

Complessità

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

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, 2005-2006



Cosa si misura?

Complessita' temporale

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

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Cosa si misura?

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

Complessita' spaziale

- Si misurano le celle di memoria occupate
- La complessita' 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, 2005-2006



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, 2005-2006



Complessità temporale asintotica

Formalmente

- n è la dimensione del problema, cioè la dimensione la dimensione dell'input dell'algoritmo
- $T(n)$ è il tempo impiegato dall'algoritmo quando l'ingresso ha dimensione n
- $f(n)$ 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 α 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

Esempi

- Calcolare la somma degli elementi di un array
 - n = numero di elementi dell'array
 - complessità: $O(n)$
- Leggere un elemento di un array
 - n = numero di elementi dell'array
 - complessità: $O(1)$

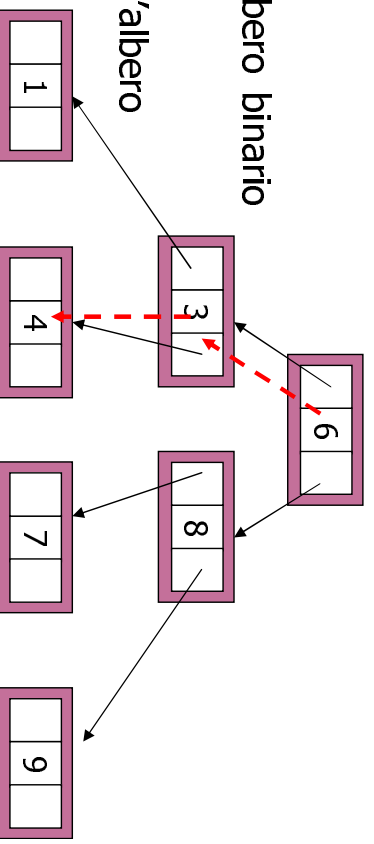
Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Complessità temporale asintotica

Esempi

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



- Inserire un elemento in albero binario ordinato bilanciato richiede circa d operazioni, d =profondità
- un albero binario bilanciato ha $n=2^{d+1}-1 \approx 2^{d+1}$ nodi
- quindi, $d \approx \log_2 n - 1$

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Complessità temporale asintotica

Esempi

- 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: $O(n!)$

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Complessità temporale asintotica

Esempi

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

```
void search(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, 2005-2006



Algoritmi facili e difficili ...

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

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Algoritmi con complessità esponenziale

- Perché gli algoritmi a complessita' esponenziale sono considerati quasi inusabili ?
- Si consideri il programma che genera tutte le classifiche finali di una competizione con n partecipanti, complessita' $O(n!)$ (è esponenziale, perché $n! \approx n^n$)
 - 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, 2005-2006



Complessità media e nel caso peggiore

- Un algoritmo può richiedere un numero di operazioni diverse per ingressi di dimensione uguale:
 - Un'ulteriore affinamento nello studio della complessità consiste nel distinguere fra *complessità media* e *complessità nel caso peggiore*
- **Esempio**
- Per trovare un elemento in una lista di n elementi occorre visitare
 - Un solo elemento nel caso fortunato in cui l'elemento da cercare sia il primo
 - n elementi, nel caso sfortunato un cui l'elemento cercato sia l'ultimo
 - $n/2$ elementi in media

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Esempi di algoritmi

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



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, 2005-2006



Il bubble sort (ordinamento a bolla)

- Il vettore viene sottoposto a più passaggi
- Un passaggio consiste in una scansione di tutto il vettore durante la quale ogni elemento *vect[i]* viene confrontato con quello successivo *vect[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, 2005-2006

Bubble sort: esempio

i=2

Primo passaggio:

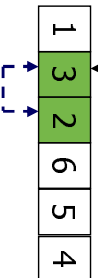
l'indice i viene inizializzato a 0

i=0



Si confronta vect[i] con vect[i+1]: si scambiano

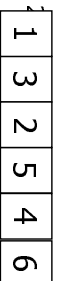
i=1



Si confronta vect[i] con vect[i+1]: non si fa niente

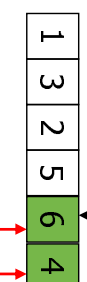
Franco Scarselli

Fondamenti di Informatica I,



Si confronta vect[i] con vect[i+1]: si scambiano

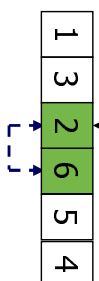
i=4



Si confronta vect[i] con vect[i+1]: si scambiano

Si confronta vect[i] con vect[i+1]: non si fa niente

i=3



Bubble sort: esempio

i=3

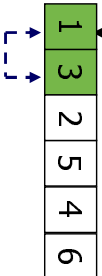


i=4

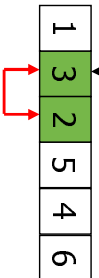
Secondo passaggio:

l'indice i viene inizializzato a 0 di nuovo

i=0



i=1



i=2



Franco Scarselli

Fondamenti di Informatica I, 2005-2006

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

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

Implementazione di bubble sort



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 primo 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, 2005-2006



Implementazione del fattoriale

Implementazione iterativa

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

Implementazione ricorsiva

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

Successione di Fibonacci

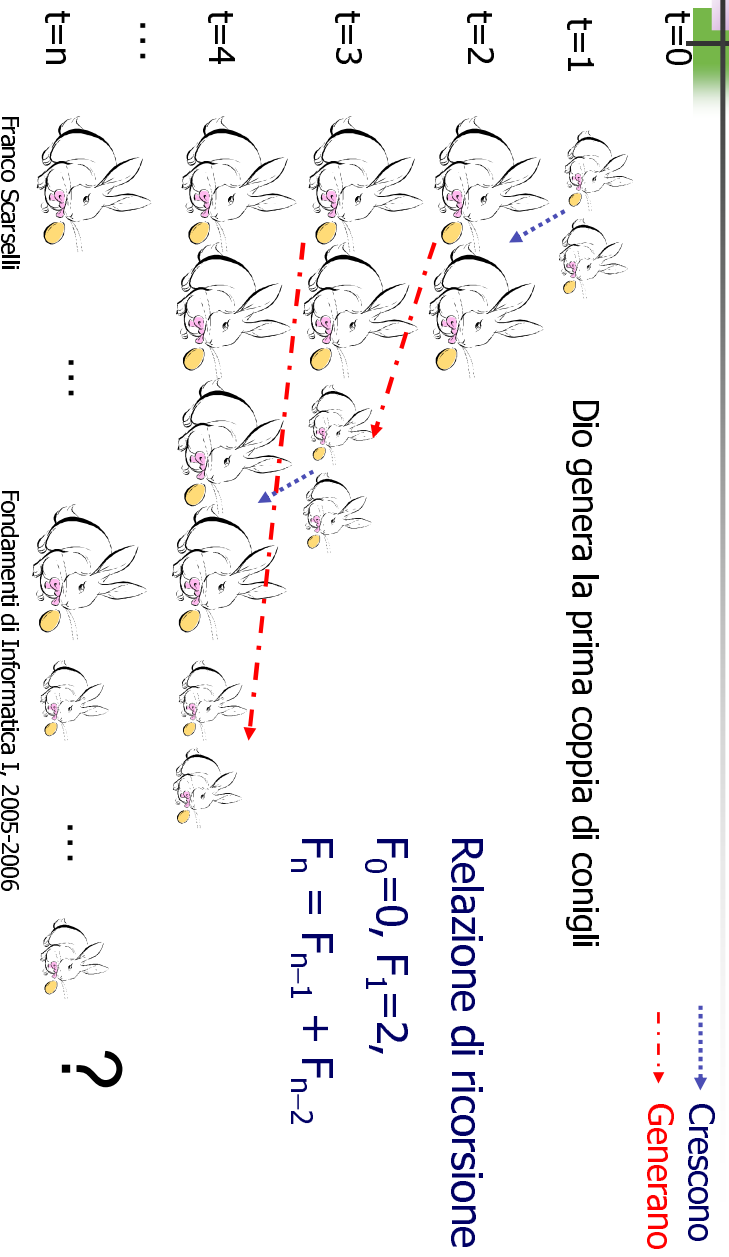
Successione di Fibonacci

- 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?

$$F_0=0, F_1=1,$$

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

Successione di Fibonacci





Implementazione della serie di Fibonacci

```
class MyMath{  
    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, 2005-2006



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 **vect[vect.length/2]**
- Se il **val** è minore si ripete la ricerca sulla metà sinistra, altrimenti si ricerca nella metà destra

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

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, 2005-2006

Implementazione della ricerca dicotomica

```
class MyMath{
    int vect[];

    public int search(int val, int from, int to){
        if (from==to) {if (vect[from]==val) {
                                return from;}}
                        else {return -1;}}
        }
        else {int center=Math.floor((from+to)/2);
                if (val<vect[center]){
                    return search(val,from,center-1);}
                else {if(val==vect[center]) {
                    return center;}}
                else{return search(val,center+1,to);}}
        }
    }
```



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, 2005-2006



Mergesort

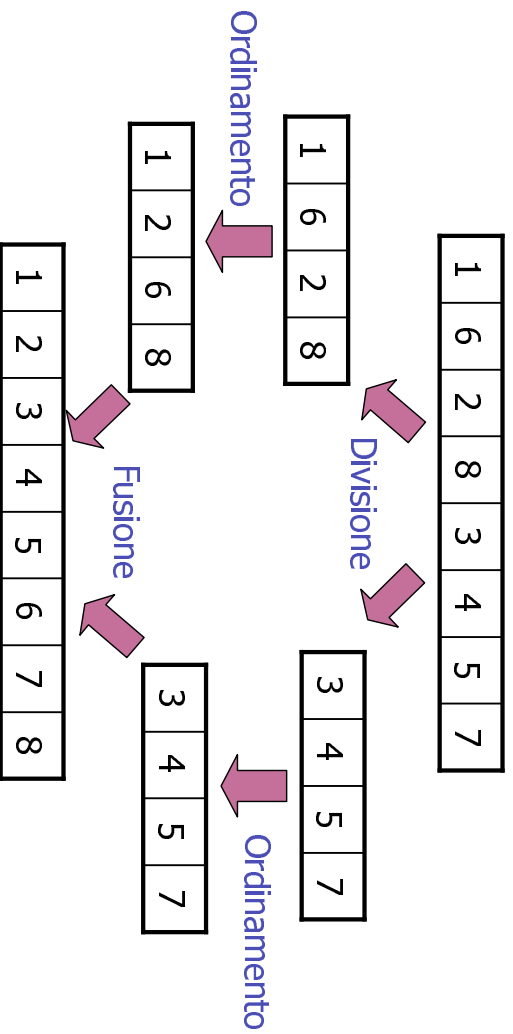
- Il mergesort è un algoritmo basato su una strategia **divide et impera**: consiste nel dividere un problema in sottoproblemi, nel risolvere i sottoproblemi e nel ricomporli per ottenere la soluzione del problema originale
- Il mergesort è composto da due fasi:
 - una fase di divisione del vettore in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

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Merge sort: l'idea sottostante

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

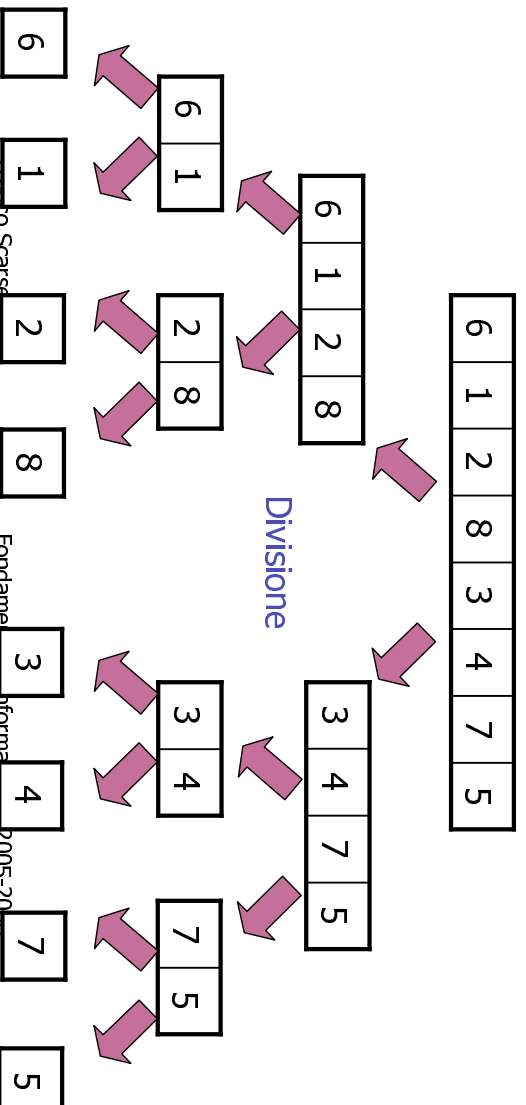


Franco Scarselli

Fondamenti di Informatica I, 2005-2006

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

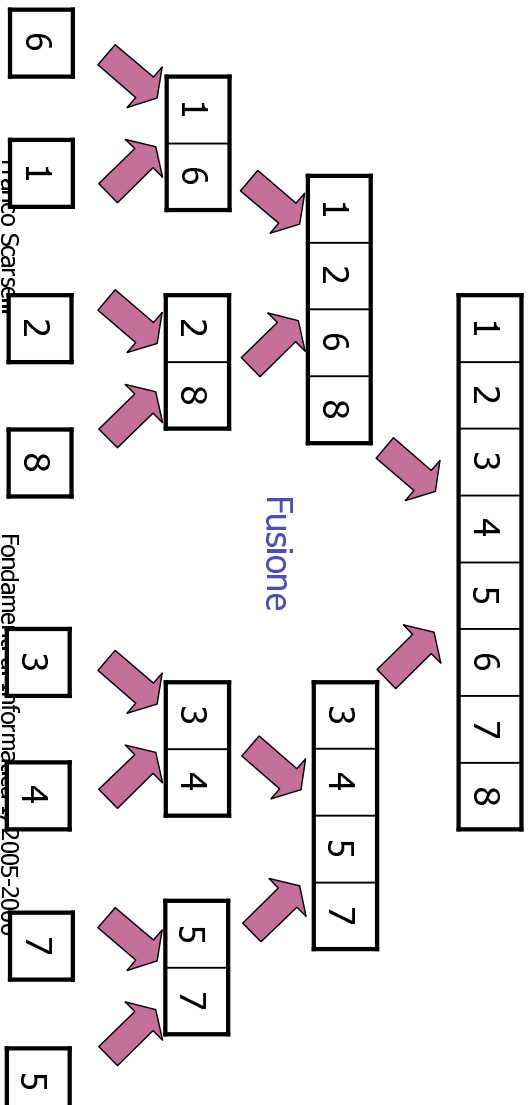


Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Mergesort: la fusione ricorsiva

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



Mergesort: l'implementazione della divisione

```
class Sorter{  
  
    public void mergesort(int vect[], int from, int to){  
        if((from==to)){  
            return;}  
        else{  
            int center=Math.floor((from+to)/2);  
            mergesort(vect,from,center);  
            mergesort(vect,center+1,to);  
            merge(vect,from,center,to);  
        }  
    }  
}
```

//continua

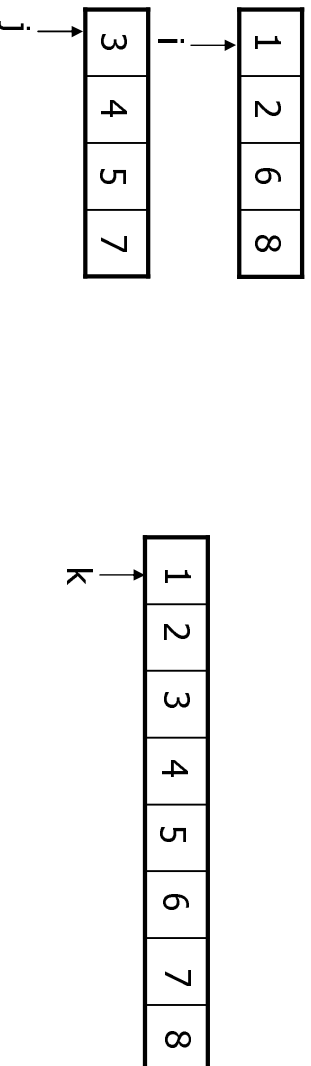
Trattato Scarsell

Fondamenti di Informatica I, 2005-2006

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: vect[i], vect[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, 2005-2006

Mergesort: l'implementazione della fusione

```
public void merge(int vect[], int from, int center, int to){
    int v2[]=new int[center-from+1];
    int v3[]=new int[to-center];
    //si copia la prima parte di vect 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]){
            vect[k]=v2[i];
            i++; k++;
        }
        else {vect[k]=v3[j];
              j++; k++;}
    }
    // continua a pagina successiva
```

Fusione

Mergesort: l'implementazione della fusione

```
// continua da pagina precedente
if (i==v1.length){
    while (i<v2.length) {
        vect[k]=v2[i];
        i++; k++;;}
    }
else {while (j<v3.length) {
        vect[k]=v3[j];
        j++; k++;;}
    }
}
```

se in v1 sono rimasti degli elementi, si copiano in vect

se in v2 sono rimasti degli elementi, si copiano in vect

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Complessità del mergesort

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

- Mergesort è un algoritmo ottimo!!
(Questo fatto non verrà dimostrato in questo corso, ma solo enunciato)

La sua complessità asintotica è la migliore possibile, ma

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

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

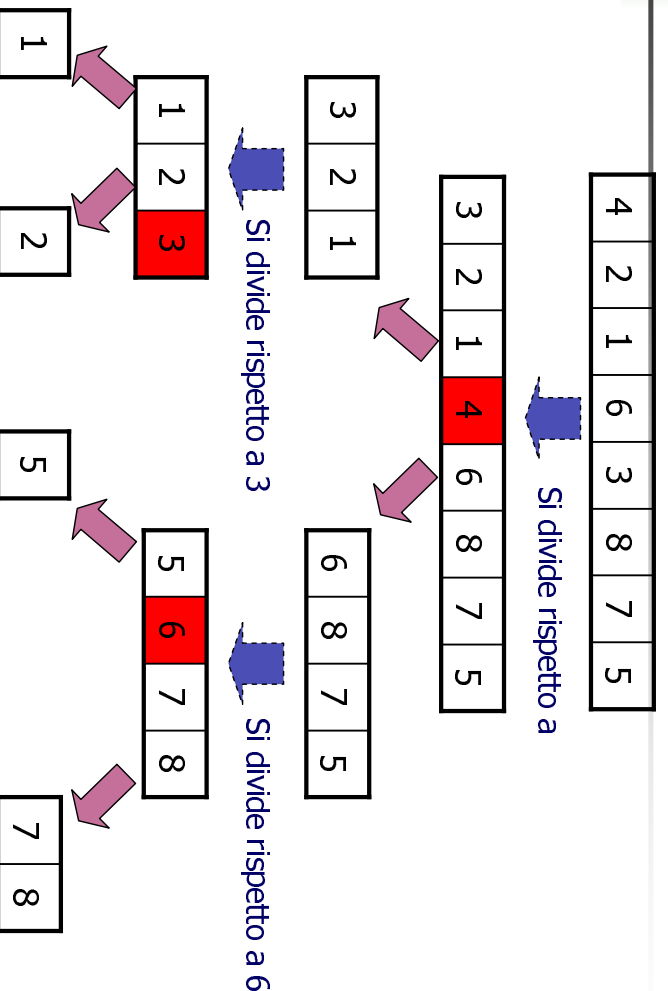
Quicksort

- Il quicksort come il mergesort è un algoritmo divide et impera.
- L'idea
- Si divide il vettore `vect` in due sottovettori che contengono rispettivamente tutti gli elementi maggiori e minori di `vect[1]`, cioè il primo elemento del vettore detto perno
 - Si ripete ricorsivamente la divisione...

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Quicksort: esempio della divisione ricorsiva



Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Quicksort: implementazione della divisione

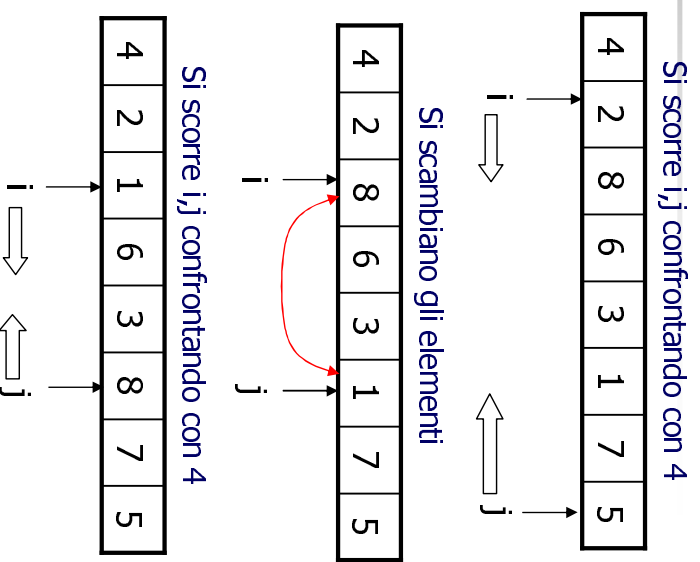
```
void quicksort(int vect[], int from, int to){  
    if (from==to) return;  
    int m=perno(vect,from,to);  
    quicksort(vect,from,m-1);  
    quicksort(vect,m+1,to);  
}
```

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

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 $\text{vect}[i] < \text{vect}[1]$
 - L'indice j scorre fino a quando $\text{vect}[j] > \text{vect}[1]$
 - Quando entrambi gli indici sono fermi si scambia gli elementi e si continua



Franco Scarselli

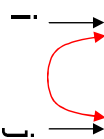
Fondamenti di Informatica I, 2005-2006

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, 2005-2006

Quicksort: implementazione del perno

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

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Quicksort: la complessità

- Il quicksort ha complessità media $O(n \log n)$
- 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, 2005-2006



Confronto fra gli algoritmi di ordinamento

Mergesort vs Quicksort

- Mergesort ha il vantaggio di avere complessità sempre $O(n \log n)$
- 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

Bubble sort

- Il bubble sort e un algoritmo simile il selection sort non sono molto usati, ma possonon essere utili quando i dati sono quasi già ordinati perché in questo caso sono molto veloci

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



Ancora sulla complessità

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



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, 2005-2006



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, 2005-2006



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, 2005-2006



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 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 e la maggior parte dei ricercatori ritengono che sia vera
- Se qualcuno riuscisse a confutarla, potrebbe sconvolgere l'informatica
 - Ad esempio, alcuni algoritmi per la codifica dei messaggi potrebbero divenire facilmente vulnerabili

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

La macchina di Turing a registri

- E' costituita da un insieme di registri R_1, R_2, R_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++

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

Un programma che somma il contenuto di R_1 e R_3

```
R4 ++  
IF R1 GOTO ciclo  
IF R4 GOTO fine  
ciclo:  R3 ++  
        R1 --  
        IF R4 GOTO ciclo  
fine:
```

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
- FORK può essere pensato anche come un'istruzione che genera più programmi paralleli

Fondamenti di Informatica I, 2005-2006

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

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 $P \neq NP$



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 che!
- Come per la tesi di Church la maggior parte dei ricercatori ritengono che valga $P \neq NP$ e
 - dimostrare che $P=NP$ sconvolgerebbe l'informatica
 - mentre, dimostrare che $P \neq NP$ è un risultato da Nobel....

Franco Scarselli

Fondamenti di Informatica I, 2005-2006



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 visita 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,

Problemi impossibili

- Esistono problemi ancora più difficili dei problemi non risolvibili da una macchina di Turing !!

Esempi

- Problema dell'Haltng

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 potrà raggiungere il secondo

Franco Scarselli

Fondamenti di Informatica I, 2005-2006

Perché il problema dell'halt non è calcolabile ?

Questo è il programma Q

Dimostrazione per assurdo

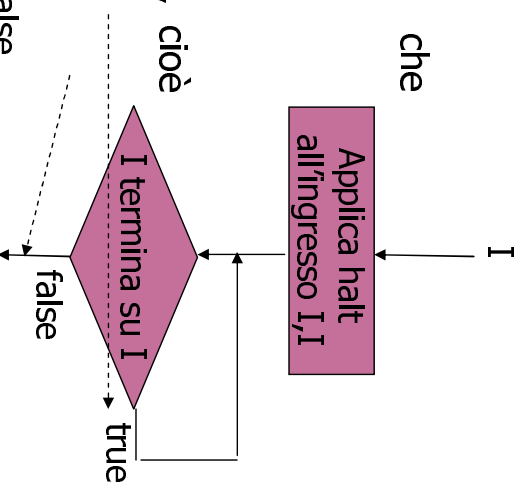
1. Supponiamo esista un programma halt che risolve il problema dell'Haltng

halt(Q,A) dice se Q si ferma con l'ingresso A

1. Sia Q il programma a destra

2. Allora il programma applicato a se stesso, cioè quando $I=Q$

- termina, ma allora vuol dire che sceglie true
- non termina, ma allora vuol dire che sceglie false



```
if (!halt(I,I)) { hai finito }  
else { entra in un ciclo infinito }
```



..... the end

Franco Scarselli

Fondamenti di Informatica I, 2005-2006