COMPLESSITÀ COMPUTAZIONALE

COSA SI INTENDE PER COMPLESSITÀ?

La Complessità Temporale, Spaziale, di Input/Output e di Trasmissione

COMPLESSITÀ TEMPORALE

Ininfluenza della velocità della macchina per molte importanti classi di algoritmi

IL MODELLO DI COSTO

- Definizione di dimensione dell'input
- Definizione di istruzione di costo unitario (passo base)
- Esempi di calcolo della complessità in numero di passi base
- Complessità nel caso migliore, nel caso medio e nel caso peggiore
- Complessità di programmi strutturati
- Complessità asintotica

COMPLESSITÀ TEMPORALE
Quanto tempo richiede l'esecuzione di un algoritmo?

COMPLESSITÀ SPAZIALE
Quanta occupazione di memoria richiede l'esecuzione di un algoritmo?

COMPLESSITÀ DI I/O (INPUT_OUTPUT)
Quanto tempo richiede la acquisizione (o il trasferimento) di informazioni da periferiche? (Memoria secondaria, tastiera, stampante,...)

Di particolare rilevanza è il tempo di accesso ad informazioni residenti in memoria secondaria. Rispetto al tempo di accesso a dati residenti in memoria centrale, l'accesso a dati in memoria secondaria è di circa sei ordini di grandezza superiore:

Memoria centrale: decine di nanosecondi (~ $10*10^{-9}$ sec)

Memoria di massa: decine di millisecondi (~ $10*10^{-3}$ sec)

Esiste un quarto criterio di complessità: la
COMPLESSITÀ DI TRASMISSIONE

☞ misura dell'efficienza di I/O di un algoritmo rispetto a "stazioni remote"
(altri computer, memorie fisicamente lontane, ecc.)

considereremo principalmente laCOMPLESSITÀ

TEMPORALE

SCOPO DELLO STUDIO DELLA COMPLESSITÀ TEMPORALE

Uno stesso problema può essere risolto in più modi diversi, cioè con algoritmi diversi.

Algoritmi diversi possono avere diversa complessità e quindi diversa efficienza.

Si supponga di avere a disposizione due algoritmi diversi per ordinare **n** numeri interi.

- il primo algoritmo riesce ad ordinare gli **n** numeri con n^2 istruzioni, il secondo con **n*log n** istruzioni.
- supponiamo che l'esecuzione di un'istruzione avvenga in un μsec (10^{-6} sec).

n operazioni	**n=10**	**n=10000**	**n=10^6**
n^2 operazioni	**0,1 msec**	**100 sec (1,5 minuti)**	**10^6 sec (~12 giorni)**
n*log n operaz.	**0,00003 sec**	**0,13 sec**	**19 sec**

Diverse classi di complessità possono avere comportamenti divergenti al variare della dimensione del problema

Quali fattori influenzano il tempo si esecuzione?

- ❑ L'algoritmo scelto per risolvere il problema
- ❑ La dimensione dell'input
- ❑ La velocità della macchina

Cerchiamo un modello di calcolo per la complessità temporale che tenga conto di

- ☞ Algoritmo
- ☞ Dimensione dell'input

Il miglioramento della tecnologia non riduce significativamente il tempo di esecuzione di molte importanti classi di algoritmi

Algoritmi di Ricerca (n operazioni)

Algoritmi di Ordinamento (n^2 operazioni)

Algoritmi Decisionali (2^n operazioni)

	Attuale Tecnologia	100 volte più veloce	1000 volte più veloce
Ricerca: n operazioni	t'	t'/100	t'/1000
Ordinamento: n^2 operazioni	t''	t''/10	t''/31,6
Scelta: 2^n operazioni	t'''	~t''' * (t'''- 6,6)	~t''' * (t'''- 9,9)

IL MODELLO DI COSTO

"dimensione dell'input" = cardinalità dei dati in ingresso.

A seconda del problema, per dimensione dell'input si indicano cose diverse:

- La grandezza di un numero (es.: problemi di calcolo)
- Quanti elementi sono in ingresso (es.: ordinamento)
- Quanti bit compongono un numero

Indipendentemente dal tipo di dati, indichiamo con "n" la dimensione dell'input.

DEFINIZIONE DI OPERAZIONE DI COSTO UNITARIO:

E' una operazione (statement) la cui esecuzione non dipende dai valori e dai tipi delle variabili.

- LETTURA scanf **()**
- SCRITTURA printf **()**
- ASSEGNAMENTO , OPERAZIONI ARITMETICHE PREDEFINITE, RETURN (**y=sqrt(x)+3; return x)**
- ACCESSO AD UN QUALUNQUE ELEMENTO DI UN ARRAY RESIDENTE IN MEMORIA CENTRALE

```
A[i] = A[1];
x =A[10000];
```

- VALUTAZIONE DI UNA QUALSIASI ESPRESSIONE BOOLEANA

```
if ((x>100) || ((j<=n) &&
                    (B == true)))
{...
```

Giustificazione delle definizioni precedenti:

La memoria centrale è detta RAM (Random Access Memory): l'accesso a qualsiasi cella avviene in tempo costante. Per constrasto, l'accesso sequenziale prevede un costo unitario soltanto per la cella *in sequenza* rispetto all'ultima cella letta.

Il costo del test di una condizione booleana composta di più condizioni booleane semplici è sempre minore o uguale a C volte il costo del test di una condizione semplice, dove C è una opportuna costante numerica, quindi, semplificando, consideriamo comunque un costo unitario.

Lo stesso vale per le operazioni aritmetiche composte.

Si indicherà una operazione di costo unitario con il termine *passo base.*

Non si considerano, in questa sede, operazioni di accesso ai file.

Un modello più sofisticato dovrebbe differenziare tra aritmetica intera e reale (floating point), tener conto di come il compilatore traduce le istruzioni di alto livello (compilatori ottimizzanti), considerare l'architettura di una specifica macchina.

ESEMPI DI CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE

```
1)    i=1;
      while (i <= n)  i=i+1;
```

Assegnamento esterno al ciclo: **`i=1`**

Ciclo while: si compone di un test **`(i<=n)`** e di un corpo costituito da una operazione di assegnamento **`(i=i+1)`**.

Per ogni test positivo si esegue un assegnamento.

Quindi:

Assegnamento esterno	**`1`**
Numero di test	**`n+1`**
Assegnamenti interni	**`1*n`**
Numero totale di passi base	**`2+2*n`**

```
2)    i=1;
      while (i <= n)
      {
         i=i+1;
         j=j*3+42;
      }
```

Il corpo del ciclo si compone di due passi base, quindi:	Assegnamento esterno:	**1**
	Numero di test:	**n+1**
	Assegnamenti interni:	**2*n**
	Numero totale di passi base:	**3*n+2**

```
3) i=1;
      while (i <= 2*n)
      {
         i=i+1;
         j =j*3+4367;
      }
```

Il test viene eseguito 2*n+1 volte ed il corpo è costituito da due passi base, quindi:	Assegnamento esterno:	**1**
	Numero di test:	**2*n+1**
	Assegnamenti interni:	**4*n**
	Numero totale di passi base:	**6*n+2**

4) Cicli Annidati

```
i=1;
while (i<= n)
{
 for (j=1;
  j <= n;J++)
    {

printf("CIAO!");

    }
  i=i+1;
  }
```

n+1 test

un ciclo for per ogni ripetizione while

una scrittura per ogni ripetizione for

un assegnamento per ogni while

Assegnamento esterno:	**1 +**
Test while:	**n+1 +**
Numero cicli while:	**n *(**
for:	**n +**
Corpo for:	**n +**
Assegnamento i:=i+1:	**1)**
Numero totale di passi base:	$2+2*n+2*n^2$

5) Attenzione: esprimere la complessità in funzione di n !!!

```
i=1;
while (i*i<= n)
   {
      i=i+1;
   }
```

Quanti test vengono eseguiti? Quante volte si cicla?

caso n=9:

```
i=1;
while (i*i<= 9)
   {i=i+1;}
```

i=1

i^2 = 1 <= 9 ? SI i:=2

i^2 = 4 <= 9 ? SI i:=3

i^2 = 9 <= 9 ? SI i:=4

i^2 = 16 <= 9 ? NO FINE

Si cicla 3 = $\sqrt{9}$ volte, eseguendo 4= $\sqrt{9}$ +1 test.

La complessità di questo blocco è dunque:

$1 + \sqrt{n} + 1 + \sqrt{n} = 2+2*\sqrt{n}$ passi base (con arrotondamento all'intero superiore)

☞ IL COSTO PUO' DIPENDERE DAL *VALORE* DEI DATI IN INGRESSO

Un tipico esempio è dato da:

```
   if (condizione)        if (condizione)
     {corpo                  {corpo 1}
istruzioni}                 else {corpo 2}
```

Nel primo caso il costo è il costo di un test più il costo di esecuzione del corpo istruzioni se la condizione è vera. Se la condizione è falsa, il costo è il solo costo del test.

Nel secondo caso il costo è dato dal costo del test più il costo del primo corpo se la condizione è vera. Se la condizione è falsa, il costo è il costo del test più il costo del secondo corpo istruzioni.

```
   for (i=1; i <=n; i++)
      {
      {aggiorna condizione}
      if (condizione) /* if annidato nel
ciclo */
            {corpo 1}  else {corpo 2}
     };
```

Anche in un caso di questo tipo il costo del ciclo dipende non solo dalla dimensione dei dati, ma anche dal loro valore: all'interno di ogni ciclo il costo del blocco <if then else> può variare a seconda del valore assunto dalla condizione.

6) Complessità nel caso migliore, nel caso peggiore e nel caso medio

Supponiamo di avere il seguente array A di quattordici elementi:

3	-2	0	7	5	4	0	8	-3	-1	9	12	20	5

La dimensione dell'input è la lunghezza dell'array (n=14 nell'esempio).

Cerchiamo l'elemento "8" con un algoritmo di ricerca lineare:

```
i=0;
while ((i <= 13) && (A[i] < > 8))
    {i=i+1;}
```

1 assegnamento + 8 test + 7 cicli = 16 passi base

Se avessimo cercato l'elemento "6"?

Poichè tale elemento non è presente nell'array, avremmo dovuto scorrere l'array per intero, con complessità:

1+ n+1 +n = 2+2*n passi base

(30 passi nell'esempio)

Per il problema della ricerca in un array non ordinato, bisogna distinguere due casi:

a) Si sa che l'elemento cercato è presente nell'array

b) Non si sa se l'elemento cercato è presente nell'array

Nella prima situazione, si possono distinguere tre casi:

- **Caso migliore**: l'elemento cercato è il primo. Il costo effettivo dell'algoritmo è 2.
- **Caso peggiore:** l'elemento cercato è l'ultimo dell'array. Il costo effettivo è 1 + 2*n
- **Caso medio:**

  Ipotesi di distribuzione uniforme: i valori siano disposti casualmente nell'array
  Poichè i valori sono distribuiti uniformemente, la probabilità di trovare l'elemento x cercato in posizione i nell'array mediante ricerca sequenziale è:

  P(x) = (1/n)*i

  dove 1/n è la probabilità di trovare un elemento fra n uniformemente distribuiti e i è la probabilità di trovare l'elemento all'i-esima ricerca.

  In totale, il numero medio di confronti da effettuare è:

  $\sum_{i=1}^{n} P(x) = (n+1)/2$

Nella seconda situazione, in cui non si sa se l’elemento cercato è presente nell’array, non si può assegnare un valore di probabilità come fatto sopra.

In questa situazione il caso migliore è la presenza dell'elemento nell’array, il "caso peggiore" è l'assenza. Il caso medio si può ricavare, se si dispone di una stima della probabilità di presenza, con una somma pesata del caso medio della prima situazione e del caso di assenza, che implica il costo di scansione dell'intero array

Già dai due casi visti nell'algoritmo di ricerca sequenziale si può capire che il concetto di "caso peggiore", "caso migliore" e "caso medio" può dipendere da:

- Problema ed Algoritmo
- Valori dei dati

Nel seguito verrà considerata la complessità nel caso peggiore, salvo diversa indicazione.

7) PROGRAMMI STRUTTURATI

Nel calcolo della complessità di un programma strutturato si deve:

- calcolare la complessità di ogni procedura e funzione
- per ogni chiamata a procedura/funzione, aggiungere la complessità della stessa al costo globale del programma

Esempio 7

```
void Stampastelle (int ns);
main ()
{    int n,m,j;
     printf("Quante stelle per riga?"); scanf("%d", &n);
     printf("Quante righe di stelle?"); scanf("%d", &m)
 for (j=1; j<=m; j++)     Stampastelle(n);
} \* main *\

void Stampastelle (int ns)
 {   int i;
     printf ("\n");
     for (i=1; i <= ns; i++)       printf ("*");
 }        \* Stampastelle *\
```

Complessità della procedura:

$1 + 2*ns$

Complessità del programma principale:

$4+m+m*(1+2*n) = 4+2*m+2*m*n$

NOTA:

Se in un algoritmo la dimensione dell'input è definita da più parametri, come in questo esempio, bisogna tenere conto di tutti.

Il costo di esecuzione di una procedura o funzione può dipendere dai parametri che le vengono dati in ingresso.

Una stessa procedura/funzione può essere chiamata, all'interno del programma principale, con dati diversi (input di dimensione diversa).

```
8)\* Parte dichiarativa come esempio 7 *\
   {        \* main *\
   printf ("Quante righe di stelle?");
scanf("%d", &m);
      for (j=1; j <= m; j++)
         Stampastelle(j);
    }    \* main *\
```

Alla j-esima iterazione, il costo di "Stampastelle" è 1+2*j, quindi la complessità è:

$2 + 2*m + m^2$

```
9) int potenza(int base, int esp);
   main () \* Genera le prime <n> potenze *\
   {int b,n,espo; \* di base <b> *\
   printf ("Scrivi la base");
   scanf("%d",&b);
   printf ("Quante potenze?");
   scanf("%d",&n);
   for (espo=1; espo <= n;
      espo++)
   printf ("%d",potenza(b,espo));
   } \* main *\
   int potenza(int base, int esp)
   {  int  i,ris;
      ris=1; for (i=1; i <=  esp;i++)
               ris=ris*base;
     return ris;}
```

Complessità:

La funzione "potenza" ha complessità: 2+2*esp

Main:	4+n+n+n
1°chiamata:	2+2*1+
2°chiamata:	2+2*2+
...	...
j-esima chiamata:	2+2*j+
...	...
n-esima chiamata:	2+2*n
totale:	$4+3*n+ 2*n +2*\sum j$
	$= 4+5*n+n + n^2$

OSSERVAZIONE

La complessità della *chiamata di procedura* dovrebbe tenere conto anche della copia di valori necessaria in caso di passaggio parametri per valore, dell'eventuale calcolo di espressioni dei parametri effettivi e delle operazioni interne necessarie per la gestione del record di attivazione. Alla luce di questo valutare la complessità del calcolo del fattoriale in versione iterativa e ricorsiva.

COMPLESSITÀ ASINTOTICA

Supponiamo di avere, per uno stesso problema, sette algoritmi diversi con diversa complessità.

Supponiamo che un passo base venga eseguito in un microsecondo (10^{-6} sec).

Tempi di esecuzione (in secondi) dei sette algoritmi per diversi valori di n.

dimensione / classe	n=10	n=100	n=1000	$n=10^6$
$\sqrt{n}$	$3*10^{-6}$	10^{-5}	$3*10^{-5}$	10^{-3}
n + 5	$15*10^{-6}$	10^{-4}	10^{-3}	1 sec
2*n	$2*10^{-5}$	$2*10^{-4}$	$2*10^{-3}$	2 sec
n^2	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
$n^2 + n$	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^3	10^{-3}	1 sec	10^5 (~1g)	10^{12} 300 secoli
2^n	10^{-3}	10^{14} secoli	------	-------

OSSERVAZIONI

- Per piccole dimensioni dell'input, osserviamo che tutti gli algoritmi hanno tempi di risposta non significativamente differenti.
- L'algoritmo di complessità esponenziale ha tempi di risposta ben diversi da quelli degli altri algoritmi (migliaia di miliardi di secoli contro secondi, ecc.)
- Ultima colonna ($n=10^6$):

Per grandi dimensioni dell'input, i sette algoritmi si partizionano nettamente in cinque classi in base ai tempi di risposta:

Algoritmo $\sqrt{n}$ ---------- frazioni di secondo
Algoritmo $n+5$, $2*n$ ---------- secondi
Algoritmo n^2, n^2+n ---------- giorni
Algoritmo n^3 ---------- secoli
Algoritmo 2^n ---------- ?

GLI "O" GRANDI

un criterio matematico per partizionare gli algoritmi in classi di complessità

Si dice che una funzione ***f(n)*** *è di ordine* ***g(n)*** *e si scrive*:

$$f(n) = \mathrm{O}(g(n))$$

se esiste una costante numerica **C** positiva tale che valga, salvo al più per un numero finito di valori di n, la seguente condizione:

$$f(n) \leq C * g(n)$$

[1]

- $2n+5=O(n)$ poichè $2n+5<=7n$ per ogni n
- $2n+5=O(n^2)$ poichè $2n+5<=n^2$ per $n>= 4$,
- $2n+5=O(2^n)$ poichè $2n+5<=2^n$ per $n>= 4$,

$n=O(2n+5)$? SI

$n^2=O(n)$? NO: $n^2 / n = n$ non limitata

$e^n =O(n)$? NO: $e^n / n >= n^2 / n = n$ non limitata

---

[1] cioè la funzione f(n)/g(n) è limitata, per g(n) non identicamente nulla, fatto che nel caso di costo di un algoritmo non si può verificare

ORDINAMENTO FRA GLI "O" GRANDI

È possibile definire un criterio di ordinamento fra gli "O" grandi.

DEFINIZIONE:

$f(n)$ è più piccola (di ordine inferiore) a $g(n)$ se valgono le due seguenti condizioni:

- $f(n) = O(g(n))$
- $g(n)$ *non è* $O(f(n))$

Esempi:

$\sqrt{n}$ è di ordine inferiore a $2n+5$ (e quindi a n)

$(2n+5) / \sqrt{n} = 2\sqrt{n} + 5/\sqrt{n} <= 2\sqrt{n} + 5$ non è limitata

n è di ordine inferiore a e^n

$n=O(e^n)$, e^n non è $O(n)$

DEFINIZIONE DI COMPLESSITÀ ASINTOTICA:

Si dice che $f(n)$ ha complessità asintotica $g(n)$ se valgono le seguenti condizioni:

a) $f(n)=O(g(n))$

b) $g(n)$ è la più piccola di tutte le funzioni che soddisfano la condizione (a)

ESEMPI:

$2n + 5$	complessità asintotica n (lineare)
$3n^2 + 5n$	complessità asintotica n^2 (quadratica)
$4*2^n$	complessità asintotica 2^n (esponenziale di base 2)
$3^n + 2n + 5^n$	complessità asintotica 5^n (esponenziale di base 5)
$2^n + 5n + n! + 5^n$	complessità asintotica (fattoriale)

DAL CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE AL CALCOLO DELLA COMPLESSITÀ ASINTOTICA

	numero passi base	complessità asintotica
esempio 2	$3n+2$	n
esempio 4	$2n^2+2n+2$	n^2
esempio 5	$2+2\sqrt{n}$	$\sqrt{n}$
esempio 7	$4+ 2m+ 2m*n$	$m*n$

In pratica, la complessità asintotica è definita dal blocco di complessità maggiore.
Non contano le costanti, nè additive, nè moltiplicative.

Esempi:

$7*3^n$ ha complessità 3^n (7 è costante moltiplicativa)

n^2+5 ha complessità n^2 (5 è costante additiva)

Nella funzione di complessità 4+2m+2m*n domina il termine "quadratico" m*n

OSSERVAZIONE

Le ipotesi semplificative del modello di costo introdotto ed il metodo di calcolo della complessità asintotica sono ipotesi molto forti.

Si pensi, ad esempio, ad un algoritmo di complessità asintotica n^2 che abbia costo, espresso in numero di passi base, $7n^2+2n+3$.

Nel nostro modello trascuriamo le costanti ed i termini di ordine inferiore ma, nelle applicazioni reali, è ben diverso che un algoritmo termini in un giorno oppure in più di una settimana!!! Basti pensare ad operazioni bancarie, ad analisi ospedaliere, ad esperimenti fisici automatizzati, ecc.

ALGEBRA DEGLI "O" GRANDI

Abbiamo visto due esempi di calcolo della complessità in numero di passi base per programmi a blocchi.

Definiamo ora, tramite l'algebra degli "O" grandi, un criterio per il calcolo della complessità asintotica di un programma strutturato.

In un programma a blocchi si possono presentare le due seguenti situazioni:

a) Blocchi in sequenza:

esempio:

```
i=1;
while (i<=n)
   {
    stampastelle(i);
    i=i+1;
    }
for (i=1; i <= 2n; i++)
     {
     scanf("%d", &num);

        }
```

1° blocco: while

2° blocco: for

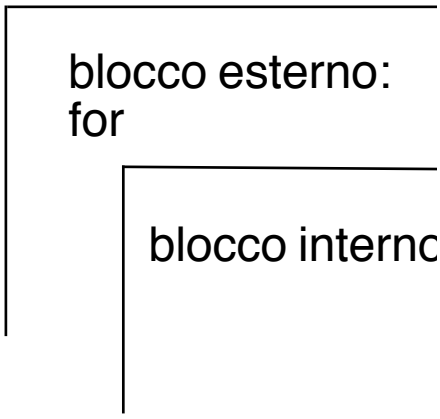
Sia *g1(n)* la complessità del primo blocco e *g2(n)* la complessità del secondo. La complessità globale è:

$$O(g1(n)+g2(n)) = O(max\{g1(n),g2(n)\})$$

→ la complessità di un blocco costituito da più blocchi in sequenza è quella del blocco di complessità maggiore

b) Blocchi annidati:

```
      for (i=1; i <= n;
i++)
       {
        scanf("%d", &j);
        printf(("%d",
j*j);
        do {
          scanf("%d",
&num);
          j = j+1;
           }
        while (j<=n)
       }
```



Sia g1(n) la complessità del blocco esterno e g2(n) la complessità di quello interno. La complessità globale è:

$$O(g1(n)*g2(n)) = O(g1(n))*O(g2(n))$$

→ la complessità di un blocco costituito da più blocchi annidati è data dal prodotto delle complessità dei blocchi componenti. Con il concetto di "O" grande e le due operazioni definite dall'algebra degli "O" grandi si può calcolare la complessità di un qualsiasi algoritmo strutturato.

```
# define  MAXN  10000; /* massimo numero
di elementi */
int Int_Array[MAXN]   /* dichiarazione
array  */

void CaricaArray(int L);
/* Carica L valori in array (L<=MAXN deciso dall'utente) */

int RicercaLineare(int L, int x);

/* restituisce la posizione di un elemento in un array
  Int_Array;  limita la ricerca alle prime L   posizioni;
  se l'elemento è presente restituisce la posizione,  altrimenti
restituisce -1 */

main ()
{
   int  L, elemento, pos;
   /* richiesta delle dimensioni reali, L  di  Int_Array */

   do {
       printf("\n Numero elementi: ");
       scanf("%d", &L);
       }
  while ((L >= MAXN) || (L < 0));

  CaricaArray(L);

printf("\n elemento da cercare: ");
scanf("%d", &elemento);
pos = RicercaLineare (L, elemento);
```

```
if (pos >= 0)
 printf(“L'elemento cercato si
trova in posizione”, “%d”, pos)
 else
   printf (“L'elemento cercato non è
           presente nell'array”)
}

void CaricaArray(int Nelem);
{ int i;
for (i=0; i <= Nelem; i++) {
printf(“Inserisci elemento %d:”, i);
scanf(“%d”, &Int_Array[i]);}...
};


int RicercaLineare(int Nelem, int x)
{int k;
k =0;
while ((k <= Nelem) && (Int_Array[k] !=
x)) ++k;
if (k <= Nelem) return k
    else  return -1;
}
```

Complessità funzione Ricerca lineare:

caso peggiore= 2+ 2MAXN, asintotica= MAXN

Primo Livello di Raffinamento

/* programma PRODOTTODIMATRICI */;

* acquisisce 2 matrici in input, restituisce la matrice prodotto (se e' correttamente calcolabile) in output *\

```
#define l  10;
#define c      10;
typedef        enum {false,true} boolean;
typedef        float Matrix [l] [c];
 Matrix    M1;                    /* prima matrice */
 Matrix    M2;                    /* seconda matrice */
 Matrix    PM;                    /* matrice prodotto */

int        d1, d2, d3, d4;
void inputmat (int *lin, int  *col, Matrix );
/* legge dimensione reale della matrice (che possono essere minori
di l e c) e memorizza gli elementi nella matrice di tipo Matrix */

void matmat (int lin1, int col1, int lin2,
                    int col2, Matrix, Matrix, Matrix
      PM,  boolean *done);
/* calcola il prodotto di 2 matrici  e lo memorizza nella matrice PM */
void printmat (int l, int c, Matrix)
/* stampa una matrice: di cui e' possibile fornire numero righe (l),
numero colonne (c) e indirizzo della matrice */
```

```
main ()
{
 boolean      done;      /* variabile logica  */

printf (“ input 2 matrici  \n ”);
                        /* input della matrice M1 */
inputmat (&d1, &d2, M1);
                         (* input della matrice M2*)
inputmat (&d3, &d4, M2);

/* calcolo della matrice PM  prodotto;
se M1 e M2 non sono moltiplicabili (col1 != l2 )
                        done= false    */
matmat ( d1, d2, d3, d4, M1, M2, PM, &done);
 if (done)
        printmat (d1, d4, PM);
}
```

```
void matmat (int l1, int c1, int l2, int c2,
                Matrix A,  Matrix B,Matrix P, boolean *dn)
/* A[l1] [c1]   B[l2] [c2]  P[l1] [c2]    */
{ int    i, j, k;
  if (c1 == l2)                                  1 +
     {
      *dn = true;                                 1 +
      for (i = 0 ; i <= l1-1; i++)
           {                                     l1 + l1*
          for (j = 0; j <= c2-1; j++)        (c2 + c2 *
            {
              P[i] [j] = 0;
              for (k = 0; k <= c1-1; k++)
                    {
                    P[i] [j] = P[i] [j] +
                         A[i] [k] * B[k] [j];
                 }                        (1 + 2*c1)
               }
          }
     }                    2+l1+2*l1*c2 + 2*l1*c1*c2
  else
     {*dn = false;
     printf (" \n"); printf (" Prodotto imposs")
     }
}
```

```
void inputmat (int *l, int *c, Matrix M)
{
printf (“ Inserisci una matrice: \n”);
printf (“ Numero di righe? “);
scanf (“%d”, l);
printf (“ Numero di colonne?”);
scanf (“%d”, c);
printf  (“ Inserisci riga per riga elementi: \n”);
                                        6   +
  for (i = 0;  i <= *l -1; i++)             l     + l *
   {
     printf (“ Nuova riga? “);
      for (j = 0 ;  j <= *c-1; j++)
        scanf (“ %f”, &M[i] [j]);                    (1+ 2*c)
   }
}        /*  inputmat */
```

```
void printmat (int l, int c, Matrix M)
{
 int i, j;
printf (“\n”); printf(“\n”);
printf (“La matrice prodotto è: \n“);  3+
     for (i = 0;  i <= l -1; i++)            l +  l*
      {
        for (j =0; i <= c-1; j++)
          printf (“%f%”, M [i] [ j]);
        printf(“/n”)
         }                                          (2*c)
}
```

*3+3l+2*l*c*

complessità della procedura inputmat:

6 + 2*l + 2*l*c

complessita' asintotica:

l*c **(quadratica)**

complessità della procedura matmat:

2 + l1 + 2*l1*c2 + 2*l1*c1*c2

complessita' asintotica:

l1*c1*c2 **(cubica)**

complessità della procedura printmat:

3 + 3*l + 2*l*c

complessita' asintotica:

l*c **(quadratica)**

complessita' asintotica del programma principale:

lin1*col2*col1

Nella complessità asintotica prevale il termine "cubico", dunque questo algoritmo ha complessità cubica.

Nel calcolo si ipotizza che le dimensioni delle matrici "differiscano di poco". Tale ipotesi è plausibile per matrici molto grandi.

COMPLESSITÀ DI ALGORITMI RICORSIVI

- Sia *S(n)* il costo incognito dell'algoritmo per input *n.*
- Dal codice dell'algoritmo, ricavare l'equazione ricorsiva di costo, espanderla e *cercare di riconoscerla* (ad esempio può risultare la somma parziale di una serie aritmetica o geometrica, ecc.)

```
1) void scrivi (int n);
    main ()

   {
      int     n;
      scanf("%d", &n);  scrivi(n)
   }

   void scrivi (int n)

   { if (n > 1) scrivi(n - 1);
     printf("%d", n);
   }  \* Esempio didattico e non efficiente! *\
```

Equazione Ricorsiva:

```
Caso Base:S(1) = 1+1= 2
Caso n:   S(n) = 1+S(n-1)+1
               = 2+S(n-1)
```

Espansione:

```
S(1) = 2
S(2) = 2 + S(1) =
S(3) = 2 + S(2) =
...
S(n) = 2n
```

Complessità asintotica

2) Numeri di Fibonacci

Definizione della successione di Fibonacci:

Fib(0)=1 *Fib(1)=1* *Fib(n)=Fib(n-1) + Fib(n-2)*

1 1 2 3 5 8 13 21 24 ...

```
int fibonacci (int m);
main ()
{
int n, i;
function fibonacci (m: integer):integer;
{
 write('Scrivi un numero:'); readln(n);
 for i := 0 to n do  writeln(fibonacci(i))
end.
 begin
   if (m = 0) or (m = 1)
   then fibonacci := 1
   else fibonacci := fibonacci(m-1) +
fibonacci(m-2)
 end;   (* Fibonacci *)
```

Equazione ricorsiva della funzione "fibonacci":

$S(n) = 1 + S(n-1) + S(n-2)$

stima di $S(n)$: $2^n - 1 <= S(n) <= 2^{n+1} - 1$

Questa stima si può dedurre dall'albero delle chiamate ricorsive di "fibonacci", ricordando che un albero binario di altezza h *completamente pieno* ha 2^{h+1}-1 nodi.

☞ scrivere un algoritmo iterativo per "fibonacci" e valutarne la complessità

3) Navigazione su strutture ad albero

```
type
   p_nodo = ^nodo;
   nodo =  record
            valore: integer;
            sinistro,destro: p_nodo
           end;
procedure CreaFoglia (var p: p_nodo; val:
integer);
(* Creazione della radice di un albero,
puntata da p *)
 var q:p_nodo;
 begin (* CreaFoglia *)
 new(p);
 p^.valore := val;
 p^.sinistro := nil;
 p^.destro := nil
 end;  (* CreaFoglia *)
procedure InsAlbero (var q: p_nodo; val:
integer);
(* Inserimento di un valore nell'albero
di radice q *)
 begin (* InsAlbero *)
 if q = nil
 thenCreaFoglia(x,p)
 else if val <= q^.valore
      then InsAlbero(q^.sinistro, val)
      else InsAlbero(q^.destro, val)
 end;  (* InsAlbero *)
```

Come misurare la dimensione dell'input di questo algoritmo?

La navigazione avviene da un livello al successivo

☞ altezza *n* dell'albero in cui inserire il valore *x.*

Sia h l'altezza *ancora da percorrere.*

Equazione ricorsiva:S(0) = 1 + 5 ... S(h) = 1 + 1 + S(h-1)

- Dato il tipo di algoritmo di costruzione dell'albero, la complessità deve risultare lineare, proporzionale all'altezza dell'albero

Espansione:

```
S(0) = 6 = 6 + 2*0
S(1) = 2 + S(0) = 6 + 2*1
S(2) = 2 + S(1) = 6 + 8 = 6 + 2*2
…
S(h) = 2 + S(h-1) = 6 + 2*h
…
S(n) = 2 + S(n-1) = 6 + 2*n  (lineare)
```

Ricerca binaria

Complessità $\lceil \log_2 n \rceil + 1$ (n = 2^h+1)

Ordinamento per selezione

Complessità n*(n-1)/2 in ogni caso

Ordinamento a bolle

Complessità massima *n*(n-1)/2*
esaminare caso migliore e medio

Complessità minima delle soluzioni per confronto di valori

- dati *n* elementi, esistono *n!* permutazioni → si cerca quella ordinata
- si supponga che ogni test partizioni l'insieme delle permutazioni candidate in due parti uguali

$$n! = 2^h - 1 \rightarrow h \approx \log_2 n! + 1 = O(n \log n)$$

Ordinamento per fusione

Complessità (di confronti) *O(n log n)*