



# Complessità

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Complessita'



- L'analisi di complessita' definisce le risorse teoricamente consumate da un algoritmo
  - **complessità** **temporale:** tempo necessario all'esecuzione dell'algoritmo
  - **complessità** **spaziale:** memoria necessaria all'esecuzione dell'algoritmo
- Poiché ad ogni algoritmo corrispondono più implementazioni (più programmi), lo studio della complessità non definisce esattamente il tempo e la memoria usata, si concentra sulle proprietà che sono indipendenti dell'implementazione fornendo un'idea di quanto sia efficiente un algoritmo

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Cosa si misura?

## Complessità temporale

- Si contano le istruzioni eseguite dall'algoritmo
- Poiché le istruzioni potrebbero essere di natura diversa spesso si cerca di individuare quelle che incidono principalmente sul tempo di esecuzione
  - Le operazioni in virgola mobile: le più lente da eseguire per una CPU; sono predominanti se il loro numero è paragonabile al numero delle altre istruzioni.
  - Gli if: le istruzioni più frequenti; sono predominanti se sono in numero molto superiore alle altre istruzioni

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Cosa si misura?

- Le istruzioni di accesso alla memoria secondaria e alle periferiche: sono decine di migliaia di volte più lente delle istruzioni svolte nella memoria principale; se un'applicazione ne richiede alcune, queste potrebbero essere predominanti. Ad esempio, nei database, l'analisi di complessità è concentrata solo sugli accessi al disco.

## Complessità spaziale

- Si misurano le celle di memoria occupate
- La complessità si concentrerà sulle celle della memoria principale, se si pensa che i dati l'algoritmo possano essere allocati in memoria principale o su quelle della memoria secondaria in caso opposto

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Complessità asintotica

- Lo studio della complessità si concentra su i casi in cui il problema è grande:
  - non importa se un programma di contabilità impiega 1 o 100 milisecondi secondi a calcolare il bilancio
  - cambia molto se il programma della segreteria impiega 1 o 10 secondi a trovare i dati di uno studente nell'archivio
- **La complessità asintotica**
  - definisce le risorse usate da un algoritmo al crescere della dimensione del problema affrontato
  - Ad esempio, come cambia il tempo di accesso ai dati quando cresce il numero degli studenti nell'archivio della segreteria...

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Complessità temporale asintotica

Formalmente si usa il concetto matematico di ordine

- sia  $n$  è la **dimensione** del problema, cioè la dimensione dell'input dell'algoritmo
- sia  $T(n)$  è il **tempo** impiegato dall'algoritmo quando l'ingresso ha dimensione  $n$
- sia  $f(n)$  è una qualsiasi funzione di  $n$ , ad esempio  $3, n, n^2, n^5, 2^n$
- Si dice che la complessità asintotica dell'algoritmo **è ordine di  $f(n)$**  e si scrive  **$O(f(n))$**  se esiste una costante  $\alpha$  tale che  $T(n) < \alpha f(n)$

Osservazione importante

- Si osservi che con questa definizione algoritmi che differiscono solo per una costante moltiplicativa appartengono allo stesso ordine
- Esempio: due algoritmi che richiedono  $4*n$  e  $7*n$  sono entrambi  $O(n)$



# Complessità temporale asintotica

## Informalmente

- L'ordine  $O(f(n))$  fornisce una misura della complessità temporale di ogni programma che implementa l'algoritmo

## Esempio

- Calcolare la somma degli elementi di un array
  - $n$  = numero di elementi dell'array
  - complessità:  $O(n)$

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Complessità temporale asintotica: array

## Array

- $n$  = numero delle posizioni dell'array
- Ricerca/inserimento/cancellazione di un elemento in un array nel caso dell'allocazione per chiave (conosco la posizione in cui si trova)
  - complessità  $O(1)$
- Ricerca/inserimento/cancellazione di un elemento in un array nel caso dell'allocazione sequenziale (conosco parte del contenuto, ad esempio il nome dello studente)
  - complessità  $O(n)$

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Complessità media e nel caso peggiore

- Un algoritmo può richiedere un numero di operazioni diverse per ingressi di dimensione uguale:
  - **complessità media**  
la complessità media su tutti i possibili ingressi
  - **complessità nel caso peggiore**  
la complessità dell'algoritmo per l'ingresso che richiede più operazioni

- Nel caso dell'array le due complessità coincidono
- Di solito, quando si parla di complessità ci si riferisce al caso nel caso peggiore

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Complessità temporale asintotica: liste

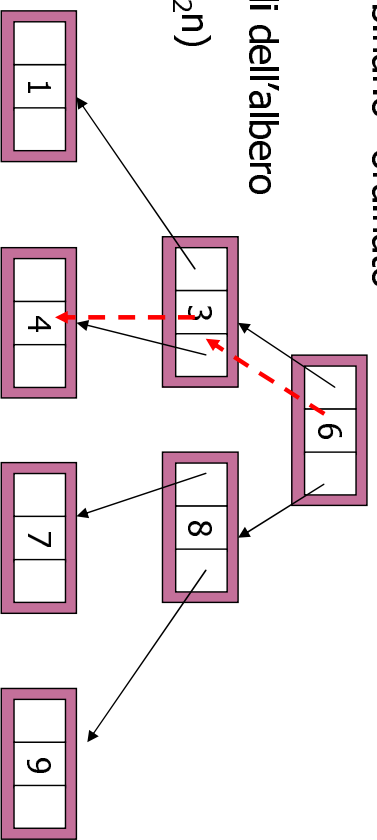
## Liste

- $n$  = numero delle posizioni della lista
- Ricerca/cancellazione di un elemento
  - **complessità  $O(n)$**
- Inserimento di un elemento
  - **complessità  $O(1)$**

# Complessità temporale asintotica: alberi

## Alberi

- Inserire, eliminare o cercare un elemento in albero binario ordinato bilanciato
  - $n$  = numero dei nodi dell'albero
  - complessità:  $O(\log_2 n)$



Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Complessità temporale asintotica: un altro esempio

## Un problema con complessità fattoriale

- Dato un programma che prende in ingresso i partecipanti ad una competizione e genera (ad esempio per stamparle) tutte le possibili classifiche finali
  - $n$  = numero di partecipanti
  - complessità:  $O(n!)$
- Si osservi che  $n!$  è un numero molto grande anche per  $n$  relativamente piccoli
  - $20! = 2.432.902.008.176.640.000 = 2.4 \cdot 10^{18}$

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Complessità temporale asintotica

## Un problema con complessità polinomiale

- Un programma che prende in ingresso due vettori  $a$ ,  $b$  e stampa tutte le coppie  $(i,j)$  tali che  $a[i]=b[j]$
- $n$ = dimensione di  $a$ ,  $m$ = dimensione di  $b$
- complessità:  $O(n*m)$

```
void stampa(int a[], int b[]){  
    for(int i=0;i<a.length;i++){  
        for(int j=0;j<b.length;j++){  
            if(a[i]==b[j]){  
                System.out.println("trovata corrispondenza fra a"+ i +" e "+ j );  
            }  
        }  
    }  
}
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Algoritmi facili e difficili ...

- A seconda della loro complessità temporale asintotica, gli algoritmi sono tipicamente divisi in classi
- algoritmi a **complessità costante**  $O(1)$  o **lineare**  $O(n)$ 
  - molto veloci
- algoritmi a **complessità polinomiale**  $O(n^3)$  per un qualche valore  $a$ 
  - usabili se l'esponente  $a$  è piccolo
- algoritmi a **complessità esponenziale**  $O(a^n)$  per un qualche valore  $a$ 
  - usabili solo per  $n$  molto piccoli



## Algoritmi facili e difficili ...

- La suddivisione fra problemi facili e difficili descritta precedentemente è quella di base, ovviamente, però la difficoltà di un problema dipende anche dalla dimensione dei dati !
- Ad esempio, sei dati sono una copia del Web (molti TeraByte), anche un algoritmo che sia  $O(n)$  può risultare lento

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

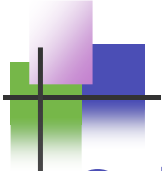


## Algoritmi con complessità esponenziale

- **Fondamentale:** perché gli algoritmi a complessità esponenziale sono considerati quasi inusabili ?
  - Perché richiedono talmente tante operazioni che probabilmente anche i calcolatori futuri non potranno risolverli per ingressi molto grandi!!

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Algoritmi con complessità esponenziale: un esempio

- Si consideri il programma che genera tutte le classifiche finali di una competizione con  $n$  partecipanti, complessità  $O(n!)$  (è esponenziale, perché  $n! \approx n^{\sqrt{n}}$ )
- Con 20 concorrenti le classifiche sono  $20! \approx 2.4 \cdot 10^{18}$ . Un computer che generi 1 miliardo di classifiche al secondo, circa  $3 \cdot 10^{15}$  l'anno, impiegherebbe circa 79 anni per generare tutte le classifiche richieste.
- Ovviamente, i computer diventeranno sempre più veloci e fra dieci anni magari saranno abbastanza veloci da fare in un mese quello per cui adesso occorrono 79 anni ma...
- comunque, fra dieci anni per risolvere il problema con 21 partecipanti occorreranno ancora 21 mesi e per 25 partecipanti circa 265650 anni!!

Franco Scarselli

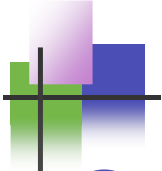
Fondamenti di Informatica I, 2006-2007



# Algoritmi importanti

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Ordinamento

---

- Il problema dell'ordinamento: dato un vettore di interi (o qualsiasi altra cosa su cui sia definito un ordine) ordinarli dal minore al maggiore
- Per questo problema esistono numerosi algoritmi: bubble sort, merge sort, quick sort, selection sort...

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Il bubble sort (ordinamento a bolla)

---

- Il vettore  $A$  viene sottoposto a più passaggi
- Un passaggio consiste in una scansione di tutto il vettore durante la quale ogni elemento  $A[i]$  viene confrontato con quello successivo  $A[i+1]$ : i due elementi vengono scambiati se non si trovano nell'ordine desiderato
- I passaggi vengono ripetuti fino quando ci sono elementi da scambiare

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

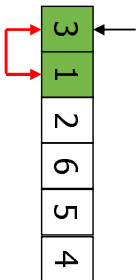
# Bubble sort: esempio

i=2

## Primo passaggio:

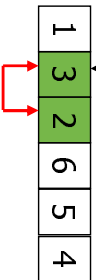
l'indice i viene inizializzato a 0

i=0



Si confronta A[0] con A[1]: si scambiano

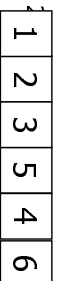
i=1



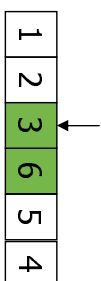
Si confronta A[1] con A[2]: si scambiano

Franco Scarselli

Fondamenti di Informatica I,

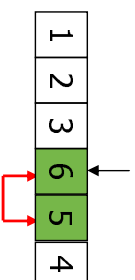


Si confronta A[4] con A[5]: si scambiano



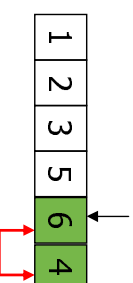
Si confronta A[2] con A[3]: non si fa niente

i=3



Si confronta A[3] con A[4]: si scambiano

i=4



# Bubble sort: esempio

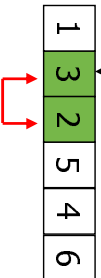
## Secondo passaggio:

l'indice i viene inizializzato a 0 di nuovo

i=0



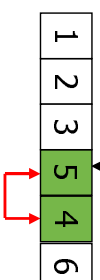
i=1



i=2



i=3



i=4

Nota che non è necessario fare l'ultimo confronto: A[5] con A[6].  
Di fatti per come funziona l'algoritmo A[6] contiene sicuramente il valore massimo.

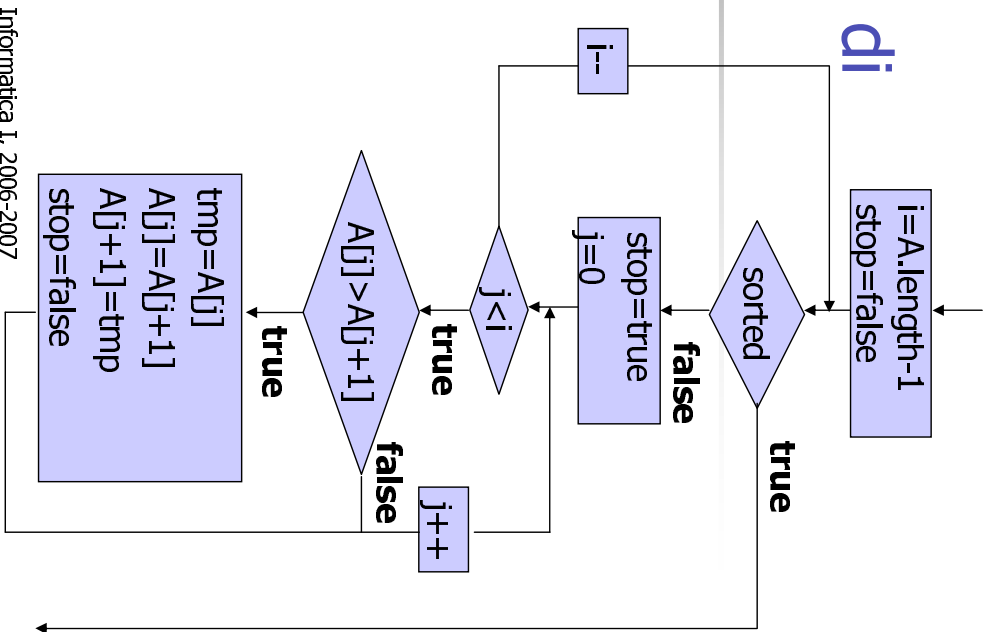
Al terzo passaggio non ci saranno scambi: l'algoritmo termina

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Implementazione di bubble sort

```
public void sort(int A[]) {
    int i = A.length-1;
    boolean stop=false;
    while(!stop){
        stop=true;
        for(int j=0;j<i;j++){
            if(A[j]>A[j+1]){
                int tmp=A[j];
                A[j]=A[j+1];
                A[j+1]=tmp;
                stop=false;
            }
        }
        i--;
    }
}
```



Informatica I, 2006-2007

## Complessità del bubble sort

### Complessità (caso peggiore)

- Confronti per passaggio
  - Al primo passaggio si fanno al più  $n-1$  confronti
  - Al secondo passaggio si fanno al più  $n-2$  confronti
  - Al terzo passaggio si fanno al più  $n-3$  confronti
- Servono al più  $n-1$  passaggi
- Quindi si fanno al più  $\sum_{i=1}^{n-1} n-i = \frac{(n-1)(n-2)}{2}$  confronti
- Bubble sort è  $O(n^2)$
- Il **caso peggiore** si verifica quando il valore minimo si trova nella cella più a destra: occorrono  $O(n^2)$  confronti
- Il **caso migliore** si verifica quando gli elementi sono già ordinati: occorrono  $O(n)$  confronti



# Ricorsione

- Un algoritmo ricorsivo è un algoritmo definito in termini di se stesso
- E composto da
  - La definizione della base della ricorsione
  - La definizione delle regole di ricorsione
- **Esempio: il fattoriale**
  - Base: il fattoriale di 1 è 1
  - Regola di ricorsione: Il fattoriale di  $n$  è  $n$  moltiplicato il fattoriale di  $n-1$
- Sebbene il calcolo del fattoriale può essere implementato con un normale ciclo for, esso può essere implementato anche in modo naturale con un programma ricorsivo che segue esattamente la definizione

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Implementazione del fattoriale

### Implementazione iterativa

```
public int factorial(int val){  
    int res=1;  
    for(int i=1;i<=val;i++){  
        res=res*i;  
    }  
    return res;  
}
```

### Implementazione ricorsiva

```
public int factorial(int val){  
    if(val==1) {  
        return 1;}  
    else {  
        return factorial(val-1)*val;  
    }  
}
```

# Successione di Fibonacci

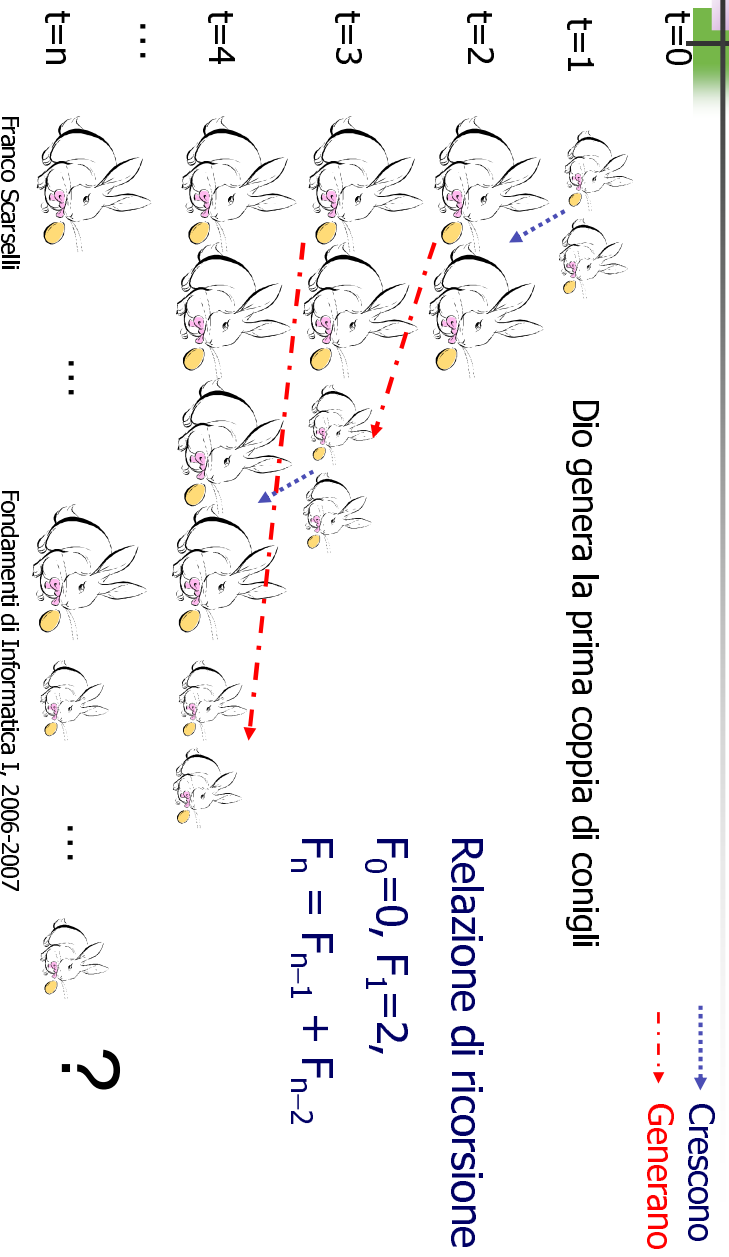
## Successione di Fibonacci

- Leonardo Pisano, detto Fibonacci, pose il seguente quesito:
  - Una coppia di conigli giovani impiega una unità di tempo a diventare adulta; una coppia adulta impiega una unità di tempo a riprodursi e generare un'altra coppia di conigli (chiaramente giovani); i conigli non muoiono mai
  - Quante coppie di conigli abbiamo al tempo  $t$  generico se al tempo  $t=0$  non abbiamo conigli e al tempo  $t=1$  abbiamo una coppia di giovani conigli?

$$F_0=0, F_1=2,$$

$$F_n = F_{n-1} + F_{n-2}$$

## Successione di Fibonacci





# Implementazione della serie di Fibonacci

```
public int fibonacci(int val){  
    if(val==0) {  
        return 0;}  
    else {  
        if(val==1) {return 2;}  
        else {  
            return fibonacci(val -1) + fibonacci(val-2);}  
        }  
    }  
}
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Ricerca dicotomica

- Per cercare un elemento in un vettore **ordinato** esiste un metodo detto ricerca dicotomica
- Si confronta il valore **val** da ricercare con l'elemento centrale del vettore `A[A.length/2]`
- Se il **val** è minore si ripete la ricerca sulla metà sinistra, altrimenti si ricerca nella metà destra

# Ricerca dicotomica:

Esempio: si vuol cercare 23

Si confronta 23 con 13

|   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 5 | 8 | 9 | 13 | 16 | 20 | 23 | 27 | 30 | 34 | 35 |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

Ci si concentra sulla metà destra (da ind. 8 a ind. 14) : si confronta 23 con 27

|   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 5 | 8 | 9 | 13 | 16 | 20 | 23 | 27 | 30 | 34 | 35 |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

Ci si concentra sulla metà sinistra (da ind. 8 a ind. 10): si confronta 23 con 20

|   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 5 | 8 | 9 | 13 | 16 | 20 | 23 | 27 | 30 | 34 | 35 |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

Ci si concentra sulla metà destra (da ind. 9 a ind. 9): trovato!!

|   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 0 | 2 | 4 | 5 | 8 | 9 | 13 | 16 | 20 | 23 | 27 | 30 | 34 | 35 |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Implementazione della ricerca dicotomica

```
public int search(int val, int A[], int from, int to){
    if (from < to){
        int center=(to-from)/2;
        if (val < A[center]){ return search(val,A, from,center-1);}
        if (val > A[center]){ return search(val,A, center+1,to);}
        if (val == A[center]) { return center;}}
    else{
        If (from == to) {
            if (A[from] == val) {return from;}
            else return -1;}
        //si esegue solo se (from > to)
        else { return -1;}}
    }
```

L'operatore di divisione / quando applicato agli interi tronca la parte frazionaria

-1 vuol dire non trovato



# Complessità della ricerca dicotomica

- La ricerca dicotomica divide il vettore in due ad ogni passo:
  - dopo p passi la dimensione del vettore è  $\frac{n}{2^p}$
  - la ricerca si ferma quando  $\frac{n}{2^p}$  è 1, cioè quando  $p = \log_2 n$
- Quindi la ricerca dicotomica è  $O(\log_2 n)$

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Mergesort

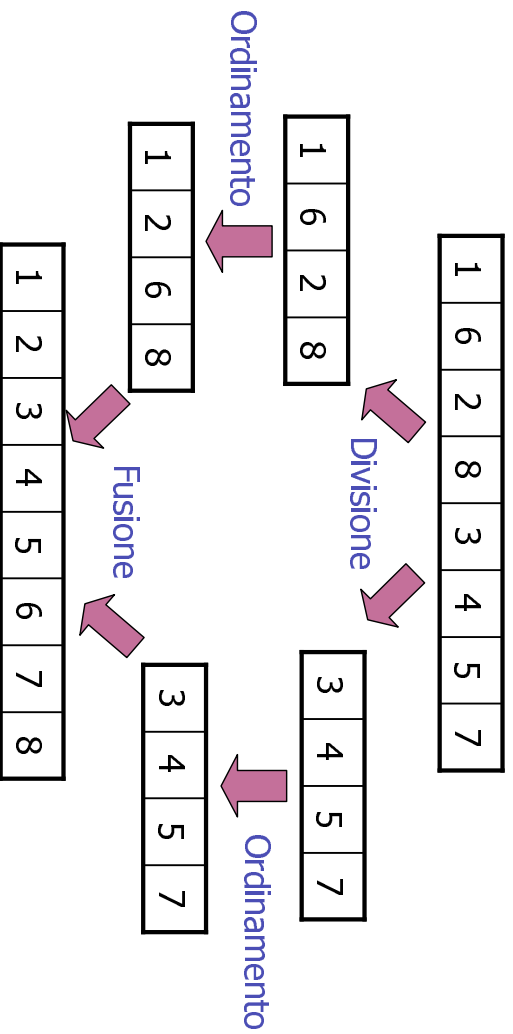
- Il mergesort è un algoritmo basato su una strategia **divide et impera**:
- Una strategia **divide et impera** consiste nel dividere un problema in sottoproblemi, nel risolvere i sottoproblemi e nel ricomporli per ottenere la soluzione del problema originale
- Il mergesort è composto da due fasi:
  - una fase di divisione del vettore in sottovettori
  - una fase di ricomposizione dei risultati (merge)

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Merge sort: l'idea sottostante

- Dato un vettore si divide in due sottovettori uguali, si ordinano i sottovettori e poi si rifondono insieme

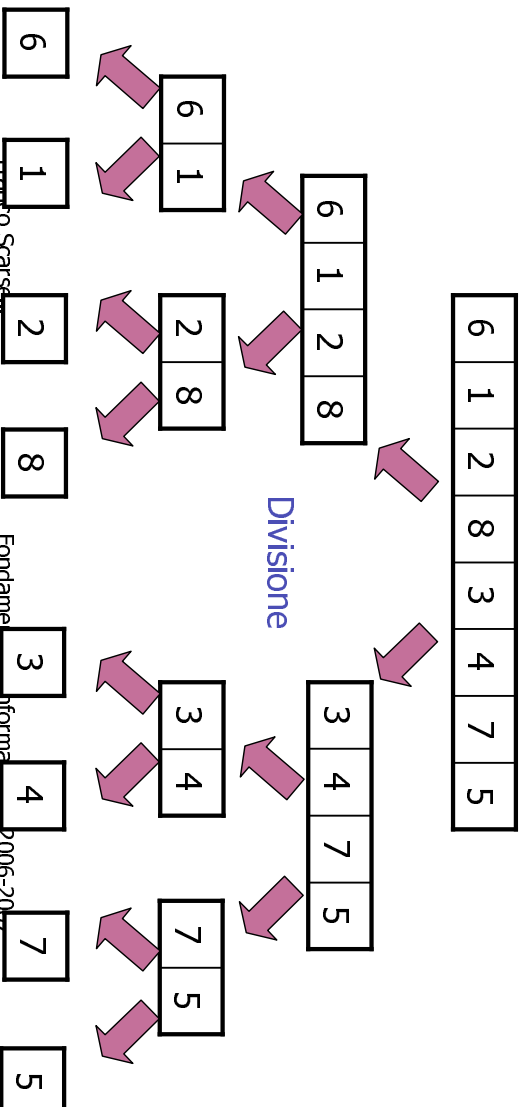


Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Mergesort: la divisione ricorsiva

- Come si ordinano i due sottovettori ?
  - applicando **ricorsivamente** la divisione fino a quando il vettore contiene un solo elemento: in tal caso l'ordinamento è banale



Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Mergesort: l'implementazione della divisione

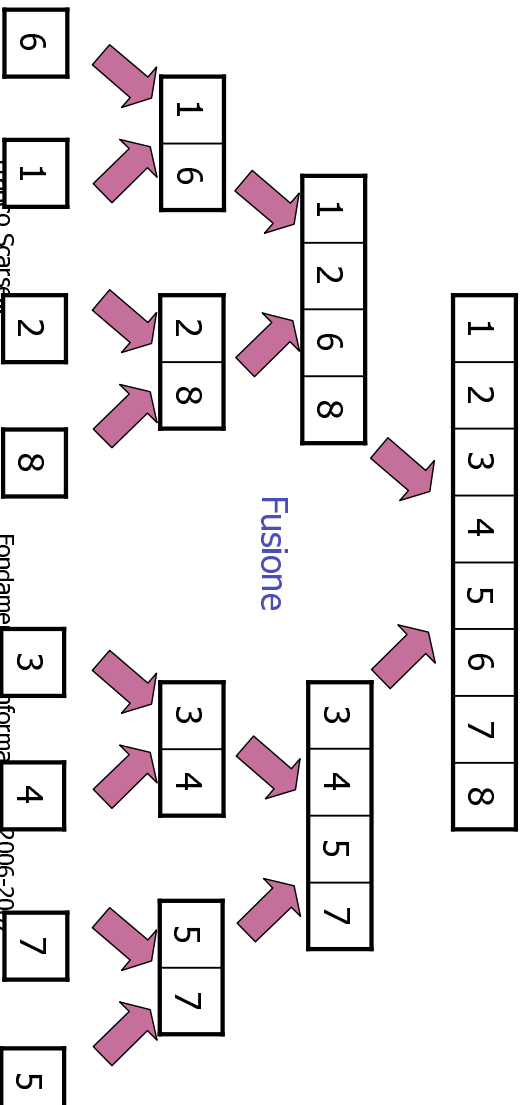
```
public void mergesort(int A[], int from, int to){  
    if((from!=to) ) {  
        int center = (from+to)/2;  
        mergesort(A,from,center);  
        mergesort(A,center+1,to);  
        merge(A,from,center,center+1,to);  
    }  
    //continua  
}
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Mergesort: la fusione ricorsiva

- Dopo aver diviso i vettori ricorsivamente, questi verranno fusi ricorsivamente



6

1

2

8

3

4

7

5

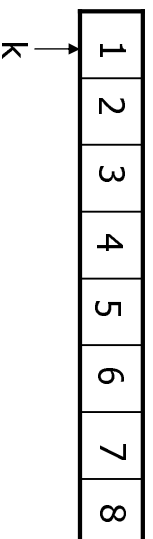
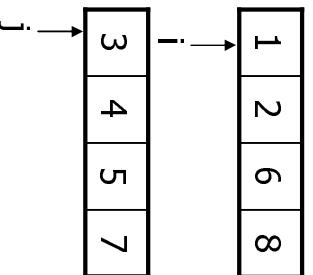
Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Mergesort: la fusione

- La fusione viene realizzata utilizzando due indici che scorrono i due vettori da fondere:

- ad ogni passo si confrontano i due elementi dei vettori indicati dagli indici:  $A[i]$ ,  $A[j]$
- si copia in uscita quello minore e si incrementa l'indice corrispondente
- si torna ad 1 fino a quando i due vettori non sono stati visitati



Franco Scarselli

Fondamenti di Informatica I, 2006-2007

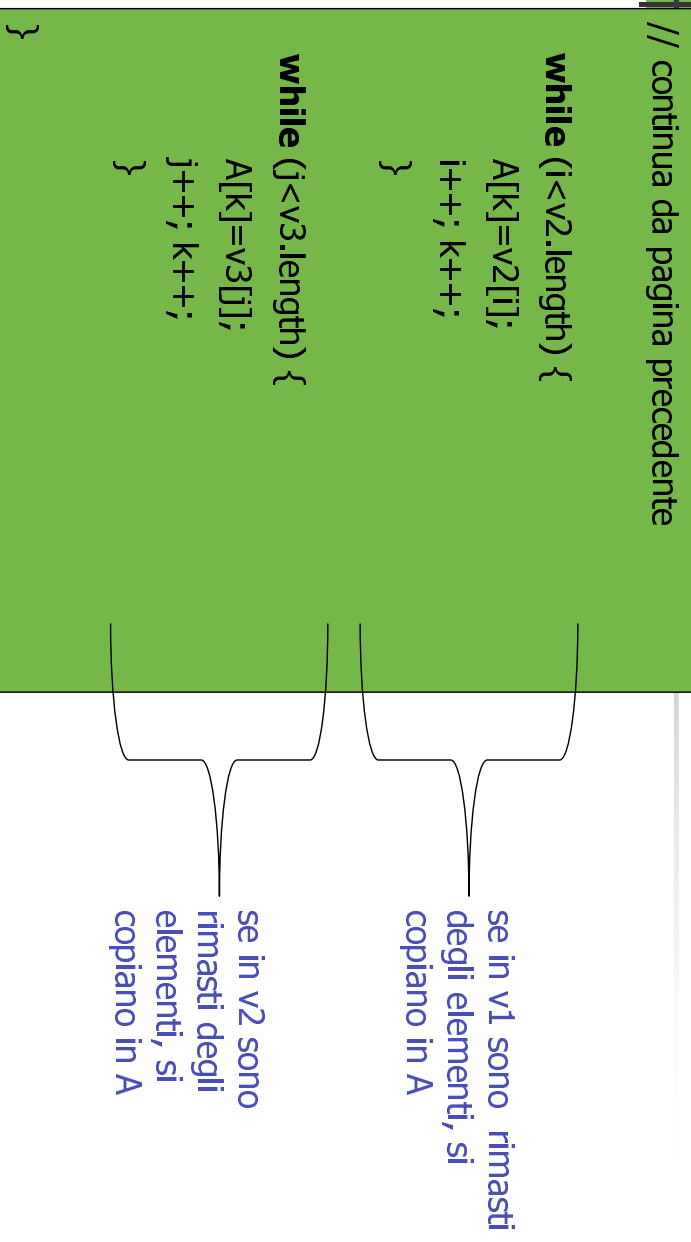
## Mergesort: l'implementazione della fusione

```
public void merge(int A[], int from1, int to1, int from2, int to2){
    int v2[] = new int[to1 - from1 + 1];
    int v3[] = new int[to2 - from2 + 1];
    ..... si copia la prima parte di A in v2 e la seconda in v3
    int i=0; j=0; k=0;

    while (i < v2.length && j < v3.length) {
        if(v2[i] < v3[j]){
            A[k] = v2[i];
            i++; k++;
        }
        else {A[k] = v3[j];
              j++; k++;}
    }
    // continua a pagina successiva
```

Fusione

# Mergesort: l'implementazione della fusione



Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Complessità del mergesort

- Il mergesort ha complessità  $O(n \cdot \log_2 n)$  sia nel caso medio che nel caso pessimo

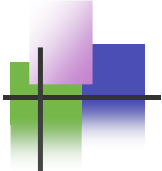
- Mergesort è un algoritmo **ottimo**!!

La sua complessità asintotica è la migliore possibile

(Questo fatto fondamentale non verrà dimostrato in questo corso, ma solo enunciato)

Comunque

- esistono algoritmi che per alcuni ingressi facciano meglio di  $n \cdot \log_2 n$
- esistono altri algoritmi con complessità  $n \cdot \log_2 n$  nel caso pessimo



# Osservazione sul passaggio di parametri in Java

## Passaggio dei parametri

- **per valore**
  - il contenuto della variabile viene copiato nel parametro: le modifiche fatte dal metodo al parametro **non modificano** il contenuto della variabile
- **per riferimento**

si passa l'handler della variabile: le modifiche fatte dal metodo al parametro **modificano** il contenuto della variabile

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Osservazione sul passaggio di parametri in Java

## Passaggio dei parametri

Java usa il passaggio per valore per i tipi semplici e quello per riferimento per gli oggetti

## Passaggio per valore (tipi semplici)

```
void modifica(int a){  
    a=3;  
}
```

```
int b=5;  
modifica(b);  
System.out.println(b);
```

Stampa 5

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Passaggio per riferimento (oggetti)

```
void modifica(int a[]){  
    for(int i=0;i<a.length;i++) a[i]=0;  
}
```

```
int b[]={1,2,3};  
modifica(b);  
for(int i=0;i<b.length;i++) System.out.println(b);
```

Stampa 0,0,0, ...

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

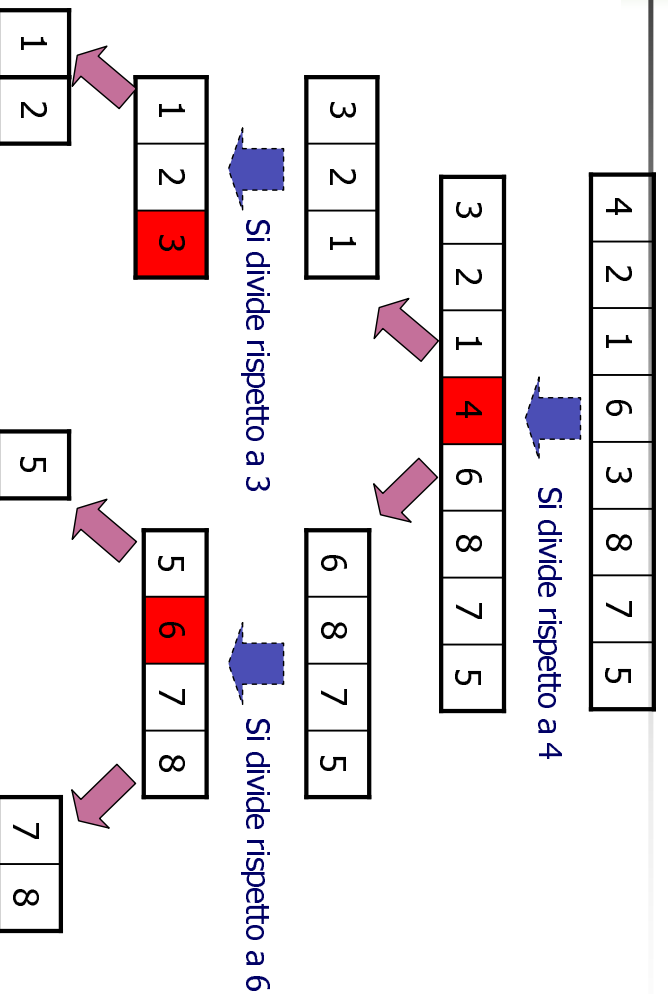
# Quicksort

- Il quicksort come il mergesort è un algoritmo divide et impera.
- L'idea
- Si divide il vettore A in due sottovettori che contengono rispettivamente tutti gli elementi maggiori e minori di  $A[0]$ , cioè il primo elemento del vettore detto perno
  - Si ripete ricorsivamente la divisione...

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Quicksort: esempio della divisione ricorsiva



Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# Quicksort: implementazione della divisione

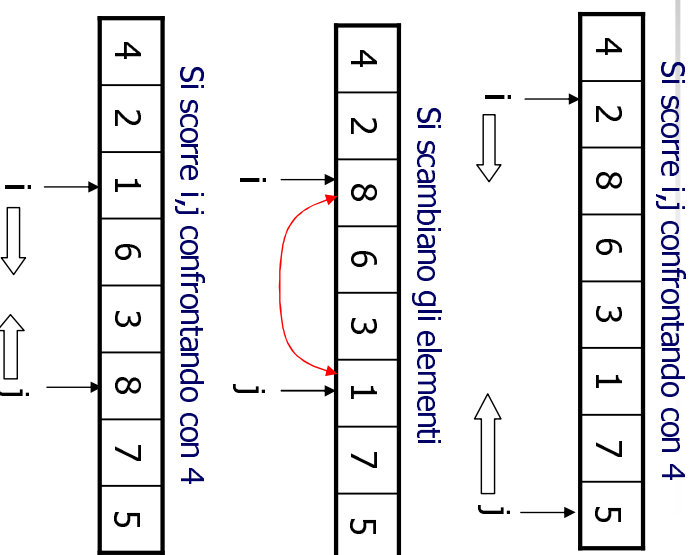
```
void quicksort(int A[], int from, int to){  
    if (from!=to){  
        int m=perno(vect,from,to);  
        quicksort(A,from,m-1);  
        quicksort(A,m+1,to);}  
}
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Quicksort: l'operazione perno

- Come si divide il vettore?
  - Si usano due indici i, j che scorrono il vettore rispettivamente da sinistra e da destra
  - L'indice i scorre fino a quando  $A[i] < A[1]$
  - L'indice j scorre fino a quando  $A[j] > A[1]$
  - Quando entrambi gli indici sono fermi si scambia gli elementi e si continua



Franco Scarselli

Fondamenti di Informatica I, 2006-2007

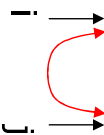
# Quicksort:

## l'operazione perno

- alla fine si scambia il perno la posizione indicata da j

Si scambiano gli elementi

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 6 | 3 | 8 | 7 | 5 |
|---|---|---|---|---|---|---|---|



alla fine si scambia il perno

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 6 | 8 | 7 | 5 |
|---|---|---|---|---|---|---|---|



|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 4 | 6 | 8 | 7 | 5 |
|---|---|---|---|---|---|---|---|

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

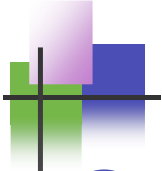
# Quicksort: implementazione del perno

```
void perno(int A[], int from, int to){
    int i=from+1, j=to;
    while(i<=j){
        while(A[i]<A[0]) i++;
        while(A[j]>A[0]) j--;
        if(i<j){scambia(A,i,j);}
    }
    scambia(A,0,j);
}
```

Scambia è un metodo che scambia gli elementi in due posizioni dell'array

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Quicksort: la complessità

- Il quicksort ha complessità media  $O(n \cdot \log n)$
- Il caso pessimo si verifica quando il perno finisce in fondo o in testa al vettore. In tal caso il quicksort ha complessità  $O(n^2)$

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Confronto fra gli algoritmi di ordinamento

### Mergesort vs Quicksort

- Mergesort ha il vantaggio di avere complessità sempre  $O(n \cdot \log n)$
- I Quicksort ha il vantaggio di non richiedere un vettore di appoggio: ordina il vettore usando il vettore dove si trovano i dati
- In media il quicksort si comporta bene, per questo motivo in pratica spesso è preferito al mergesort

### Altri algoritmi

- Esistono anche altri algoritmi di ordinamento: bubblesort, heapsort, ...
- Heapsort è un altro algoritmo ottimo

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Ancora sulla complessità

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## Problemi e algoritmi

- Anche per i problemi si parla di complessità
- Tipicamente non si riesce a definire univocamente complessità di un problema, perché
  - Lo stesso problema può essere risolto con algoritmi diversi che hanno complessità diverse
  - Anche si riesce a stabilire qual'è il **migliore** algoritmo per la risoluzione di un problema, non è se in futuro potranno esserne inventati algoritmi migliori
- Per questi motivi si parla solo di **limiti inferiori** e **limiti superiori** alla complessità di un problema

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Limiti inferiori

- In alcuni casi è possibile dimostrare che nessun algoritmo che risolve un problema può impiegare meno risorse di un certo limite (**limite inferiore**)
- **Esempi banali**
  - Nessun algoritmo che genera tutte le classifiche possibili di  $n$  concorrenti può farlo in meno di  $n!$  operazioni (la complessità inferiore è  $O(n!)$ )
  - Nessun algoritmo può fare la somma degli elementi di un vettore con  $n$  elementi in meno di  $n$  operazioni (la complessità inferiore è  $O(n)$ )
- **Esempio non banale**
  - Nessun algoritmo può ordinare un vettore di  $n$  elementi in meno di  $n \cdot \log n$  nel caso peggiore (la complessità inferiore è  $O(n \cdot \log n)$  nel caso peggiore)(Questo è un fatto ben noto di cui non diamo dimostrazione in questo corso)

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Algoritmi ottimi

- Un algoritmo si dice **ottimo**, quando la complessità è uguale al limite inferiore
- **Esempi**
  - Mergesort è ottimo!!
  - Si consideri problema di sommare gli elementi di un vettore: un algoritmo che scorre tutti gli elementi e li somma uno ad uno richiede  $O(n)$  operazioni: tale algoritmo è anche ottimo perché la sua complessità corrisponde con quella minima
  - Si consideri problema di inserire un elemento un albero binario ordinato bilanciato che contiene  $n$  elementi: l'algoritmo visto a lezione richiede  $O(\log n)$  operazioni, per cui  $\log n$  è un limite superiore per tale problema. Si può dimostrare che tale complessità corrisponde con il limite inferiore e tale algoritmo è anche ottimo.



# Algoritmi e computer

- Dubbi:

- la complessità è indipendente dal computer su cui gira il programma?
- Ad esempio, se uno inventasse un calcolatore in grado di generare contemporaneamente tutte le classifiche di n concorrenti, allora quel problema non avrebbe più complessità n!
- Ad esempio, potrebbe esistere in futuro un computer in grado di ordinare un vettore di qualsiasi lunghezza in una sola istruzione
- Nessuno conosce la risposta, ma fino ad adesso nessuno è mai riuscito progettare un computer con queste capacità
- Tutti i calcolatori conosciuti sono equivalenti in termini di capacità di calcolo ad un computer semplicissimo: **la macchina di turing**

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



## La macchina di Turing

- Alan Turing è un scienziato vissuto nella prima parte del novecento considerato uno dei padri dell'informatica
- Esistono varie versioni della macchina di Turing, quella più simile ai nostri calcolatori è a quella a registri
- Le macchine di Turing sono semplici computer equivalenti in termini di capacità di calcolo a qualsiasi computer esistente

# La macchina di Turing a registri

- E' costituita da un insieme di registri di lavoro  $R_1, R_2, R_3, \dots$  e un insieme di registri di ingresso  $I_1, I_2, I_3, \dots$
- Ogni registro è una cella di memoria che contiene un intero non negativo
- I programmi sono costituiti da tre semplici tipi di istruzioni a cui può   
■ **incremento:**  $R_i++$    
Un programma che somma il contenuto di  $R_1$  e  $R_3$   
Il registro i viene incrementato di uno
- **decremento:**  $R_i--$   
Il registro i viene decrementato di uno.  
Se il registro è già 0, l'istruzione non ha effetto
- **salto condizionato:** IF  $R_i$  GOTO  $L_1$   
Se il registro i contiene un valore maggiore di 0 si va all'istruzione  $L_1$

```
IF I1 GOTO ciclo
I2++
I1--
IF I1 GOTO ciclo
fine:
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## La tesi di Church-Turing

- La tesi di **Church-Turing** afferma che  
Ogni problema intuitivamente calcolabile (risolvibile) da un qualsiasi elaboratore è calcolabile da una macchina di Turing purché abbia a disposizione sufficiente tempo e memoria
- Nessuno è mai riuscito a confutare la tesi di Church-Turing
- La maggior parte dei ricercatori ritengono che sia vera

# Il problema dell'halt

- Supponiamo esista un programma `halt` che risolve il problema dell'Halt: `halt(P,I)` restituisce
  - `true` se `P` si ferma con l'ingresso `I`
  - `false` se `P` non si ferma con l'ingresso `I`
- Sia `Q` il programma qui accanto
- Cosa succede se applico `Q` a `Q` stesso?
  - `Q(Q)` termina o no ?
- Se `Q(Q)` termina allora `halt(Q,Q)` dovrebbe essere vero ... ma allora `Q(Q)` non dovrebbe terminare
- Se `Q(Q)` non termina allora `halt(Q,Q)` dovrebbe essere falso ... ma allora `Q(Q)` dovrebbe terminare
- **Quindi il programma `halt` non esiste!**

```
void Q(Program P){  
    while (halt(P,P)) {}  
}
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

## Problemi impossibili

- Esistono problemi molto difficili ... dei problemi non calcolabili da una macchina di Turing e (se la tesi di Church è vera) da nessun calcolatore!

### Esempi

- **Problema dell'Halt**  
Dato un programma e un suo ingresso, dire se il programma terminerà (o entrerà in ciclo)
- **Problema di Post**

Data un programma e due stati (uno stato è definito da un certo valore delle variabili), dire se a partire dal primo stato si potrà raggiungere il secondo

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# La macchina di Turing non deterministica

Un programma che assegna a caso i volatori  $\{0, 1\}$  a  $R_1$  e  $R_2$  e accetta solo se entrambi sono uno

- Nella macchina non deterministica, i programmi includono anche altre istruzioni
  - scelta casuale: FORK
    - prende in ingresso un insieme di istruzioni e ne esegue una a caso
  - istruzione di accettazione: ACCEPT
    - quando viene incontrata il programma termina correttamente
- Un problema è risolvibile se esiste un programma e una scelta casuale per cui il programma termina con ACCEPT e fornisce la risposta desiderata

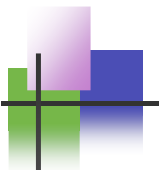
```
R4++  
FORK{ R1++,  
      R1-- }  
FORK{ R2++,  
      R2-- }  
IF R1 GOTO cont  
IF R4 GOTO no  
cont:  
IF R2 GOTO ok  
IF R4 GOTO no  
ok:  
ACCEPT  
no:
```

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

# La macchina di Turing non deterministica

- La macchina non deterministica non calcola di più di quella deterministica
  - Si può dimostrare che tutto ciò che è calcolabile sulla macchina di Turing non deterministica è calcolabile anche su quella deterministica
- Si pensa però che la macchina non deterministica sia più efficiente di quella deterministica
- Il non determinismo può essere pensato anche come parallelismo:
  - FORK può essere pensato anche come un'istruzione che genera più programmi paralleli
- Il parallelismo può permettere di risolvere i problemi velocemente: ad esempio, si può cercare un elemento in un vettore guardando contemporaneamente tutte le posizioni del vettore



# Problemi P e NP

- Ci sono tre classi di problemi che sono molto conosciuti nella teoria della complessità
    - $P =$  problemi risolvibili in tempo polinomiale sulla macchina di Turing  
Sono problemi facili, considerati risolvibili in un tempo ragionevole
    - $NP =$  problemi risolvibili in tempo polinomiale sulla macchina di Turing non deterministica
- Includono sia i problemi facili, che anche la quasi totalità dei problemi che si incontrano in situazioni pratiche

- **Ovviamente vale  $P \subseteq NP$ , ma non è noto se  $P \neq NP$**

Franco Scarselli

Fondamenti di Informatica I, 2006-2007

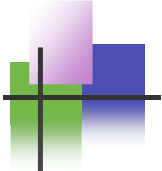


# Problemi NP-completi

- Esiste una classe di problemi **detti NP-completi** tutti equivalenti fra di loro che hanno una proprietà sorprendente:
  - Sarebbe sufficiente trovare un algoritmo polinomiale per uno solo di essi che avremmo trovato un algoritmo polinomiale per tutti!!
  - Inoltre, tutti i problemi in NP sarebbero risolvibili in tempo polinomiale sulla macchina di Turing deterministica, cioè avremo dimostrato  $NP=P$ !!

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# Problemi NP completi: esempi

- **Soddisfacibilità**  
Data una espressione booleana nelle variabili  $x_1, x_2, \dots, x_n$ , dire se esiste un assegnamento alle variabili tali che l'espressione è vera?
- **Il problema del commesso viaggiatore**  
Dato un insieme di  $n$  città con le relative distanze, trovare il cammino più breve che partendo da una città, le visiti tutte tornando in quella di partenza
- **Programmazione lineare intera**  
Dato una matrice  $A$  e due vettori  $b, c$ , trovare in vettore di interi  $x$ , che soddisfa  $Ax \leq b$  e minimizza  $cx$   
Possono essere risolti con la programmazione lineare problemi come quello di definire l'orario dei treni e degli autobus, definire l'orario delle lezioni, ....

Franco Scarselli

Fondamenti di Informatica I, 2006-2007



# $P=NP$ e tesi di Church

- Attualmente si pensa che  $NP \neq P$ ,
  - ma se nessuno è ancora riuscito a dimostrarlo
- Si pensa anche che la tesi di Church sia vera:
  - ovviamente questo non si può dimostrare, ma, è eventualmente, solo confutabile
- Molti metodi sono basati sulla teoria della complessità:
  - ad esempio, gli algoritmi di crittazione sono basati sul fatto che decryptare una chiave è molto complesso e richiederebbe un tempo molto lungo
  - se la tesi di Church non fosse vera o  $P=NP$ , tali metodi non sarebbero più efficaci