LA COMPLESSITA' DEGLI ALGORITMI

Tra i problemi che ammettono soluzione esistono problemi "facili" e "difficili".

Teoria della complessità (anni '70):

- complessità di un problema;
- valutazione dell'efficienza degli algoritmi.

Qualsiasi programma richiede ***spazio di memoria*** e ***tempo di calcolo***. Ci concentriamo su quest'ultimo per valutare la ***complessità dei programmi***.

Esempio:

Moltiplicazione di due matrici quadrate n×n di interi:

$$C = A \times B$$

Per calcolare C[i,j] si eseguono 2n letture, n moltiplicazioni, n-1 addizioni e 1 scrittura.
Per calcolare C: n^2*(2n letture, n moltiplicazioni, n-1 addizioni ed 1 scrittura):

$2n^3$	letture
n^3	moltiplicazioni
$n^2*(n-1)$	addizioni
n^2	scritture

$$time_{Alg(C=A \times B)}(n) = 2n^3 + n^3 + n^2*(n-1) + n^2$$

Dato un algoritmo A la complessità viene determinata contando il numero di operazioni aritmetiche e logiche, accesso ai file, letture e scritture in memoria, etc.

Ipotesi semplificativa:

Tempo impiegato proporzionale al numero di operazioni eseguite (ciascuna a costo unitario)

Non ci si riferisce a una specifica macchina.

Valutare la complessità degli algoritmi ci consente di scegliere tra loro quello più efficiente (a minor complessità).

Il tempo impiegato per risolvere un problema dipende sia dall'algoritmo utilizzato che dalla ***"dimensione" dei dati*** a cui si applica l'algoritmo.

Oggetto di ingresso:	Dimensione:
vettore	n elementi
matrice	"
matrice quadrata	" "
matrice quadrata	n righe
lista	*n elementi*
albero	*n elementi; altezza*

La ***complessità*** dell'algoritmo viene espressa in funzione della ***dimensione delle strutture dati***.

$time_{Alg(C=A \times B)}(n)=2n^3+n^3+n^2*(n-1)+n^2=4*n^3$

$time_{Alg(P)}(n)=2^n$

Ordini di grandezza:

n	log_2 n	n*log_2 n	n^2	n^3	2^n
2	1	2	4	8	4
10	3.322	33.22	10^2	10^3	$> 10^3$
10^2	6.644	664.4	10^4	10^6	$\gg 10^{25}$
10^3	9.966	9996.0	10^6	10^9	$\gg 10^{250}$
10^4	13.287	1328.7	10^8	10^{12}	$\gg 10^{2500}$

Avendo a disposizione un elaboratore che esegue 10^3 operazioni al secondo, l'algoritmo risolutivo del problema P con ingresso di dimensione:

10	impiega	1 sec	
20	"	1000 sec	(17 min)
30	"	10^6 sec	(>10giorni)
40	"	10^9 sec	(>>10 anni)

Comportamento asintotico: notazione O

Individuare con esattezza $time_A(n)$ è spesso molto difficile.

Spesso è sufficiente stabilire il comportamento asintotico della funzione quando le dimensioni dell'ingresso tendono ad infinito (***comportamento asintotico dell'algoritmo***).

Definizione:

Un algoritmo (programma) ha costo O(f(n)) (o complessità O(f(n))) se esistono opportune costanti a,b, n' tali che:

$time_A(n) < a*f(n)+b$ per ogni n>n'.

O(f(n)), limite superiore al comportamento asintotico di una funzione.

Esempi:

$3*n^2+4*n+3 = O(n^2)$
perchè: $3*n^2+4*n-3 <= 4*n^2$ per ogni n>3.

$time_{Alg(A=B \times C)}(n)=4*n^3 = O(n^3)$

Attraverso la notazione O( ), gli algoritmi vengono divisi in ***classi***, ponendo nella medesima classe tutti quelli la cui ***complessità è dello stesso ordine di grandezza***.

Si hanno così algoritmi (funzioni) di complessità di ordine:

costante	1, ...
sotto-lineare	$\log n$, n^k con $k<1$
lineare	n
polinomiale	$n*\log n$, n^2, n^3, ... n^k con $k>1$
esponenziale	c^n, n^n, ...

Esempi:

$\log n$	⇒	complessità logaritmica	$O(\log n)$
$c1*n+c2$	⇒	complessità lineare	$O(n)$
n^2+n	⇒	complessità quadratica	$O(n^2)$
n^k+c3*n	⇒	complessità polinomiale	$O(n^k)$
2^n+n	⇒	complessità esponenziale	$O(k^n)$

Dato un problema P e due algoritmi A1 e A2 che lo risolvono siamo interessati a determinare quale ha complessità minore (è “il migliore”).

Esempio:

$time_{A1}(n)=3n^2+n$

$time_{A2}(n)=n^2+10n$

Per $n>=5$, $time_{A2}(n)<time_{A1}(n)$. La complessità di A2 è minore per ingressi con dimensione maggiore di 5.

Definizione:

Dato un problema P e due algoritmi A e B che lo risolvono con complessità $time_A$ e $time_B$, diciamo che A è ***migliore*** di B nel risolvere P se:

(1) $time_A=O(time_B)$
(2) $time_B$ non è $O(time_A)$

Esempio (cont.):

Nell'esempio, $time_{A1}=O(time_{A2})$ e viceversa. Sia A1 che A2 hanno comportamento asintotico quadratico ($O(n^2)$). Sono ***equivalenti*** dal punto di vista del costo computazionale.

Hanno costo diverso (B2 è migliore di B1), invece:

$time_{B1}(n)=3n^2+n$

$time_{B2}(n)=n*\log n$

Comportamento asintotico: notazione Ω

Definizione:

Un algoritmo (programma) ha costo $\Omega(g(n))$ (o complessità $\Omega(g(n))$) se esiste una opportuna costante positiva c tale che:

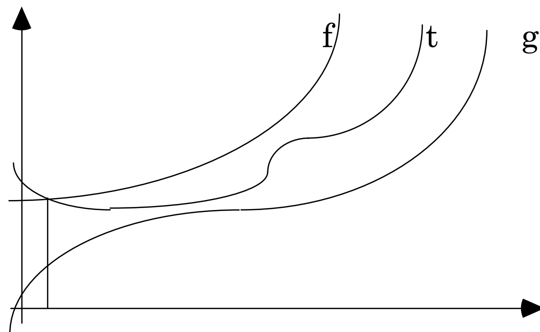
$time_A(n) > c*g(n)$

per un numero infinito di valori di n.

Esempio:

$3n^2+n = \Omega(n^2)$ (ma anche $\Omega(n)$ e $\Omega(n*\log n)$)

$\Omega(g(n))$, rappresenta un limite inferiore al comportamento di una funzione.



$t(n)=O(f(n))$ $t(n)=\Omega(g(n))$

Si ha una valutazione esatta del costo di un algoritmo quando le due delimitazioni coincidono.

Delimitazioni alla complessità di un problema:

Un problema ha ***delimitazione superiore*** $O(f(n))$ alla sua complessità se ***esiste almeno un*** algoritmo di soluzione per il problema con complessità $O(f(n))$.

Un problema ha ***delimitazione inferiore*** $\Omega(g(n))$ alla sua complessità se è possibile dimostrare che ***ogni*** algoritmo di soluzione per il problema ha complessità almeno $\Omega(g(n))$.

Un algoritmo di soluzione di un problema P è ***ottimale*** quando l'algoritmo ha complessità $O(f(n))$ e la delimitazione inferiore alla complessità del problema è $\Omega(f(n))$.

Problema con complessità ***lineare*** quando ogni algoritmo che lo risolve ha complessità $O(n)$ e $\Omega(n)$.

Problema con complessità ***polinomiale***: $O(n^k)$ e $\Omega(n^k)$ con $k>1$.

Problema intrattabile**:** è un problema risolubile, ma per il quale non esiste alcun algoritmo con complessità polinomiale che lo risolve (ad esempio, problema del commesso viaggiatore).

Istruzione dominante:

Permette di semplificare in modo drastico la valutazione della complessità.

Definizione:

Dato un algoritmo (o programma) A il cui costo di esecuzione è t(n), un'istruzione di A è detta ***dominante*** quando , per ogni intero n, essa viene eseguita (nel caso di input con dimensione n) un numero d(n) di volte tale che:

$$t(n) < a\, d(n) + b$$

per opportune costanti a, b.

Viene eseguita un numero di volte proporzionale al costo dell'algoritmo.

Se esiste un'istruzione dominante, la complessità dell'algoritmo è O(d(n)).

Esempio:

Ricerca esaustiva di un elemento in un vettore di interi di dimensione N.

```
#define N 15
typedef int vettore[N];
```

Viene realizzata dalla seguente funzione:

```
int ricerca (vettore vet, int el)
{int  i=0;
 int Trovato=0;
 while ((i<N)&&(!Trovato))       /* (1) */
   { if (el==vet[i]) Trovato=1;  /* (2) */
     i++;}
 return Trovato;
}
```

(1) e (2) istruzioni dominanti eseguite N+1 e N volte (caso peggiore)

Il costo è lineare.

ALGORITMI DI RICERCA SU VETTORI:

Esempio 1:

Ricerca del valore minimo e massimo di un vettore

```
#include <stdio.h>
#define N 15
typedef int vettore[N];

   /* dichiarazione di due funzioni */
int minimo (vettore vet);
int massimo (vettore vet);

main ()
{int i;
 vettore a;

 printf ("Scrivi %d numeri interi\n", N);
 for (i = 0; i < N; i++)
            { scanf ("%d", &a[i]); }
 puts ("L'insieme dei numeri è: ");
 for (i = 0; i<N; i++)
       { printf(" %d",a[i]); }
 puts("\n");
 printf ("Il minimo vale %d e il
          massimo è %d\n",
          minimo(a), massimo(a));
}
```



```
int minimo (vettore vet)
{int  i, min;
 for  (min = vet[0], i = 1; i < N; i ++)
      {if (vet[i]<min) /* istr. dom. */
             min = vet[i]; }
 return min;
}
```

```
int massimo (vettore vet)
{int  i, max;
 for (max = vet[0], i = 1; i < N; i ++)
    {if (vet[i]>max) /* istr. dominante*/
             max=vet[i];}
   return max;
}
```

Per la ricerca sia del minimo sia del massimo, l'istruzione dominante viene eseguita N-1 volte.

Entrambi gli algoritmi hanno un costo O(N), se N è la dimensione del vettore.

Esempio 2:

Ricerca esaustiva di un elemento in un vettore

```
#include <stdio.h>
#define N 15
typedef int vettore[N];
typedef enum {falso,vero} boolean;
main ()
{boolean ricerca (vettore vet, int el);
 int i;
 vettore a;
 printf ("Scrivi %d numeri interi\n", N);
 for (i = 0; i < N; i++)
     scanf ("%d", &a[i]);
 scanf("\n ");
 scanf ("Valore da cercare: %d",&i);
 if (ricerca(a,i)) printf("\nTrovato\n");
 else printf("\nNon trovato\n");
}

boolean ricerca (vettore vet, int el)
{int  i=0;
 boolean T=falso;
 while ((i<N)&&(T==falso)) /* istr.
                              dominante */
   { if (el==vet[i]) T=vero;
     i++;}
 return T;
}
```


Molto spesso il costo dell'esecuzione di un programma (di un algoritmo) dipende non solo dalla dimensione dell'ingresso, ma anche dai particolari valori dei dati in ingresso.

È possibile distinguere diversi casi: caso migliore, caso peggiore, caso medio.

Di solito la complessità viene valutata nel ***caso peggiore*** (e talvolta nel ***caso medio***).

Esempio 2 (cont.):

Per la ***ricerca sequenziale*** in un vettore il costo dipende dalla posizione dell'elemento cercato.

Caso migliore: l'elemento è il primo del vettore. (1 confronto)

Caso peggiore: l'elemento è l'ultimo o non è presente: l'istruzione dominante è eseguita N volte (N dimensione del vettore). Il costo è ***lineare***.

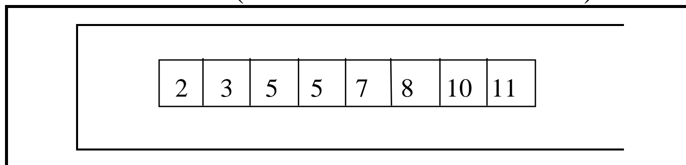
Caso medio: ciascun elemento sia equiprobabile

$\Sigma_{(i=1..N)}$ Prob(el(i))*i=$\Sigma_{(i=1..N)}$ (1/N)*i=(N+1)/2

Esempio 3:
Sapendo che il vettore è ***ordinato***, la ricerca può essere ottimizzata (ricerca binaria).

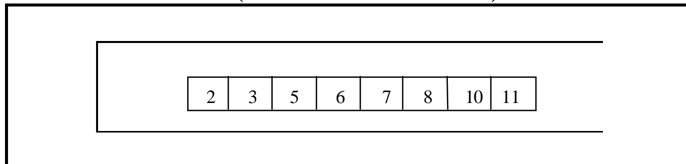
Vettore ordinato:
Esiste una relazione d'ordine totale sul dominio degli elementi del vettore e:

∀i,j: i<j si ha V[i]<=V[j]
(***in senso non decrescente***)



se invece:

∀i,j: i<j si ha V[i]<V[j]
(***in senso crescente***)



In modo analogo si definiscono l'ordinamento in senso ***non crescente*** e ***decrescente***.

Supponiamo di cercare un elemento in un vettore ordinato in senso non decrescente.

Esempio 3 (cont.):
Ricerca binaria su un vettore ordinato in senso non decrescente con componenti di indice da *first* a *last*.

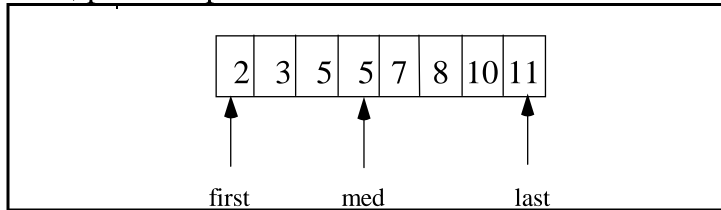
La tecnica di ***ricerca binaria*** rispetto alla ricerca esaustiva, consente di eliminare a ogni passo metà degli elementi del vettore.

Vettore con indici da *first* a *last*; indice mediano *med=(first+last)*/2
Si confronta l'elemento cercato *el* con quello mediano del vettore, V[*med*]. Se *el*==V[*med*], fine della ricerca (trovato=vero). Se no, si ripete nella prima sotto-parte o nella seconda.

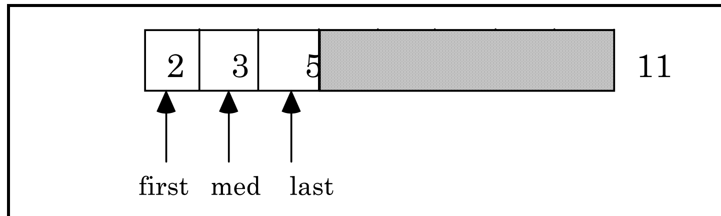
Ripeti fino a che il vettore ha almeno una componente (*first <=last*):

> Se *el*==V[*med*], fine della ricerca (trovato=vero);
>
> se *el*<V[*med*], ripeti la ricerca nella prima metà del vettore (indici da *first* a *med*-1);
>
> se *el*>V[*med*], ripeti la ricerca nella seconda metà del vettore (indici da *med*+1 a *last*).

Si cerchi 4, per esempio:

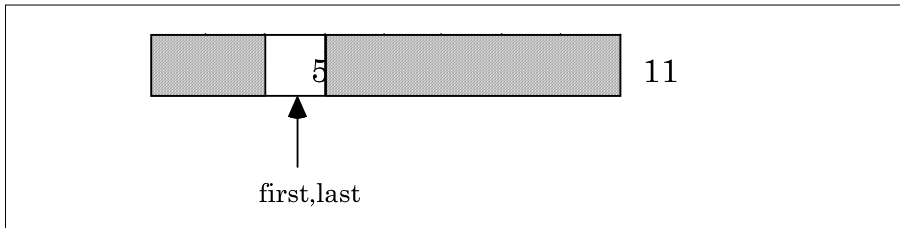



```
med= (first+last) / 2
el<V[med]
```




```
el>V[med]
```

Vettore ad una componente:




```
/* funzione ricerca binaria su un vettore
   di interi   */

typedef enum {falso, vero} boolean;

boolean ricerca_bin (vettore vet, int el,
                      int *pos)
{int   first=0,
       last=N-1,
       med=(first+last)/2 ;
 boolean T=falso;
 while ((first<=last)&&(T==falso))
                          /* istr. dom. */
   { if (el==vet[med])
          {T=vero; *pos=med;}
     else
        if (el < vet[med])  last=med-1;
        else   first=med+1;
     med = (first + last) / 2;
   }
 return  T;
}
```

Chiamata:

```
 if (ricerca_bin(a,i,&p))
   printf("\nTrovato in posizione
          %d\n", p);
 else printf("\nNon trovato\n");
```

Esempio 3 (cont.)

La ***complessità della ricerca binaria*** in un vettore, come per la ricerca sequenziale, dipende dalla posizione dell'elemento cercato.

Caso migliore:
l'elemento cercato è quello mediano nel vettore: 1 confronto

Caso peggiore:
l'elemento cercato è l'ultimo o non è presente nel vettore.

Il ciclo while è ripetuto finchè ci si riduce ad un vettore mono-dimensionale (first==last).

Poichè ad ogni passo di iterazione la dimensione del vettore dimezza, si hanno al più k passi (con k finito e proporzionale a $\log_2 N$):

Passo	n. elementi vettore	n. confronti
1	N	1
2	N/2	1
3	N/4	1
...		
k	$N/2^{(k-1)}$	1
k+1	1	1

$$\Sigma_{(i=1..k+1)}\ 1 = k+1 = \log_2 N+1$$

0 Esercitazione 4.1:

È dato un vettore di dimensione N+k contenente N numeri interi, ordinati in senso non decrescente.
Si supponga di ricevere uno alla volta k interi.

a) Si scriva il programma che li inserisce nel vettore, mantenendo l'ordine a ogni passo di inserimento.
b) Determinare le condizioni che generano il maggior numero di confronti tra elementi, in funzione di N e k.

```
#include <stdio.h>
#define dim 10
#define k 4
typedef int vettore[dim];
typedef enum {falso,vero} boolean;
int N=6;  /var. globale */

void main (void)
{ void inserisci (vettore vet, int el);
  int i, elem;
  vettore V;

 /* ciclo lettura vettore parziale */
  printf("Scrivi %d interi ordinati\n",N);
  for (i = 0; i < N; i++)
       scanf ("%d", &V[i]);
  scanf("\n ");

/* ciclo lettura k elementi */
  printf("Inserisci %d interi \n",dim-N);
  for (i = dim-k; i < dim; i++)
     { scanf ("%d", &elem);
       inserisci(V, elem); }
 printf("\n ");}
```

```
void inserisci (vettore vet, int el)
/* N elementi gia’ inseriti */
{int  i=0;
 boolean T=falso;
 int j;
 while ((i<N)&&(T==falso))       /* !T */
   { if (el<=vet[i]) T=vero;
     else i++;}
 if (T==vero)
   for(j=dim-2;j>=i;j--)
             vet[j+1]=vet[j];  /* shift */
 et[i]=el;
 N++;       /* aggiorna num. el. inseriti */
}
```

Il caso peggiore (<u>per numero di confronti</u>) si presenta quando ogni singolo elemento da inserire deve essere inserito nella prima locazione libera del vettore.

Esempio (N=6):

5	6	7	8	9	10	-	-	-	-

e sequenza di ingresso: 11   12   13   14

Vengono eseguiti due confronti per il test del ciclo while e un confronto (test if).

Passo	n. elementi vettore	n. confronti
1	N	3*(N) +2
2	N+1	3*(N+1)+2
3	N+2	3*(N+2)+2
...		
k	N+k-1	3*(N+k-1)+2

$$\sum_{(i=1..k)} 3*(N+i) +2 =$$

$$=2*k+3*N*k + 3*\sum_{(i=1..k)} i =$$

$$= 2*k+3*N*k + 3* k*(k+1)/2$$