

Lezione 1: Introduzione agli Algoritmi

<http://idefix.mi.infn.it/~palombo/didattica/Lab-TNDS/CorsoLab/LezioniFrontali>

Fernando Palombo

Algoritmi: Nozioni Introduttive

Problema computazione: serie di domande (omogenee) a cui dare una risposta. Esempio : risolvere una equazione di secondo grado a coefficienti costanti:

$$a x^2 + bx + c = 0$$

Istanza di questo problema è una specifica domanda del problema considerato . Per esempio determinare le soluzioni della particolare equazione:

$$3 x^2 + 5x + 6 = 0$$

Un problema può essere descritto come un insieme di coppie ordinate (x,y) dove x è una istanza del problema e y la soluzione dell'istanza x

Algoritmi: Nozioni Introduttive

Dat un problema computazionale noi vogliamo realizzare una procedura che con una sequenza di passi successivi ci permetta di risolverlo. Noi chiamiamo **ALGORITMO** questa procedura.

Un algoritmo risolve il problema se ricevendo in input i dati che rappresentano una istanza, esso fornisce la soluzione all'istanza.

Esistono problemi semplicissimi a cui dare una risposta (tipo quello considerato) e problemi estremamente complicati (tipo gestire l'attività produttiva di una multinazionale).

La Fisica (da quella sperimentale a quella teorica) fa uso costante di algoritmi.

Esistono librerie con algoritmi già codificati (per esempio in C++) ed ottimizzati

Algoritmi: Nozioni Introduttive

Spesso l'algoritmo giusto per il proprio problema non esiste; talvolta esiste ma bisogna adattarlo o aggiornarlo. In certe situazioni l'algoritmo esiste ma non è accessibile.

Quindi saper scrivere un algoritmo o mettere le mani per cambiare o adattare un algoritmo esistente alle proprie esigenze è importante per un fisico

Vorrei aggiungere a questo punto che avere queste competenze è un **bonus** enorme per chi cerca lavoro. Un fisico (o matematico) che sappia programmare è molto spesso preferito agli informatici !!

Algoritmi: Nozioni Introduttive

Non tutti i problemi sono risolvibili (anzi la maggior parte non lo è!):
il problema può avere infinite soluzioni o necessita di infinite azioni, essere un problema di cui non è stata trovata la soluzione o essere un problema per il quale si è dimostrato che non esiste metodo risolutivo automatizzabile. Noi consideriamo solo problemi che ammettono un metodo risolutivo.

Progetto e Design :Analizzato il problema bisogna identificare chiaramente i passi logici per arrivare alla soluzione. Questa è la fase di progetto e design dell'algoritmo , **di gran lunga la fase più complessa e delicata.**

Codifica: Definita la successione di passi per arrivare alla soluzione, l'algoritmo va scritto in un linguaggio che il computer capisce. Questa è la fase del programmatore. L'algoritmo deve fornire la stessa soluzione indipendentemente dal linguaggio di programmazione scelto. Nei casi semplici designer e programmatore coincidono.

Verifica: controllo che l'algoritmo fornisca la soluzione giusta!!

Proprietà degli Algoritmi

Un algoritmo deve essere:

Finito: Tra l'inizio e la fine l'algoritmo deve compiere un numero finito di passi;

Deterministico: In corrispondenza allo stesso input, l'algoritmo fornisce sempre la stessa soluzione;

Non ambiguo: Tutti i passi da eseguire devono essere interpretati in modo non ambiguo dal computer. Un passo ambiguo per esempio è dire al computer di sostituire $\sin x$ con x per valori piccoli (??) di x .

Realizzabile: tutti i passi devono poter essere realizzati in un tempo finito






Diagramma di Flusso e Pseudo-codice

Due formalismi principali per rappresentare un algoritmo: **diagrammi di flusso (flowchart) e pseudo-codice**. Il primo utilizza simboli grafici mentre nel secondo l'algoritmo è descritto a parole.

Nel flowchart c'è un blocco iniziale, un blocco finale, un numero finito di blocchi di azione. Nel flowchart valgono le seguenti condizioni:

- 1) Ciascun blocco di azione e ciascun blocco di controllo hanno una sola freccia entrante ed una sola freccia uscente;
- 2) Ciascuna freccia entra in un blocco o si inserisce in un'altra freccia;
- 3) Ciascun blocco è raggiungibile dal blocco iniziale;
- 4) Il blocco finale è raggiungibile dal blocco iniziale.

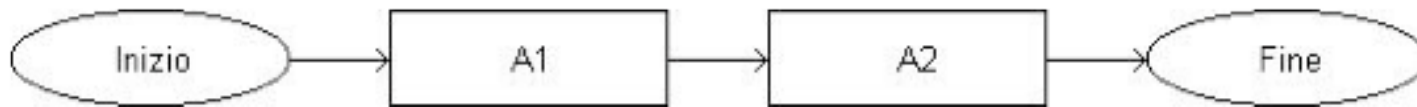
Blocchi Elementari

Blocchi elementari	Significato
	Limiti del processo
	Blocco azione
	Simbolo decisionale
	Interconnessione
	Comunicazioni con l'esterno

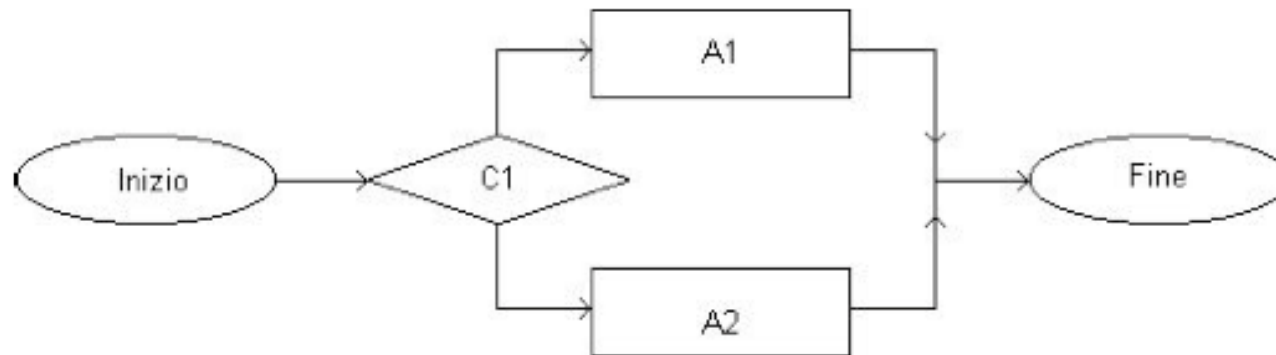
Schemi di Composizione

Tra i possibili schemi di composizione (che soddisfano le condizioni appena viste) ci sono quelli **fondamentali** chiamati di Sequenza, Selezione e Iterazione

Sequenza: composizione in sequenza di azioni elementari:

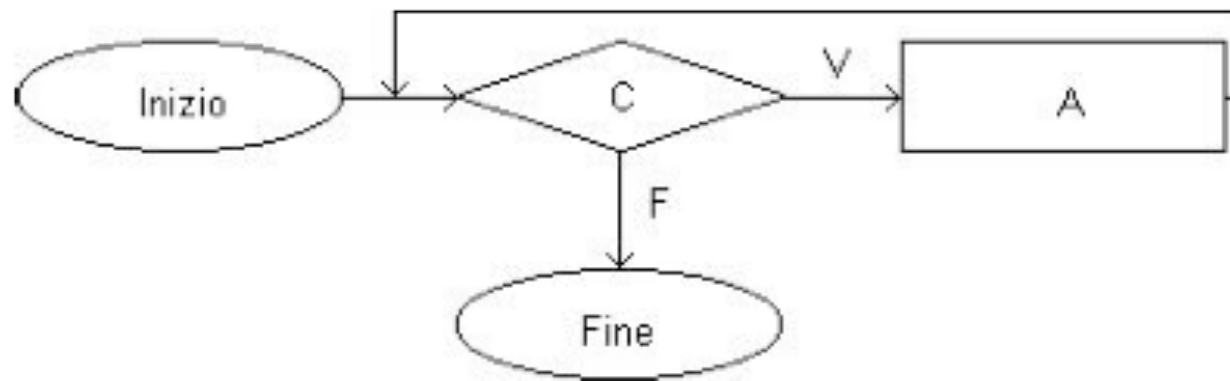


Selezione: il cammino percorso dipende dal controllo

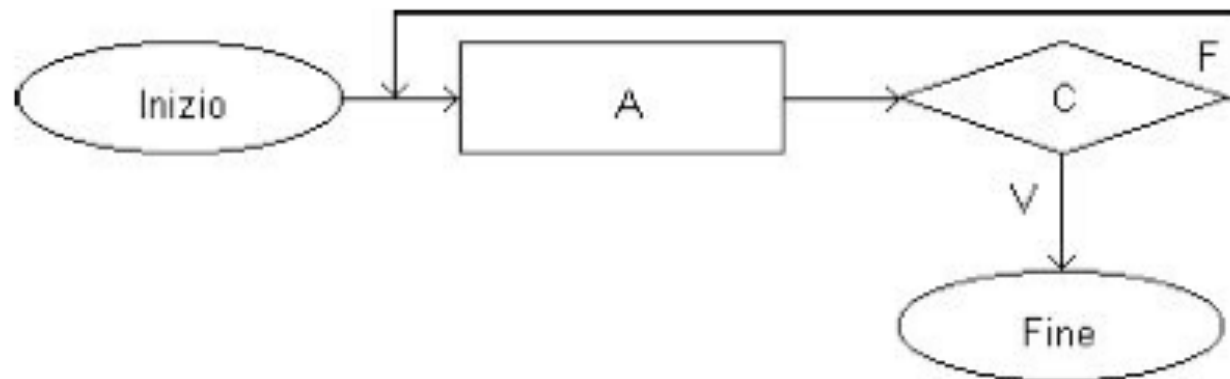


Schemi di Composizione

Iterazione: questo schema indica che l'azione deve essere ripetuta sino a che lo richiede l'esito del controllo.



Controllo
In testa



Controllo
a monte

Algoritmi Strutturati

Due algoritmi si dicono equivalenti (debolmente) se sottoposti agli stessi dati in input o non terminano o terminano producendo gli stessi risultati.

Un flowchart si dice strutturato se riconducibile ai tre schemi di composizione fondamentali

Teorema di Böhm-Jacopini

Dato un algoritmo costruito con un flowchart qualsiasi, è sempre possibile realizzare un altro flowchart equivalente al primo e strutturato.

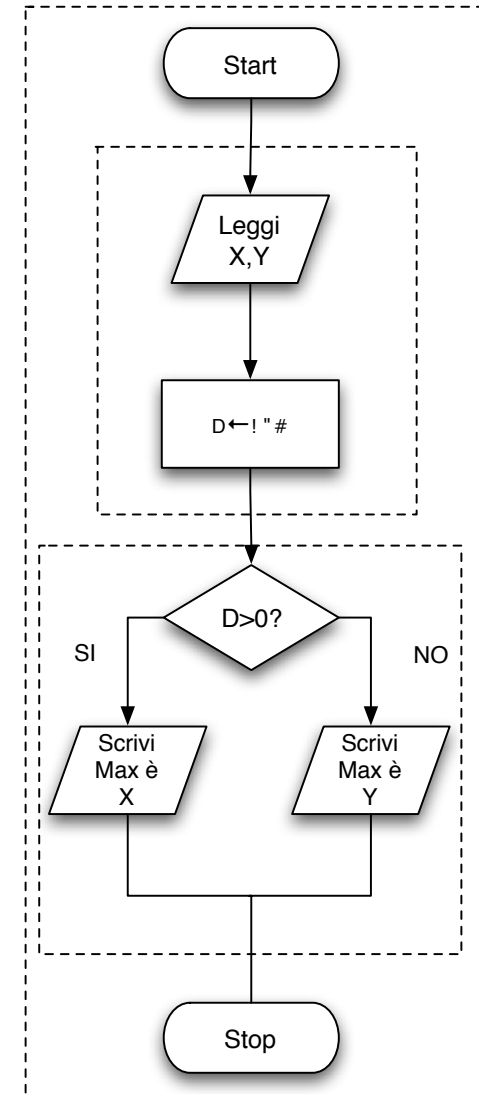
Questo importante teorema mi dice che io posso sempre fare a meno di salti incondizionati (GO TO), che rendono estremamente complesso seguire il flusso logico nell'algoritmo.

Diagramma di Flusso

Diagramma di flusso di un algoritmo strutturato che cerca il massimo tra due numeri naturali (in figura)

Sviluppo degli algoritmi di tipo **Top-Down** : si inizia da un livello alto di astrazione raffinando via via la descrizione dell'algoritmo sino ad arrivare ai dettagli.

Il diagramma di flusso è di facile comprensione, convertibile facilmente in più programmi codificati in diversi linguaggi di programmazione. Sono però ingombranti e spesso usano più fogli per cui diventa difficile seguire il flusso logico come anche eseguire modifiche al diagramma



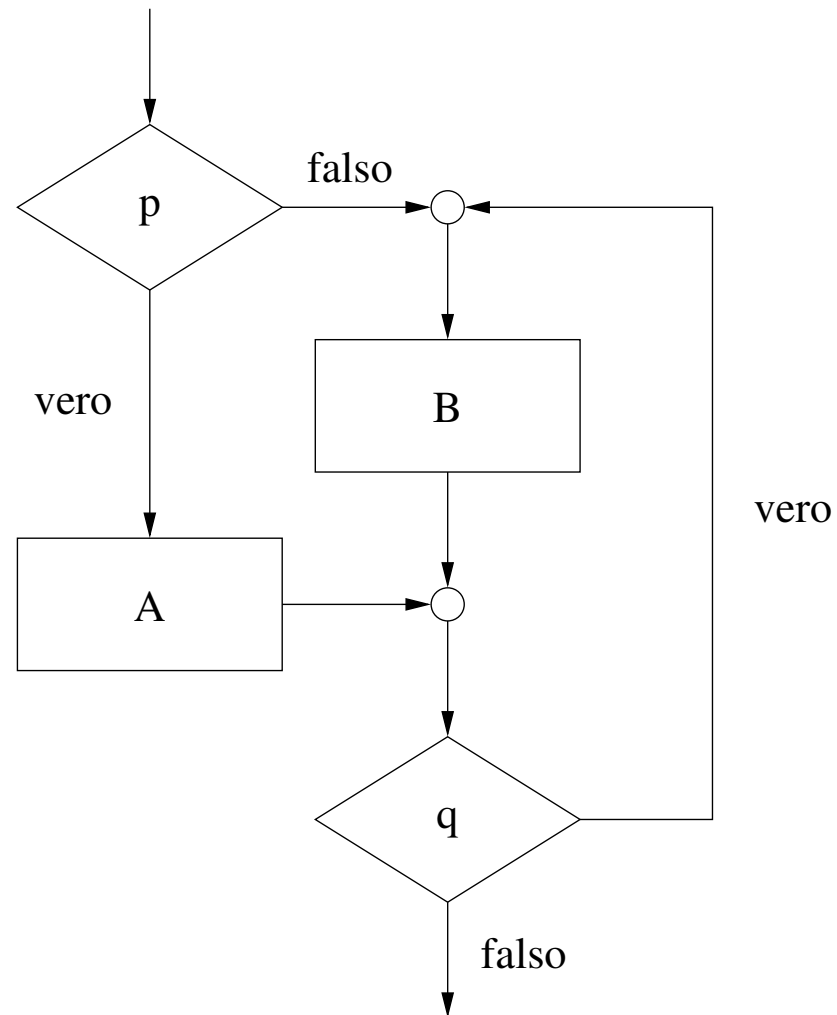
Pseudo-codice

Questo metodo di rappresentazione di un algoritmo è molto semplice ed efficace perché utilizza il linguaggio naturale. L'algoritmo è descritto a parole. Questo in generale rende molto facile il passaggio alla effettiva descrizione in un linguaggio di programmazione.

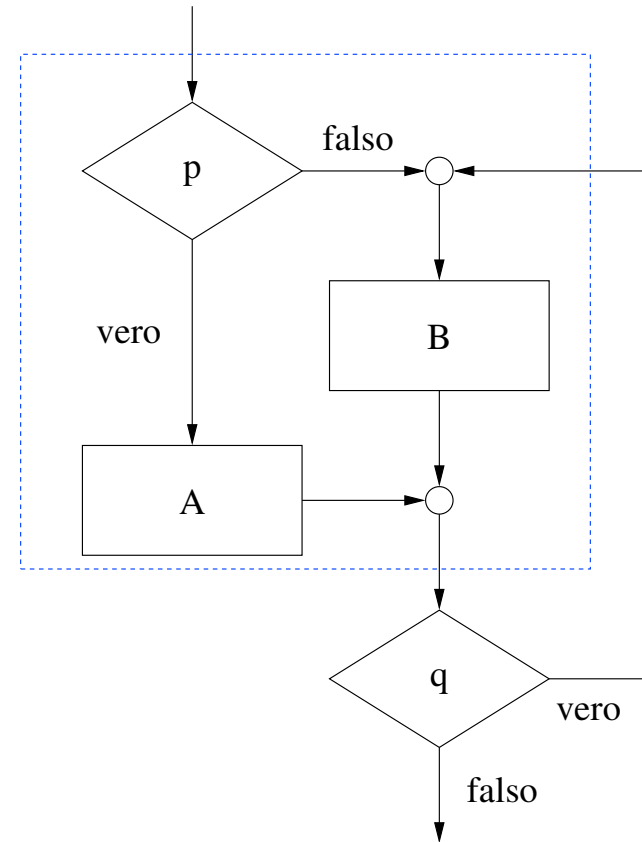
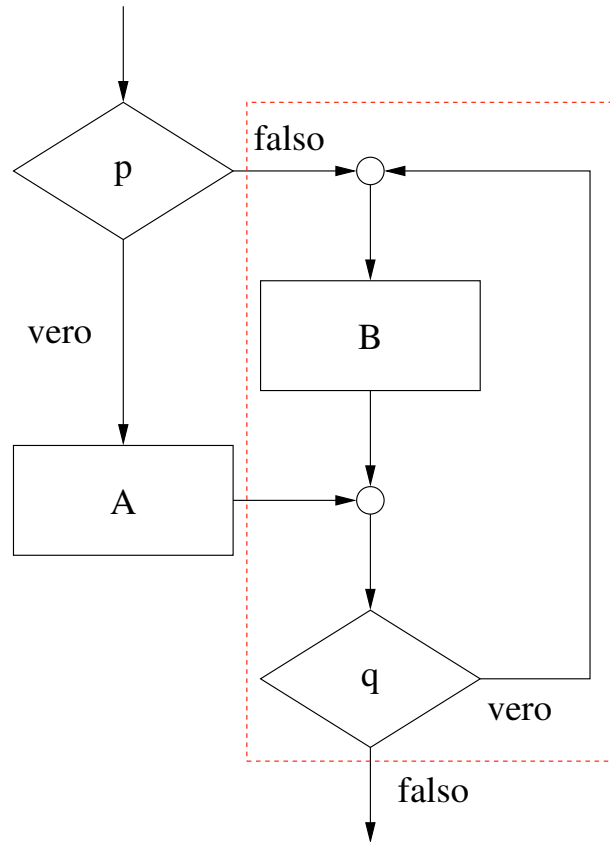
Esempio di pseudo codice (ricerca del massimo tra due numeri naturali):

leggi un numero dall'esterno e assegnalo alla variabile X
leggi un numero dall'esterno e assegnalo alla variabile Y
calcola la differenza tra X e Y e assegnala alla variabile D ($D \leftarrow X - Y$)
Se D è positivo, allora scrivi “ il massimo è X”
altrimenti scrivi “ il massimo è Y”

Esempio di Blocco non Strutturato



Esempio di Blocco non Strutturato



I blocchi evidenziati hanno entrambi due linee di ingresso

Confronto tra Due Algoritmi

Moltiplicazione di due numeri complessi: $(a + ib)(c + id) = [ac - bd] + i[ad + bc]$

Input: a, b, c, d output: $ac - bd, ad + bc$

Supponendo che una moltiplicazione di numeri reali costi 1 e una somma costi 0.01 allora il costo totale è **4,02**

È possibile fare meglio?

$$M1 = ac$$

$$M2 = bd$$

$$A1 = M1 - M2 = ac - bd$$

$$M3 = (a+b)(c+d) = ac + ad + bc + bd$$

$$A2 = M3 - M1 - M2 = ad + bc$$

Con questo metodo (di Gauss) il costo è di **3.05** (25% più efficiente)

Confronto tra Due Algoritmi di Somma

Sommare i primi cento numeri interi positivi:

$$1+2+3+\dots +99+100$$

Metodo diretto : sommo i numeri uno alla volta

Le operazioni necessarie sono:

- 1 operazione di lettura [numero di interi da sommare (100 nel nostro caso)]
- 2 operazioni di inizializzazione (la somma e il contatore dell'iterazione)
 - 100 confronti (Per $x < 100$)
- 2×100 operazioni di somma nel loop
- 1 operazione di stampa

Quindi in totale **304** operazioni

Se in input invece di 100 abbiamo 200 il numero di operazioni richieste sarebbe **604**

Questo numero è proporzionale a 100 (input)

Leggi 100

$x=0$

Sum=0

Per $x < 100$

{ $x=x+1$

Sum=Sum+x

}

Stampa Sum

Confronto tra Due Algoritmi di Somma

Sommare i primi cento numeri interi positivi

: $1+2+3+\dots+99+100$

Uso la **formula di Gauss**:

$$1+2+\dots+99+100=100*(100+1)/2$$

Leggi 100

$$\text{Sum}=100*(100+1)/2$$

Stampa Sum

Operazioni richieste:

1 lettura dell'input

1 somma + 1 moltiplicazione + 1 divisione)

1 operazione finale di stampa

Si noti che il numero di operazioni richiesto è indipendente dall'input.

Se invece di 100 avete 1000, l'algoritmo fa sempre 5 operazioni

Tempo di esecuzione costante indipendente dall'input mentre l'algoritmo precedente

ha un tempo di esecuzione proporzionale al numero di addendi da sommare.

La maggiore efficienza di questo algoritmo rispetto al precedente cresce con la dimensione dell'input

Complessità Computazionale

In fase di progettazione di un algoritmo un aspetto molto importante è l'analisi dell'algoritmo in termini di tempo di esecuzione e utilizzo delle altre risorse del sistema di elaborazione (come la CPU). Si parla quindi di **efficienza** di un algoritmo:

- Efficienza di esecuzione **T** (detta anche complessità computazionale o temporale);
- Efficienza in termini di memoria occupata **S** (detta anche complessità spaziale)

La misura di queste efficienze dipende da moltissimi fattori: CPU del computer, dal compilatore usato, dalla architettura del computer (numero di processori, frequenza di clock, ecc). Bisogna riferirsi ad un modello astratto di computer (mono-processore ad accesso casuale ed un unico tipo di memoria di dimensioni illimitate)

In questo modello ogni istruzione è eseguita in successione ed in un tempo costante. Una cella di memoria può contenere un dato numerico di qualsiasi valore

Complessità Temporale T

Accenniamo alla **Complessità Temporale T** perché riveste più importanza. Considerazioni analoghe valgono per la complessità spaziale S .

T dipende naturalmente dalla particolare istanza dell'input e questa è una fra tutte quelle possibili. Ne segue che se vogliamo confrontare due algoritmi, bisogna farlo in relazione all'input → Nozione di **dimensione** dell'istanza:

Ad ogni istanza si associa un numero naturale **n** (che diciamo dimensione) che intuitivamente rappresenta la quantità di informazione contenuta nel dato.

Ad esempio un intero M ha dimensione n uguale a $1 + \log_2 M$, cioè al numero di cifre necessarie a rappresentare M in notazione binaria, la dimensione di un vettore è pari al numero delle sue componenti, ecc.

Notiamo che $T(n)$ dipende anche da qualche particolarità del set di dati in input

Complessità Temporale T

Può capitare che con due istanze diverse x e x' e della stessa dimensione n , le efficienze temporali siano diverse : $T_x(n)$ diverso da $T_{x'}(n)$. Ad esempio un algoritmo che deve ordinare un vettore di 100 componenti impiegherà tempi diversi a seconda che il vettore sia già ordinato oppure no.

Per questo motivo nell'analisi di $T(n)$ si considerano i tre casi possibili:

- il **caso migliore** (configurazione dell'input) col minimo tempo di esecuzione
- il **caso peggiore** col massimo tempo di esecuzione
- il **caso medio** che corrisponde al comportamento medio di $T(n)$ al variare della configurazione iniziale.

A seconda dell'algoritmo si sceglie il caso (o i casi) da considerare. In talune applicazioni si considera il caso peggiore perché è quello che si verifica più frequentemente. In altri casi questo caso è troppo pessimistico.

Il caso medio assume una distribuzione uniforme sulle istanze (ipotesi talvolta grossolana). Il caso migliore spesso è troppo ottimistico.

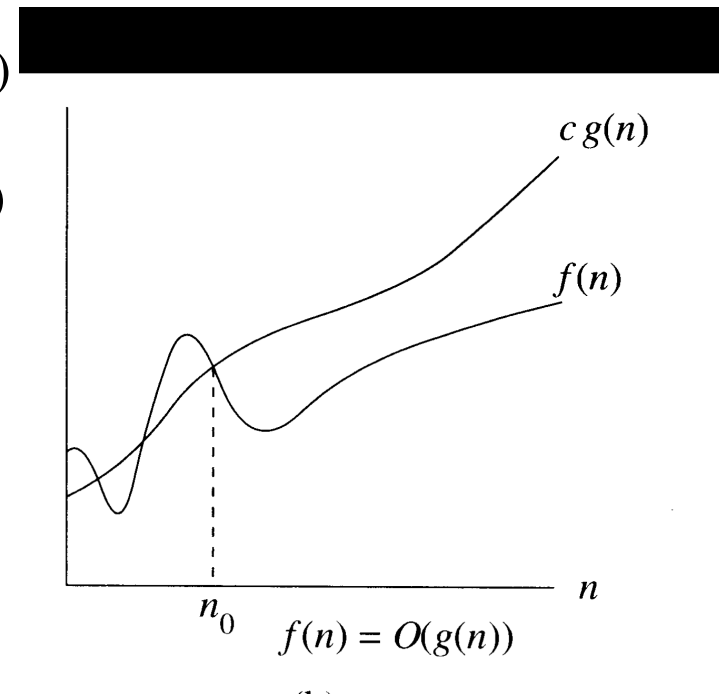
Comportamento Asintotico

Spesso è utile avere una idea della complessità temporale di un algoritmo senza fare riferimento alla dimensione n dei dati in input.

Si considerano gli andamenti di $T(n)$ nel caso limite di n sufficientemente grande. Una notazione asintotica molto importante è quella $O(n)$ (o grande di n):

Una funzione $f(n)$ appartiene all'insieme $O(g(n))$ se esistono due costanti positive c e n_0 e per ogni $n \geq n_0$ per cui $0 \leq f(n) \leq c g(n)$

Quindi per dimensioni n del dato in input superiori ad una certa dimensione n_0 la funzione $g(n)$ maggiora la funzione $f(n)$



Confronto di Algoritmi di Diverso T

Per avere una idea del comportamento asintotico, supponiamo che il computer esegua ogni operazione elementare in un *microsecondo* e consideriamo sei algoritmi con diversa complessità T e che operino su istanze di varie dimensioni. In tabella i tempi di esecuzione:

Complessità	$n = 10$	$n = 20$	$n = 50$	$n = 100$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
n	$10\mu s$	$20\mu s$	$50\mu s$	$0,1ms$	$1ms$	$10ms$	$0,1s$	$1s$
$n \log_2 n$	$33,2\mu s$	$86,4\mu s$	$0,28ms$	$0,6ms$	$9,9ms$	$0,1s$	$1,6s$	$19,9s$
n^2	$0,1ms$	$0,4ms$	$2,5ms$	$10ms$	$1s$	$100s$	$2,7h$	$11,5g$
n^3	$1ms$	$8ms$	$125ms$	$1s$	$16,6mn$	$11,5g$	$31,7a$	$\approx 300c$
2^n	$1ms$	$1s$	$35,7a$	$\approx 10^{14}c$	∞	∞	∞	∞
3^n	$59ms$	$58mn$	$\approx 10^8c$	∞	∞	∞	∞	∞

a = anno, c = secolo, ∞ > millennio

Confronto di Algoritmi di Diversa T

Complessità in tempo	Max Dimensione
n	6×10^7
$n \log_2 n$	28×10^5
n^2	77×10^2
n^3	390
2^n	25

Dimensioni massime processabili in un minuto da cinque algoritmi di diversa complessità temporale T.

Complessità in tempo	Max dim. su C1	Max dim. su C2
n	d_1	$M \cdot d_1$
$n \lg n$	d_2	$\approx M \cdot d_2$ (per $d_2 \gg 0$)
n^2	d_3	$\sqrt{M} \cdot d_3$
2^n	d_4	$d_4 + \lg M$

Due computer C1 e C2 con velocità di calcolo M1 e M2 ($M2 = M \times M1$).

Dimensioni massime processabili . La maggiore potenza del computer è sfruttata pienamente da algoritmi lineari (n) o quasi lineari ($n \log n$), qualche vantaggio per quelli con complessità dell'ordine n^2 . Cambiamento tecnologico di fatto **ininfluente** per algoritmi a complessità esponenziale (2^n).

Algoritmi di Ricerca

Data una sequenza di elementi, controllare se un elemento (chiave) fa parte della sequenza oppure no. La sequenza possiamo realizzarla con un array e facciamo la scansione utilizzando un indice.

La sequenza di n oggetti confrontabili $(a_1, a_2, \dots, a_{n-1})$ è ordinata se $a_i \leq a_{i+1}$ per ogni $i = 0, 1, \dots, n-2$

Con sequenze non ordinate la ricerca è di tipo **lineare** e ciò richiede che la chiave venga confrontata con al più tutti gli elementi della sequenza. La ricerca si ferma al momento in cui l'elemento è trovato .

Con sequenze ordinate la ricerca sfrutta il fatto che la sequenza è ordinata. In questi casi si usano algoritmi di **ricerca binaria** (molto più efficienti)

Ricerca binaria

Si confronta la chiave con l'elemento centrale della sequenza.
Se sono uguali la ricerca termina.

Se la chiave è maggiore dell'elemento centrale allora la ricerca continua solo nella sottosequenza a destra. Se invece la chiave è minore dell'elemento centrale la ricerca continua solo nella sottosequenza a sinistra.

Il meccanismo viene ripetuto. Ad ogni passo il numero degli elementi si “dimezza” e la ricerca termina quando il numero di elementi nella sottosequenza è uno o zero.

Il caso peggiore si ha quando la chiave non si trova nella sequenza. Nel caso peggiore si eseguono **$\log N$** passi (N numero di elementi nella sequenza).

Algoritmi di Ordinamento

Data una sequenza di N elementi a priori non ordinati, riscrivere la sequenza con tutti gli elementi ordinati. Si tratta di permutare gli elementi per ordinarli.

Consideriamo due esempi di algoritmi di ordinamento con complessità computazionale di ordine N^2 (cioè di algoritmi che richiedono un numero di passi dell'ordine di N^2):

Selection Sort e **Bubble Sort**

e due esempi di algoritmi con complessità computazionale dell'ordine di $N \log N$):

Quick Sort e **Merge Sort**

Selection Sort

Idea di base:

- 1) Si analizza la sequenza S di N elementi e si cerca il minimo. Trovato il minimo questo viene scambiato e messo al primo posto $S(1)$.
- 2) Questo procedimento viene ripetuto con i restanti $N-1$ elementi $S(2)-S(N)$. Trovato il nuovo minimo questo viene messo al posto $S(2)$ e il procedimento continua con i restanti $S(3)-S(N)$.
- 3) E così via sino a raggiungere l'elemento $S(N)$.

Algoritmo particolarmente semplice. Efficienza asintotica $O(N^2)$

Bubble Sort

Idea di base:

- 1) Si confrontano due elementi contigui della sequenza S. Se sono ordinati non si fa nulla e si passa alla coppia di elementi successivi. Se non sono ordinati i due elementi vengono scambiati, ordinandoli e così via.

Alla fine delle coppie il numero più grande sarà alla testa (o alla coda) della sequenza.

- 2) La procedura viene ripetuta per i restanti N-1 elementi.
- 3) E così via iterativamente per tutti gli altri elementi.

Efficienza asintotica **$O(N^2)$**

Bubble Sort esempio

Ora vediamo e commentiamo un esempio di codice per gli algoritmi di ordinamento il Bubble sort e il Merge sort.

Il codice si trova in Applicazioni-Web/Algoritmi

Scaricatelo e fate qualche prova!

Bubble Sort : esempio

```
#include <iostream>
#include <iomanip>
using namespace std;

//ritorna p, in ordine crescente
void Order (int * p, int * q) {
    int temp;
    if(*p>*q)
    {temp=*p;
    *p=*q;
    *q=temp;
    }
}

//Ordinamento (bubble) della lista di interi a[]
void Bubble(int * a, int n) {
    int i,j;
    for (i=0; i<n; i++)
        for (j=n-1; i<j; j--)
```

Bubble Sort : esempio

```
Order(&a[j-1], &a[j]);}
```

```
int main() {  
int i, n(15);  
int a[] = {13, -86, 27, -138, 266, -92, -3, 22, -77, 2 45, -36, 12, 27, -38, 200};  
cout << " Lista iniziale di interi da ordinare"<<endl;  
for(i=0; i<n; i++) cout << a[i] << setw(5);  
cout << endl;  
Bubble(a,n);  
cout << "Lista ordinata in ordine crescente"<< endl;  
for(i=0; i<n; i++) cout << a[i] << setw(5);  
cout << endl;  
return 0;}
```

```
=====
```

```
g++ -o bubble bubble.cc  
./bubble
```


Bubble Sort: esempio

Lista iniziale di interi da ordinare

13 -86 27 -138 266 -92 -3 22 -77 245 -36 12 27 -38 200

Lista ordinata in ordine crescente

-138 -92 -86 -77 -38 -36 -3 12 13 22 27 27 200 245 266

Quick Sort

- ❑ Si sceglie a caso un elemento della sequenza S , $x=S(J)$ (che diciamo pivot)
- ❑ Si spostano a sinistra di x (quindi con indice minore di J) tutti gli elementi più piccoli di x e a destra di x (quindi con indice maggiore di J) tutti quelli maggiori.

Per esempio prendiamo la sequenza :

13 36 4 21 48 2 6 **34** 78 68 1 31 e scegliamo il
34 come pivot

Riorganizziamo la sequenza così:

13 4 21 2 6 1 31 **34** 36 48 78 68

Ora consideriamo le due sottosequenze (degli elementi a sinistra di 34 e a destra di 34) e in ognuna scegliamo un pivot per esempio 21 e 48.

Quick Sort

13 4 **21** 2 6 1 31 **34** 36 **48** 78 68

Ora ripetiamo la procedura per i due nuovi pivot, ottenendo:

134 2 6 1 **21** 31 **34** 36 **48** 78 68

Ripetiamo la stessa procedura sino ad avere sottosequenze di due elementi. Si confrontano questi due elementi ordinandoli (se non lo sono). Alla fine si ottiene

1 2 4 6 15 21 31 34 36 48 68 78

$O(N \log N)$ nel caso medio

$O(N^2)$ nel caso peggiore

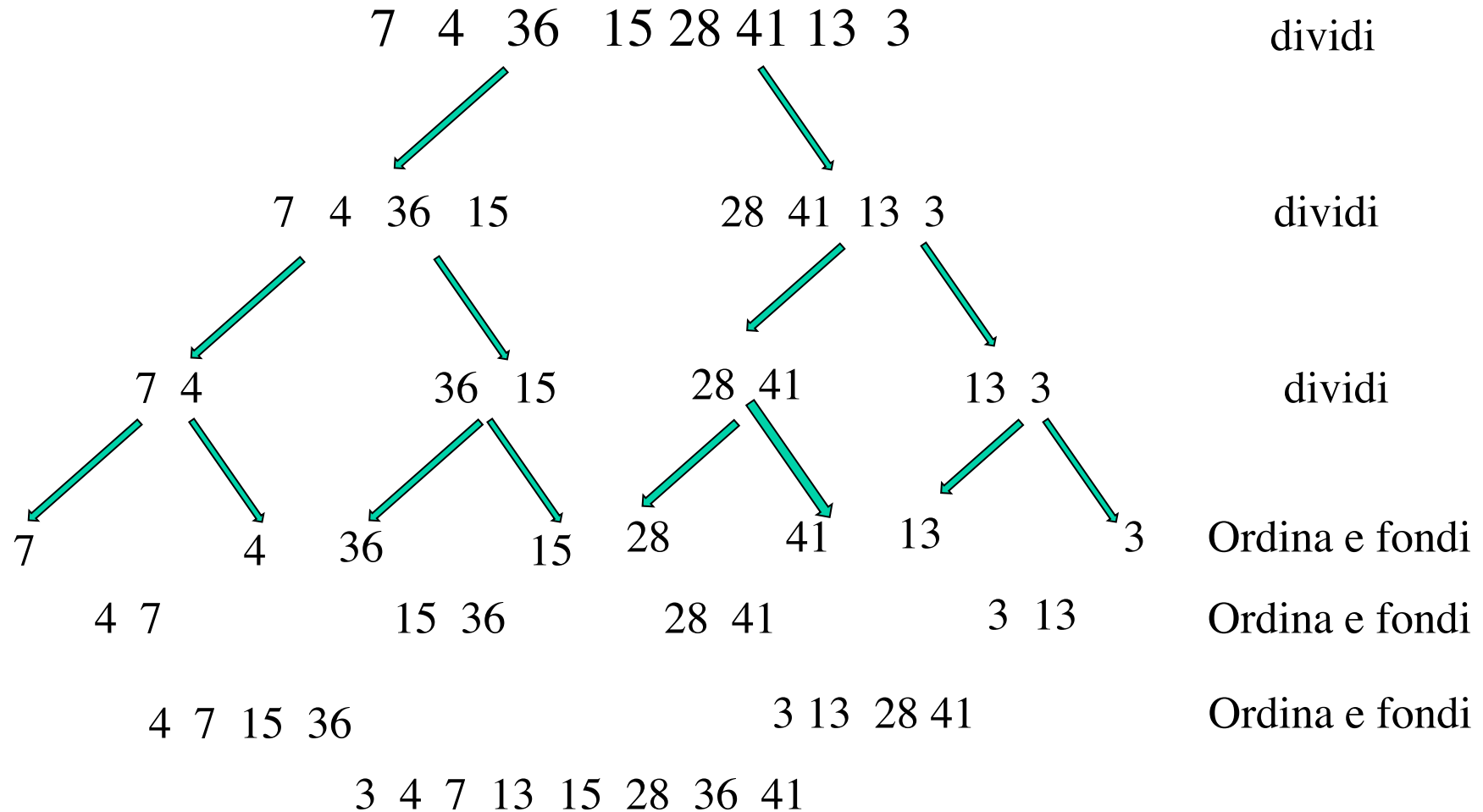
Merge Sort

Idea di base:

- 1 dividi la sequenza in due sottosequenze;
- 2 dividi ogni sottosequenza in due sottosequenze ;
- 3 ecc, applicando merge sort in modo ricorsivo;
- 4 quando si arriva a sottosequenze ordinate, queste vengono fuse.

Merge Sort

Esempio



Efficienza asintotica **$O(N \log N)$**

Merge Sort : esempio

```
#include <iostream>
#include <iomanip>
#define MAXSIZE 1024
using namespace std;
//combina due array ordinati a[] e b[] di dimensioni m e n in un array ordinato di
// dimensione m + n
void Merge (int *a, int *b, int *c, int m, int n) {
int i=0,j=0,k=0;
while(i<m && j<n) if (a[i]<b[j]) c[k++]=a[i++];
else c[k++]=b[j++];

// confronta il primo elemento dell'array a col primo elemento dell'array b; se è minore
//allora l'elemento dell'array a è spostato nell'array c e l'indice di a è aumentato di 1
//altrimenti è l'elemento di b ad essere spostato in c e l'indice dell'array b è aumentato
//di uno. Il processo continua sino a che un array resta vuoto. Gli elementi dell'altro
//array vengono messi in coda all'array c ed il merge è completato
```

Merge Sort : esempio

```
while(i<m)  c[k++]=a[i++];
```

```
while(j<n)  c[k++]=b[j++];
```

```
} // end Merge()
```

```
//usa la funzione Merge() per ordinare un array key[] di dimensione  n
```

```
//in questa versione si suppone che n sia multiplo di 2
```

```
void MergeSort (int *key,int n,int *error) {
```

```
int j,k,m, w[MAXSIZE];
```

```
//w è uno spazio di lavoro
```

```
for (m=1; m<n; m*=2) ;
```

```
if (n==m && n<=MAXSIZE) //n è una potenza di 2  <= MAXSIZE
```

```
for (k=1; k<n; k*=2) {
```

```
for (j=0; j<n-k; j+=2*k)
```

```
    Merge(key+j,key+j+k,w+j,k,k);
```

```
for (j=0; j<n; j++) //copia w in key
```

```
key[j] = w[j];
```

```
}
```

```
else {if(n!=m) { cout << " Error #1: size n of array not a power of 2"<< endl;
```

Merge Sort : esempio

```
else {if(n!=m) { cout << " Error #1: size n of array not a power of 2"<< endl;
*error=1;
}
if(n>MAXSIZE) {
cout << " Error #2: size of array too big" << endl;
*error=2;
}
}
} end //MergeSort()
```

```
int main() {
int error=0,i,n=32;  int a[] = {23,-129,-15,
-37,15,8,-39,231,0,371,-85,44,-22,90,1,-123,-109,-159,
39,-27,8,-19,211,0,271,-65,24,-72,-90,111,88,-99};
cout << "Lista iniziale"<<endl;
for (i=0; i<n; i++) cout << a[i] << setw(5);
cout << endl;
```


Merge Sort: esempio

```
MergeSort (a, n, &error);  
    if(error == 0) {  
        cout << "Lista ordinata in ordine crescente " << endl;  
        for (i=0; i<n; i++)  
            cout << a[i] << setw(5);  
    }  
    cout << endl;  
    return 0;  
} //end main()
```

=====

g++ -o merge merge.cc

./merge

Lista iniziale

23 -129 -15 -37 15 8 -39 231 0 371 -85 44 -22 90 1 -123 -109 -159
39 -27 8 -19 211 0 271 -65 24 -72 -90 111 88 -99

Lista ordinata in ordine crescente

-159 -129 -123 -109 -99 -90 -85 -72 -65 -39 -37 -27 -22 -19 -15 0 0 1
8 8 15 23 24 39 44 88 90 111 211 231 271 371
