

Gocce di Java

Metodi e ricorsione

Pierluigi Crescenzi

Università degli Studi di Firenze

- ▶ **Interfaccia** di un metodo
 - ▶ Consente di utilizzarlo senza sapere nulla dell'implementazione
 - ▶ Consente di implementarlo senza sapere nulla del suo utilizzo
- ▶ Deve dire
 - ▶ Cosa fare per invocare il metodo (sintassi)
 - ▶ Cosa il metodo è in grado di fare (semantica)
- ▶ Sintassi: regole molto rigide
 - ▶ Dichiarazione e definizione
 - ▶ Nome del metodo
 - ▶ Dati passati al metodo dall'esterno
 - ▶ Tipo di dato dell'eventuale valore di ritorno
 - ▶ Corpo del metodo
 - ▶ Invocazione
 - ▶ Nome del metodo
 - ▶ Lista di valori racchiusa tra parentesi tonde
- ▶ Semantica
 - ▶ Può essere specificata in modi diversi (spesso imprecisi)

Istruzioni return

- ▶ Un metodo non void deve includerne almeno una
 - ▶ Includerne una non implica che sia eseguita

```
int massimoDueNumeri( int n1, int n2 ) {  
    int massimo = n1;  
    if (n1<n2) {  
        massimo = n2;  
        return massimo;  
    }  
}
```

▶ Parametri formali

- ▶ Spazi da riempire con valore al momento di invocazione

▶ Esempio

```
double numeroNepero( double epsilon )
```

- ▶ epsilon: parametro formale

▶ Argomenti

- ▶ Valori con cui riempire gli spazi bianchi

▶ Esempio

```
double numeroE = numeroNepero( 0.00001 );
```

- ▶ 0.00001: valore da assegnare a epsilon
- ▶ Assegnazione avviene come prima cosa
 - ▶ Parametro formale è variabile inizializzata con valore di argomento corrispondente

Massimo comun divisore di numeri

- ▶ Dichiarazione del metodo

```
int mcd( int a, int b )
```

- ▶ Invocazione del metodo

```
int divisore = mcd( 90, 42 );
```

- ▶ 90 assegnato ad a
 - ▶ 42 assegnato a b
- ▶ Meccanismo di sostituzione basato su **passaggio per valore**
 - ▶ Se argomento è variabile: solo valore viene assegnato
 - ▶ Se parametro formale modificato: variabile non cambia
 - ▶ Diverso per array e oggetti

- ▶ Variabile dichiarata all'interno di un blocco
 - ▶ Sparisce quando esecuzione del blocco termina

- ▶ Esempio

```
{
    int numero = 1;
    System.out.println( numero );
    numero = numero+1;
}
System.out.println( numero );
```

- ▶ Genera errore di compilazione
- ▶ **Ambito di visibilità**: zona di programma in cui si può usare variabile
 - ▶ Due variabili possono avere stesso nome se hanno diverso ambito di visibilità

```
void stampaDoppio( int n ) {
    int numero = 2*n;
    System.out.println( "Il doppio e' "+numero );
}
int numero = 3;
System.out.println( "Il numero e' "+numero );
stampaDoppio( numero );
System.out.println( "Il numero e' "+numero );
```

Variabili locali

```
int f1( int b ) {
    int a = b+1;
    return a+1;
}
int f2( int a ) {
    int b = a-2;
    return b-2;
}
int b = 10, a = 10;
System.out.println( "a: "+a+", b: "+b );
b = f1( a );
a = f2( b );
System.out.println( "a: "+a+", b: "+b );
```

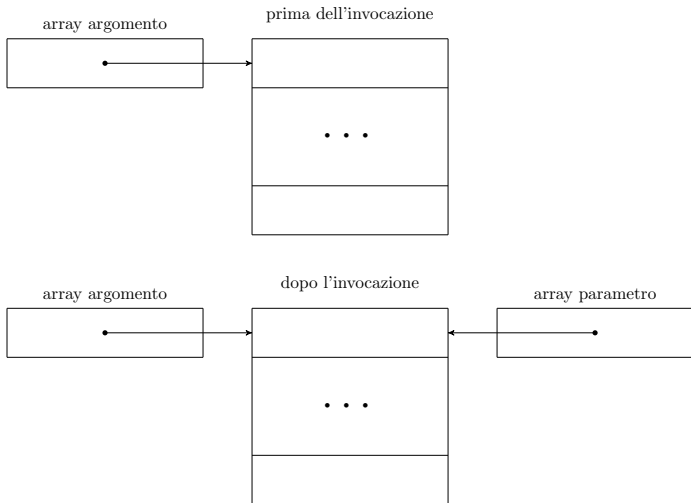
▶ Regole di buon senso

- ▶ Più semplice dichiarare le variabili fuori del blocco
- ▶ Non dichiarare variabili all'interno di cicli
 - ▶ Eccezione per inizializzazione di ciclo `for`

Uguaglianza di array

- ▶ Uguali se stessa lunghezza e stessi elementi

```
boolean arrayUguali( int[] a,int[] b ) {  
    if (a.length!=b.length) {  
        return false;  
    }  
    for (int i = 0; i<a.length; i++) {  
        if (a[i]!=b[i]) {  
            return false;  
        }  
    }  
    return true;  
}
```

- ▶ Operando su array parametro formale, si opera su array argomento

Array come parametri formali

```
void raddoppiaArray( int[] numero ) {
    for ( int i = 0; i<numero.length; i++ ) {
        numero[i] *= 2;
    }
}

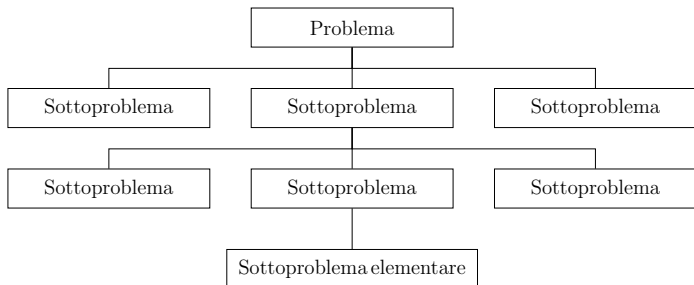
void stampaArray( int[] numero ) {
    System.out.print( "Array: " );
    for (int i = 0; i<numero.length; i++) {
        System.out.print( numero[i]+" " );
    }
    System.out.println( );
}
```

Ordinamento di array

- ▶ Selection sort
 - ▶ Molto semplice e facile da capire
- ▶ Esegue le seguenti operazioni
 - ▶ Cerca nell'array l'elemento con il valore più piccolo
 - ▶ Lo scambia con il primo elemento
 - ▶ cerca tra gli elementi dell'array, escluso il primo, l'elemento con il valore più piccolo
 - ▶ Lo scambia con il secondo elemento
 - ▶ Ripete fino a che tutti gli elementi sono al posto giusto


```
void selectionSort( int[] a ) {
    int i,j,indiceProssimoMinimo,min,temp;
    for (i = 0; i<a.length-1; i++) {
        min = a[i];
        indiceProssimoMinimo = i;
        for (j = i+1; j<a.length; j++) {
            if (a[j]<min) {
                min = a[j];
                indiceProssimoMinimo = j;
            }
        }
        temp = a[i];
        a[i] = a[indiceProssimoMinimo];
        a[indiceProssimoMinimo] = temp;
    }
}
```

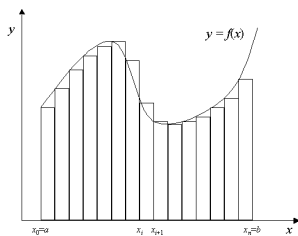
- ▶ Scomporre problema in sotto-problemi più semplici
 - ▶ Fino a raggiungere sotto-problemi elementari



- ▶ Sotto-problemi il più possibile autonomi e indipendenti
 - ▶ Affrontabili in modo indipendente dagli altri
 - ▶ Riutilizzabili in altri contesti
- ▶ A ogni sotto-problema corrisponde procedura risolutiva

- ▶ In Java, metodi corrispondono a procedure
 - ▶ Interfaccia del metodo specifica come procedura va usata
 - ▶ Indipendenza di procedura implica che solo variabili locali siano modificate
- ▶ Indipendenza non implica nessuna relazione
 - ▶ Spesso, una procedura usa altre procedure
 - ▶ La prima non può essere usata fin quando le seconde non sono implementate
 - ▶ Sviluppo può proseguire in parallelo

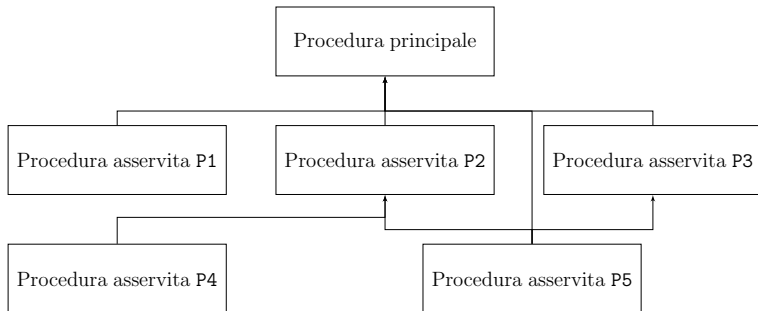
Integrale di una funzione



```
double integrale( double a, double b, int n ) {  
    double h = (b-a)/n;  
    double area = 0.0;  
    for (int i = 0; i < n; i++) {  
        area = area + h * f( a + i * h );  
    }  
    return area;  
}  
  
double f( double x ) {  
    return 4 / (1 + x * x);  
}
```


- ▶ Pre-condizioni e post-condizioni
 - ▶ Contratto firmato tra chi sviluppa procedura e chi usa procedura
- ▶ Pre-condizione
 - ▶ Requisito che deve essere soddisfatto da chi invoca il metodo
 - ▶ Metodo non responsabile se invocato in violazione di pre-condizioni
- ▶ Post-condizione
 - ▶ Valore restituito calcolato correttamente
 - ▶ Sistema in un determinato stato dopo esecuzione di metodo
 - ▶ Metodo responsabile di effetti collaterali

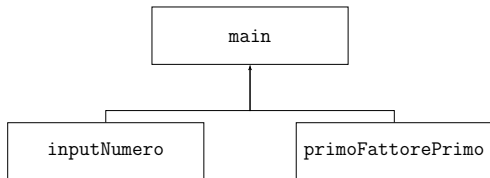
- ▶ Schema di programma basato su programmazione procedurale



- ▶ Stessa procedura asservita a più procedure
- ▶ In Java, procedura principale chiamata `main`

Fattori primi

1. Richiedere all'utente un numero intero compreso tra 2 e 1000
 2. Se il numero inserito dall'utente è primo, stampare nuovamente il numero (in quanto unico fattore primo di se stesso)
 3. Altrimenti calcolare il più piccolo fattore primo, stampare tale fattore e aggiornare il numero dividendolo per il fattore calcolato, ripetendo questa operazione fino a quando il numero non diventa minore di 2
- Tre procedure



▶ Procedura di input

```
int inputNumero( int l,int r ) {  
    int n = Input.getInt( "Numero (" + l + "-" + r + ")" );  
    while ((n < l) || (n > r)) {  
        n = Input.getInt( "Numero (" + l + "-" + r + ")" );  
    }  
    return n;  
}
```

▶ Pre-condizione

- ▶ Argomenti: numeri interi positivi con l non maggiore di r

▶ Post-condizione

- ▶ Valore di ritorno: numero incluso in $[l, r]$

▶ Procedura di ricerca fattore primo

```
int primoFattorePrimo( int n ) {  
    for (int i = 2; i<=n/2; i++ ) {  
        if (n%i==0) {  
            return i;  
        }  
    }  
    return n;  
}
```

▶ Pre-condizione

- ▶ Argomento: numero interi positivi

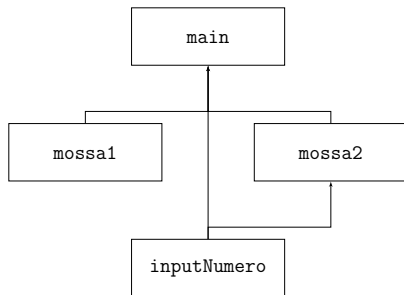
▶ Post-condizione

- ▶ Valore di ritorno: primo fattore primo di n

► Procedura di ricerca fattore primo

```
void main( ) {  
    int numero = inputNumero( 2, 1000 );  
    int fattore = 0;  
    System.out.println( "Numero: "+numero );  
    if ( primoFattorePrimo( numero ) == numero) {  
        System.out.println( "Fattore: "+numero );  
    } else {  
        while (numero>1) {  
            fattore = primoFattorePrimo( numero );  
            System.out.println( "Fattore: "+fattore );  
            numero = numero/fattore;  
        }  
    }  
}
```

Il gioco della corsa della pedina



- ▶ `mossa1`: giocatore digitale
- ▶ `mossa2`: giocatore umano
- ▶ `inputNumero`: per chiedere primo giocatore e mossa umana

```
int mossa1(int p) {  
    int mossa = 1;  
    switch (p) {  
        case 0:  
            case 3: mossa = 2;  
    }  
    return mossa;  
}
```



```
int inputNumero( int l,int r, int p ) {  
    int n = Input.getInt( "("+l+"-"+r+", "+p+" )" );  
    while ((n<l)|| (n>r)) {  
        n = Input.getInt( "("+l+"-"+r+", "+p+" )" );  
    }  
    return n;  
}
```

```
int mosca2(int p) {  
    return inputNumero(1,2,p);  
}
```

```
void main() {
    int p = 0;
    int vincitore = 0;
    int turno = inputNumero(1,2,-1);
    int mossa = 0;
    while (vincitore==0) {
        if (turno==1) {
            mossa = mossa1(p);
        } else {
            mossa = mossa2(p);
        }
        p = p+mossa;
        if (p==5) {
            vincitore=turno;
        }
        turno = 3-turno;
    }
    System.out.println("Vincitore: "+vincitore);
}
```

- ▶ Strumento potente per manipolazione di strutture matematiche
- ▶ Consente semplice formulazione di algoritmi
- ▶ Consente semplice dimostrazione di correttezza di algoritmi
- ▶ Metodi ricorsivi
 - ▶ Direttamente ricorsivi
 - ▶ Invocano sè stessi
 - ▶ Indirettamente ricorsivi
 - ▶ Invoca un altro metodo che invoca (direttamente o indirettamente) primo metodo

- ▶ Metodo direttamente ricorsivo per cercare numero intero in array ordinato

```
boolean esiste(int[] a, int l, int r, int k) {  
    if (l>r) {  
        return false;  
    } else {  
        int m = (l+r)/2;  
        if (a[m]==k) {  
            return true;  
        } else {  
            if (k<a[m]) {  
                return esiste(a,l,m-1,k);  
            } else {  
                return esiste(a,m+1,r,k);  
            }  
        }  
    }  
}
```

► Metodi indirettamente ricorsivi

```
void stampaNumeroDispari( int n ) {
    System.out.println( "Numero dispari: "+n );
    if (n>1) {
        stampaNumeroPari( n-1 );
    }
}

void stampaNumeroPari( int n ) {
    System.out.println( "Numero pari: "+n );
    stampaNumeroDispari( n-1 );
}

void stampa( int n ) {
    if (n%2==0) {
        stampaNumeroPari( n );
    } else {
        stampaNumeroDispari( n );
    }
}
```

Metodi ricorsivi e numerologia

- ▶ Numero del destino
 - ▶ Somma cifre data di nascita ripetuta sino a quando una cifra
- ▶ Esempio: 21 Giugno 1961
 - ▶ 21061961
 - ▶ $2+1+0+6+1+9+6+1=26$
 - ▶ $2+6=8$
 - ▶ Numero del destino: 8
 - ▶ Persona “nata per guidare e a cui il destino fornirà le opportunità per esprimere i propri talenti nativi”

```
int sommaCifre( int numero ) {
    int risultato = 0;
    while (numero>0) {
        risultato += numero%10;
        numero = numero/10;
    }
    return risultato;
}
int destino( int numero ) {
    int temp = sommaCifre( numero );
    if (numero>9)
        return destino( temp );
    else
        return temp;
}
void main() {
    int n = Input.getInt( "Data (GGMMAAAA)" );
    System.out.println( "Destino: "+destino( n ) );
}
```


- ▶ In alcuni casi, è facile trasformare metodo ricorsivo in metodo non ricorsivo
 - ▶ Schema $A = \text{if}(E)\{S; A\}$
 - ▶ Diventa $A = \text{while}(E)\{S\}$
 - ▶ Schema $A = S; \text{if}(E)\{A\}$
 - ▶ Diventa $A = \text{do}\{S\}\text{while}(E);$
- ▶ Facendo uso di pila, ogni metodo ricorsivo può essere reso non ricorsivo

Ricorsione e fattoriale di un numero

- ▶ Metodo ricorsivo

```
int fattorialeRicorsivo( int n ) {  
    if (n>1) {  
        return n*fattorialeRicorsivo( n-1 );  
    } else {  
        return 1;  
    }  
}
```

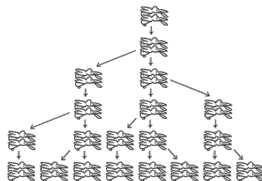
- ▶ Metodo non ricorsivo

- ▶ Già visto: fa uso di for

Numeri di Fibonacci

- ▶ Divertissement matematico
 - ▶ Crescita popolazione conigli
 - ▶ Una coppia all'inizio
 - ▶ Ogni coppia diviene fertile dopo 1 mese
 - ▶ Ogni coppia fertile riproduce una nuova coppia ogni mese
 - ▶ Quante coppie di conigli dopo n mesi?

Mese 0	1
Mese 1	1
Mese 2	2
Mese 3	3
Mese 4	5
Mese 5	8

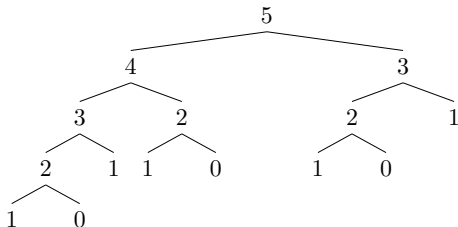


- ▶ $F_n = F_{n-1} + F_{n-2}$
 - ▶ F_{n-1} : coppie già esistenti
 - ▶ F_{n-2} : coppie nuove

► Metodo ricorsivo

```
long fiboRic( int n ) {  
    if (n<2) {  
        return n;  
    } else {  
        return fiboRic( n-1 )+fiboRic( n-2 );  
    }  
}
```

► Albero delle chiamate ricorsive



▶ Metodo iterativo

```
long fibonacc( int n ) {  
    int i = 1;  
    long x = 1, y = 0;  
    while (i<n) {  
        i = i+1;  
        x = x+y;  
        y = x-y;  
    }  
    return x;  
}
```

- ▶ Costo lineare nell'indice che si vuole calcolare
- ▶ Ricorsione da evitare se soluzione iterativa ovvia
 - ▶ Confinata a casi in cui rende codice più facile da capire

Le torri di Hanoi

- ▶ 3 pioli e $n = 64$ dischi sul primo piolo (vuoti gli altri due)
- ▶ Ogni mossa sposta un disco in cima a un piolo
- ▶ Un disco non può poggiare su uno più piccolo
- ▶ Spostare tutti i dischi dal primo al terzo piolo
 - ▶ Leggenda: finito lo spostamento, il mondo scomparirà
- ▶ Metodo ricorsivo

```
void risolviHanoi( int n,int s,int d ) {  
    if (n>0) {  
        risolviHanoi( n-1,s,6-s-d );  
        System.out.println( "Da "+s+" a "+d );  
        risolviHanoi( n-1,6-s-d,d );  
    }  
}
```

- ▶ Termina
 - ▶ A ogni nuova esecuzione il numero di dischi è diminuito di 1
- ▶ Quanti passi esegue?

Numero di mosse

- ▶ Se $n = 1$: 1
- ▶ Se $n = 2$: 3
- ▶ Se $n = 3$: 7
- ▶ Se $n = 4$: 15
- ▶ Se $n = 5$: 31
- ▶ In generale: $2^n - 1$
- ▶ Dimostrazione
 - ▶ Caso base $n = 1$: $2^1 - 1 = 1$
 - ▶ Passo induttivo: $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$
- ▶ 1 mossa/sec: circa 585 miliardi di anni!

Generazione di sequenze binarie e di permutazioni

- ▶ Sequenza binaria equivalente a sottoinsieme di n elementi
- ▶ Metodo ricorsivo
 - ▶ Pone bit in posizione b a 0
 - ▶ Genera tutte le sequenze con tale bit (ricorsivamente)
 - ▶ Pone bit in posizione b a 1
 - ▶ Genera tutte le sequenze con tale bit (ricorsivamente)

```
void generaBinarie( int [] a, int b ) {  
    if ( b==0 ) {  
        stampaArray( a );  
    } else {  
        a[b-1] = 0;  
        generaBinarie( a, b-1 );  
        a[b-1] = 1;  
        generaBinarie( a, b-1 );  
    }  
}
```


- ▶ Generazione ricorsiva di permutazioni
 - ▶ Ciascuno elemento occupa ultima posizione
 - ▶ Rimanenti elementi ricorsivamente permutati

```
void generaPermutazioni(char[] a, int p) {  
    char temp;  
    if (p==0) {  
        stampaArray( a );  
    } else {  
        for (int i = p-1; i>=0; i--) {  
            temp = a[i];  
            a[i] = a[p-1];  
            a[p-1] = temp;  
            generaPermutazioni( a, p-1 );  
            temp = a[i];  
            a[i] = a[p-1];  
            a[p-1] = temp;  
        }  
    }  
}
```

Ricorsione e divide-et-impera

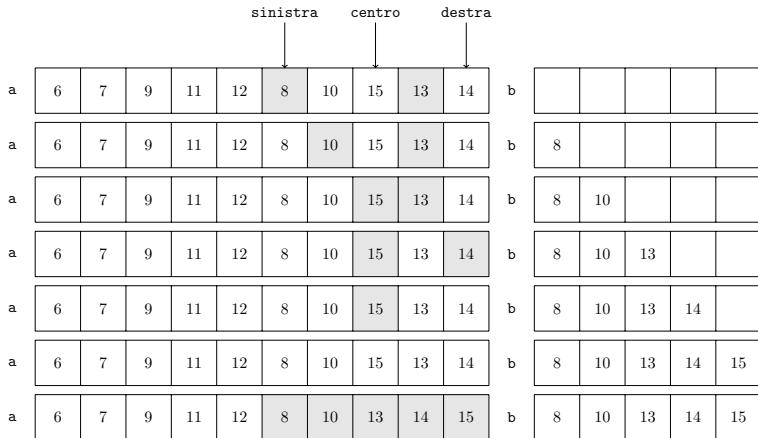
► Ordinamento per fusione

1. Decomposizione in sotto-problemi: se almeno due elementi, divide in due sotto-array uguali
2. Soluzione ricorsiva: ordina ricorsivamente due sotto-array
3. Ricombinazione delle soluzioni: fonde due sotto-array ordinati in un array ordinato

► Implementazione

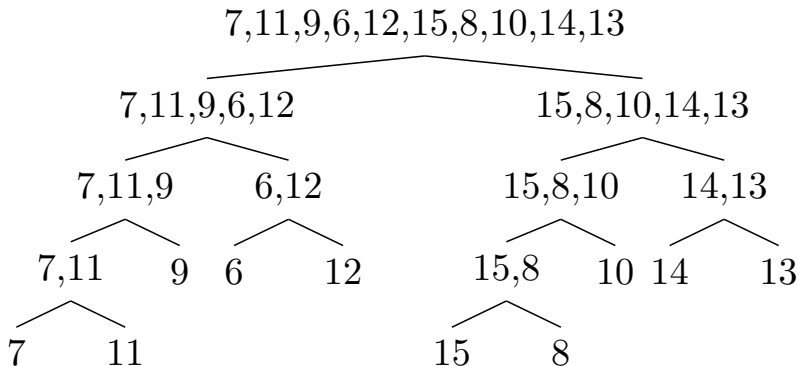
```
void ordina( int[] a, int sinistra, int destra) {  
    if (sinistra < destra) {  
        int centro = (sinistra + destra) / 2;  
        ordina(a, sinistra, centro);  
        ordina(a, centro + 1, destra);  
        fondi(a, sinistra, centro, destra);  
    }  
}
```

► Fusione con array di appoggio



```
void fondi(int[] a, int s, int c, int d) {
    int[] b = new int[d-s+1];
    int i = s, j = c + 1, k = 0;
    while ((i <= c) && (j <= d)) {
        if (a[i]<a[j]) {
            b[k] = a[i]; i = i + 1; k = k + 1;
        } else {
            b[k] = a[j]; j = j + 1; k = k + 1;
        }
    }
    while (i <= c) {
        b[k] = a[i]; i = i + 1; k = k + 1;
    }
    while (j <= d) {
        b[k] = a[j]; j = j + 1; k = k + 1;
    }
    for (i = s; i <= d; i = i + 1) a[i] = b[i-s];
}
```

▶ Albero delle chiamate ricorsive



- ▶ Tabella 9×9 contenente numeri compresi tra 1 e 9
- ▶ Divisa in 3×3 sottotabelle (di taglia 3×3)
- ▶ Alcune celle contengono numeri, altre vuote
- ▶ Riempire le celle vuote in modo che
 - ▶ Ogni riga contiene una permutazione di $1, 2, \dots, 9$
 - ▶ Ogni colonna contiene una permutazione di $1, 2, \dots, 9$
 - ▶ Ogni sottotabella contiene una permutazione di $1, 2, \dots, 9$

▶ Esempio

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

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

- ▶ Soluzione ottenibile attraverso implicazioni logiche
- ▶ Ad esempio, nella sottotabella in alto a destra il 3 può stare solo in basso a sinistra

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

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

- ▶ Partendo dalla configurazione a sinistra, giungiamo nella configurazione a destra che ammette diverse scelte per ogni casella
- ▶ **Backtrack**: *algoritmo che esplora tali scelte, annullando gli effetti nel caso che non conducano a soluzione*

► Algoritmo di backtrack

1. Cerca prima casella vuota c
2. Determina cifre possibili per c : prova ad assegnare a c una dopo l'altra tali cifre
3. Se ultima casella, restituisce configurazione finale (soluzione trovata)
4. Altrimenti, procede ricorsivamente con prossima casella vuota
 - Se non produce soluzione accettabile, annulla scelta fatta e prova successiva (procedimento termina se finite le scelte possibili)

► Prossima cella vuota

- Cella identificata da array di sue coordinate

```
int[] prossimaCasella( int[] c, int[][] s ) {
    for (int j = c[1] + 1; j < s.length; j++) {
        if (s[c[0]][j] == 0) {
            return new int[] { c[0], j };
        }
    }
    for (int i = c[0] + 1; i < s.length; i++) {
        for (int j = 0; j < s.length; j++) {
            if (s[i][j] == 0) {
                return new int[] { i, j };
            }
        }
    }
    return new int[] { -1, -1 };
}
```

- ▶ Decidere se cifra compatibile con cella
 - ▶ n : radice quadrata dimensione tabella

```
boolean accettabile(int d,int [] c,int n,int [][] s){
    for (int i = 0; i < n * n; i++)
        if (s[c[0]][i] == d) return false;
    for (int i = 0; i < n * n; i++)
        if (s[i][c[1]] == d) return false;
    int fr = (c[0] / n) * n;
    int fc = (c[1] / n) * n;
    for (int i = fr; i < fr + n; i++) {
        for (int j = fc; j < fc + n; j++) {
            if (s[i][j] == d) {
                return false;
            }
        }
    }
    return true;
}
```

- ▶ Determinare cifre posizionabili in cella
 - ▶ Usa accettabile

```
boolean[] cifreAmmissibili(int[] c,int n,int[][] s){
    boolean[] r = new boolean[n * n];
    for (int d = 1; d <= n * n; d++) {
        r[d - 1] = accettabile(d, c, n, s);
    }
    return r;
}
```

► Risolvere il Sudoku

► Ricorsivo

► Usa cifreAmmissibili e prossimaCasella

```
boolean risolubile(int[] c, int n, int[][] s) {
    boolean[] a = cifreAmmissibili(c, n, s);
    for (int d = 1; d <= n*n; d++) {
        if (a[d-1]) {
            s[c[0]][c[1]] = d;
            int[] nc = prossimaCasella(c, s);
            if (nc[0] >= 0 && !risolubile(nc, n, s)) {
                s[c[0]][c[1]] = 0;
            } else {
                return true;
            }
        }
    }
    return false;
}
```

► Il metodo principale

```
void main() {
    int[][] s = { { 4, 2, 0, 0 }, { 0, 0, 0, 3 },
                 { 0, 1, 3, 0 }, { 0, 0, 0, 2 } };
    int n = 2;
    stampaMatrice( s );
    int[] p = prossimaCasella(new int[] { 0, -1 }, s);
    System.out.println(risolubile(p, n, s));
    stampaMatrice( s );
}

void stampaMatrice(int[][] s) {
    for (int i = 0; i < s.length; i++) {
        for (int j = 0; j < s.length; j++) {
            System.out.print(s[i][j] + " ");
        }
        System.out.println();
    }
}
```

► Schema del programma

