sia e costruiamo la **tabella delle corrispondenze**; costruiamo un numero che sia tale che le sue cifre siano pari alla diagonale della tabella più uno. Tale numero non è descrivibile con la tabella appena costruita, quindi (0, 1) non è numerabile, quindi R non è numerabile. c

Gli insiemi equicardinali ai naturali sono indicati con **Aleph-zero**, mentre i reali sono indicati con **Aleph-one** (ordini d'infinito).

L'**ordinamento lessicografico** è un ordinamento naturale delle stringhe, applicabile anche alle macchine di Turing. Ogni stringa presa in ingresso da una macchina di Turing deve essere numerabile; è numerabile anche l'insieme delle macchine di Turing, così come l'insieme dei linguaggi decidibili, ma l'**insieme dei linguaggi non decidibili** è **non numerabile**. Lo si può dimostrare via diagonalizzazione: enumerando tutte le stringhe binarie di lunghezza n, con n arbitrario, e scrivendo agli "incroci" 0 o 1 a seconda se la stringa appartiene o meno al linguaggio, invertendo la diagonale si trova una stringa che non appartiene al linguaggio.

Un esempio di problema non decidibile è la **correttezza di un programma**. Non è possibile decidere se un programma termina per ogni input, tantomeno verificare che sia corretto.
Definiamo il linguaggio L-stop: {<ct, x> c C ^ x c Sigma* ^ T(x) termina)
La macchina T-stop **non esiste**: possiamo dimostrarlo supponendo per assurdo dalla complementare che termini solo se y non appartiene al linguaggio, e può non terminare se vi appartiene. Tuttavia, in base a questa definizione, se T[diag] prendesse T(-1)[diag] in input, non terminerebbe – quindi si ha un assurdo e T-stop non esiste. c

Altro esempio di linguaggio **non decidibile: T-stop complemento**.
Oppure, sapere se una data istruzione di un programma sarà mai eseguita.

Non si può decidere se una grammatica di tipo 2 è ambigua né se due grammatiche di tipo 2 sono equivalenti.
Non è possibile neanche provare che una macchina si fermerà con input = 0.

Strumento chiave delle dimostrazioni è la **riduzione**: dimostrare che una soluzione a un problema A può essere trovata cercando la soluzione del problema B, oppure provando che B non è decidibile.

HALTING PROBLEM

Il **problema della terminazione** o *halting problem*, ossia il saper determinare se dato un input la computazione di una macchina di Turing terminerà o meno, è indecidibile.

Altro problema indecidibile collegato all'halting problem è il determinare se il programma termina per **input 0**: supponendo per assurdo che sia decidibile, si può controprovare l'assunzione ipotizzando di risolvere *L(stop)* attraverso *L(stop-0)*. Stabiliamo che abbia in ingresso <ct, x> che appartiene a L solo se T(x) termina, e con 0 solo se T(0) termina. Una macchina ausiliaria T'(x) viene data in input a T(x) e termina solo se T(x) termina. T'(x) **cancella l'input**, poi lo processa. Quindi attraverso T(stop-0) si costruisce T(stop), pertanto T(stop) <= T(stop-0). Questo porta a una contraddizione perché T(stop-0) è indecidibile. c

---

Altri problemi indecidibili: sapere se L è accettato da una macchina T; sapere se L è vuoto.
La **riduzione** può essere descritta da una funzione che associa a ogni elemento del dominio (insieme delle stringhe riconosciute) ad almeno uno del codominio (valori assunti dalla computazione per date funzioni), e questo solo per gli elementi del dominio.

Il problema dell'**involucro convesso** *(convex hull)* ha un lower bound di O(n*log(n)). A partire da un insieme che contiene tutti i punti, si forma un nuovo insieme di coppie di punti, in cui ciascuno è associato al suo quadrato. L'algoritmo *ordina i punti* (da qui la motivazione che porta a stabilire il lower bound) e considera quelli posti sulla stessa **parabola**.
Rivestono particolare importanza, in questo ambito, le riduzioni polinomiali. La direzione della riduzione **non è equivalente** (ovvero a -> b != b -> a) e ha un suo **costo**.

Esistono anche macchine di Turing che possono scrivere la propria stessa codifica, dette macchine **autoreferenziali**. Il problema dell'autoreferenzialità ha costituito la base della dimostrazione di incompletezza dell'aritmetica fornita da Godel.
** Il file SelfRef.java fornisce un esempio di programma autoreferenziale.*

Il modello di calcolo proposto con la macchina di Turing è semplice e potente, permette di definire il concetto di algoritmo, calcolabilità e decidibilità. Esistono macchine più potenti delle macchine di Turing? Quelli proposti finora sono **al più equivalenti**. La **tesi di Church-Turing**, solida e universalmente approvata, sostiene che ogni funzione calcolabile è anche T-calcolabile; l'asserto non può tuttavia essere provato, perché comporterebbe il confronto diretto delle macchine di Turing con **qualsiasi** altro modello di calcolo.

Riconoscibilità -> *Decidibilità*
Accettabilità -> *Semidecidibilità*

Le macchine di turing **riconoscono** linguaggi **context-sensitive** e **accettano** linguaggi di **tipo 0**. Quindi i linguaggi fino al tipo 1 sono decidibili, mentre quelli di tipo 0 sono semidecidibili. Esistono grammatiche per i linguaggi almeno semidecidibili, ma non per i problemi decidibili. Ci sono inoltre diversi livelli di indecidibilità.

Le **macchine di Turing non deterministiche** hanno lo stesso potere computazionale delle macchine di Turing deterministiche. Si può dimostrare che, data una macchina non deterministica, è sempre possibile simularla con una macchina deterministica, sebbene la computazione diventi più complessa. La macchina non deterministica rifiuta se e solo se **tutti i rami** portano al rifiuto della configurazione d'ingresso; l'accettazione avviene se esiste anche **un solo ramo** che accetta la configurazione. La tecnica di visita è **breadth-first**: in tal modo, le eventuali computazioni accettanti vengono trovate nel minor tempo possibile. Quindi, l'equivalente deterministico di una simile computazione richiederebbe *tempo esponenziale*.

Viene dunque definito **linguaggio di tipo 0** un linguaggio per cui esiste una macchina di Turing non deterministica che **semidecide L**. Tale macchina verifica, per ogni ramo, se la computazione è accettante; se non lo è, e se ci sono produzioni ulteriori, continua la computazione, senza garanzia che termini. Il non determinismo non è strettamente necessario, ma velocizza le operazioni.

Il collegamento tra linguaggi context-sensitive e le macchine di Turing è di natura simile: un **linguaggio context-sensitive** è un linguaggio per cui esiste una macchina di Turing che **decide L**. La computazione avviene in maniera analoga, ma la terminazione è garantita dal fatto che se la configurazione non viene accettata la si rifiuterà perché applicando una produzione data la stringa sarà più corta dell'originale, il che infrangerebbe una delle regole delle grammatiche context-sensitive.

I problemi risolvibili (upper bound) in **tempo polinomiale** hanno la caratteristica di essere polinomiali rispetto a tutti i modelli di calcolo, anche se con costanti a esponente diverse. Anche le loro estensioni sono risolvibili in tempo polinomiale. Fanno tutti parte della **classe P**, la classe dei problemi T-calcolabili in tempo n^k con macchine di Turing *deterministiche*; la sua equivalente per macchine di Turing *non deterministiche* è la **classe NP**. Intuitivamente si può dire che P != NP, ma l'assunto non è ancora stato dimostrato.

COSTO COMPUTAZIONALE

Le due risorse fondamentali nell'informatica sono il tempo e lo spazio; delle due, la più rilevante è il tempo. Se un algoritmo usa un certo quantitativo di spazio, lo usa effettivamente: la funzione spazio **s(n)** implica un utilizzo minimo di tempo pari a **Theta(s(n))**. Viceversa, un impiego di tempo pari a Theta(n) implica un impiego di spazio pari a O(n), poiché può anche essere inferiore a Theta(n).

La **classe P** risulta più importante delle altre per la sua inclusività: non specificare l'esponente in n^k comporta più intercompatibilità tra i diversi modelli di calcolo, poiché a meno della costante k ciò che è calcolabile in tempo polinomiale è calcolabile in tempo polinomiale in **tutti i modelli di calcolo**.

Viene spontaneo domandarsi se la **codifica** incida o meno sul **costo** di un algoritmo. Ipotizziamo di rappresentare un numero n in unario. Se abbiamo un algoritmo che ha un ingresso z in binario e costa O(z), passando al sistema unario il costo diventa O(log(n)).
Tuttavia, tale salto avviene solo al passaggio alla base unaria: per le codifiche in **base maggiore di uno** i costi asintotici sono gli stessi.

Ci si può anche chiedere se sia restrittivo limitarsi a problemi sui linguaggi. Si possono dare diversi esempi algoritmici di problemi applicati all'informatica, per esempio ai **grafi**:

- *Decisione*: esiste un percorso A -> B?
- *Ottimizzazione*: qual è il percorso A -> B più breve?
- *Enumerazione*: quanti percorsi A -> B esistono?
- *Ricerca*: dato x, trova un percorso A -> B di lunghezza minore di x.

I problemi sui linguaggi si limitano ai **problemi decisionali**.

Fino a che punto si può riutilizzare la soluzione algoritmica del problema decisionale sopra riportato, per risolvere anche gli altri tre? Per rispondere a questa domanda ipotizziamo che la soluzione al problema decisionale sia già fornita. In prima istanza possiamo confrontare, per il problema di ricerca, il valore x col valore **M** costituito dalla **somma dei pesi** di tutti gli archi. Già se x > M si può rispondere positivamente alla domanda. Altrimenti, si può dimezzare progressivamente l'intervallo M (come nel teorema degli zeri) fino ad arrivare a un unico valore. Riassumendo: è possibile **risolvere problemi non decisionali** attraverso modelli risolutivi del loro problema decisionale collegato.

I problemi in **P** sono considerati **trattabili**, quelli al di fuori di P sono intrattabili. Esaminiamo brevemente dei possibili costi asintotici per un'operazione atomica di 10^(-9) secondi:

	Costo	**Costo**	**Costo**	**Costo**	**Costo**
Dimensione	**n^2**	**n^3**	**n^5**	**2^n**	**3^n**
10	0.1 ms	1 ms	0.01 ms	1 ms	59 ms
30	0.9 ms	27 ms	24.3 ms	1 s	2.4 giorni
50	2.5 ms	0.125 ms	0.31 s	13 giorni	2.3 * 10^5 secoli

* Garey e Johnson hanno pubblicato una raccolta di problemi a costo esponenziale nel 1979. Da allora i progressi per gli algoritmi trattati non sono stati incisivi.

Un esempio classico di algoritmo polinomiale è l'algoritmo di Euclide che calcola il massimo comune divisore in una coppia di numeri nel seguente modo:
if(b == 0) return a; else return Euclid(b, a % b)
Il costo è **logaritmico rispetto al valore di b**, ovvero **lineare rispetto alla dimensione di b**.

SODDISFACIBILITA'

Altra classe di rilievo è **K-SAT**, la classe della **soddisfacibilità**, che stabilisce se una formula booleana in forma normale congiuntiva CNF (AND di OR) è soddisfacibile. Un noto algoritmo appartenente a questa classe è il **DPLL**. In K-SAT ogni espressione ha esattamente K letterali; tuttavia c'è una grande differenza già tra 2-SAT e 3-SAT: il primo è polinomiale, il secondo è esponenziale.

La polinomialità di 2-SAT può essere spiegata col suo funzionamento: presa una variabile **x**, le assegnamo il valore **true**. Ogni altra clausola è dichiarata non (ancora) soddisfatta. Si sceglie a caso una clausola **non soddisfatta**; se c'è un true, la si dichiara soddisfatta e si ripete il passo, se invece c'è un false su uno dei due si assegna true all'altro e si dichiara la clausola soddisfatta, mentre se entrambi sono false e anche x ha false si rifiuta, altrimenti si assegna **false** a x e si ripete la procedura.

Portando l'enunciato in **DNF** (OR di AND) il costo diventa lineare: basta valutare ogni espressione assegnando nelle parentesi i valori che la verificano, e controllare se almeno una espressione è verificata (il che avviene sempre a meno di un'espressione in cui si presenta x ^ !x). Verrebbe quindi spontaneo pensare di usare l'**algoritmo per DNF anche per CNF**, ma la conversione da DNF a CNF ha costo esponenziale.

Le riduzioni interessanti sono quelle polinomiali. Un esempio di riduzione esponenziale è invece il problema della **3-colorabilità**. Si può tradurre in un problema sui grafi: se ogni nodo è tale che (u, v) ha col(u) != col(v) il problema è risolto. Ogni grafico è 4-colorabile.
Altro esempio di algoritmo a costo esponenziale è il **cammino Hamiltoniano**, anche noto come **problema del commesso viaggiatore**, che consiste nel domandarsi se un grafo ammette un cammino che passi per tutti i nodi una sola volta.

Gli algoritmi sopra presentati hanno tuttavia una caratteristica in comune: data una soluzione per il problema, eventualmente trovata tramite un passo di **guessing non-deterministico**, può risolvere l'algoritmo a partire da quella soluzione in tempo polinomiale. Alcuni algoritmi non sono tuttavia riducibili a questa forma.

Se una formula **CNF** è soddisfacibile, esiste una sua assegnazione che, in tempo polinomiale, può essere svolta e verificare che la proposizione è soddisfacibile. Il problema della 3-colorabilità (sottogruppo della **k-colorabilità**) è anch'esso, molto probabilmente, di classe NP, anche se vale lo stesso principio per una configurazione che colora il grafo in maniera corretta. Lo stesso vale per il **cammino Hamiltoniano**: dato un grafo percorribile secondo Hamilton si può verificare in tempo polinomiale la strada che conduce alla soluzione. Un altro problema correlato è il **partizionamento** di un insieme in due sottoinsiemi la cui somma dei numeri sia uguale, che ammette soluzione polinomiale per insiemi che hanno partizioni perfette.

I problemi sopra elencati come alcuni dei più difficili problemi di **NP**: i problemi **NP-completi**. Un linguaggio è NP-completo se $\subseteq$ NP e per qualunque L' $\subseteq$ NP => L' <= L. Attraverso una riduzione polinomiale è possibile trasformare ogni problema di NP in un NP-completo. La loro forma ridotta è NP-HARD.

Il **teorema di Cook** afferma che SAT è NP-HARD e si trova in NP. Ha anche provato che le formule booleane possono assumere qualsiasi valore. La macchina deve essere opportunamente settata e funzionante. Poco dopo che Cook rivelò la sua scoperta uscirono altri 100 problemi NP-completi.

Se L' <= L(0) L è NP-HARD.
SAT $\subseteq$ 3-SAT. Anche 3-SAT è **NP-completo.** Ogni sua sottovalutazione ha una a e un f(!a)).
Per le clausole a **letterali multipli**, ad esempio: (a v !b v c v d v !e)
=> (a v !b v E) ^ (!E v c v F) ^ (!F v d v !e)

La scomposizione avviene raggruppando i primi due e gli ultimi due termini in clausole separate, connettendo tutti gli altri tra le due formule così ottenute. Da **k clausole** se ne ottengono **2 + k – 4**. Le variabili aggiuntive introdotte possono essere soddisfatte anche solo soddisfacendo una parentesi che le contenga, rendendole così ininfluenti sul resto dei risultati ancora da verificare.
Abbiamo così dimostrato che ogni problema SAT può essere **ricondotto** a un problema **3-SAT**, che è quindi NP-completo. c

Il problema del **vertex cover** consiste nel trovare un sottoinsieme di vertici con massimo k elementi tali che, comunque presi due sottoinsiemi del grafo, esiste sempre un nodo che li collega. Anche questo è un problema NP-completo.
Scomponiamo quindi il grafo in **triangoli**, con ogni triangolo contenente ai vertici le tre variabili in una data parentesi. I vertici del triangolo vengono poi collegati ai vertici che rappresentano le clausole in esso contenuti, mantenendo tanti archi quante sono le variabili e aventi ai due estremi la **variabile** e la **variabile negata**. Ci sono n variabili (nodi) e m clausole (archi). La condizione quindi è che k = n + 2*m.
Per soddisfare una clausola è sufficiente che uno solo dei vertici del triangolo sia verificato. I rami della clausola non verificata (ad esempio i rami !a se la clausola verificata è a) sono tolti dal grafico. Se un triangolo si trova completamente **scollegato** da vertici, la clausola non è soddisfacibile.

IndependentSet: G = (V, E) V' $\subseteq$ V (u, v) $\subseteq$ (v' x u') (u, v) !$\subseteq$ E |v'| >= k

Altro problema decisionale famoso è il **PL{0, 1}** che consiste nell'assegnazione di valori alle incognite di un **sistema di disequazioni** in modo tale da verificarle. Anche questo è NP-completo. Si può verificare la sua appartenenza a NP ipotizzando di assegnare, con un passo non deterministico, tutte le possibili assegnazioni alle variabili, e procedere per ogni set di assegnazioni con passi deterministici. È anche NP-hard.

Esempio di **riduzione**: (x v y v !z) => x + y + (1 – z) > 0