UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Ingegneria
"Enzo Ferrari"

Fondamenti di Informatica II

Complessità computazionale degli Algoritmi

AGENDA

- **Valutazione della “bontà” di un algoritmo**
- **Complessità temporale**
- **Calcolo della complessità in numero di passi base**
- **Comportamento asintotico di una funzione**
- **Notazione O e algebra degli O**
- **Notazione Ω**

VALUTAZIONE DELLA "BONTÀ" DI UN ALGORITMO

Un problema può essere risolto da algoritmi diversi. È quindi necessario trovare un metodo per confrontare la "bontà" degli algoritmi.

Complessità computazionale

Costo di un algoritmo in termini di quantità di risorsa richiesta per il calcolo

Complessità temporale (risorsa = tempo)

Complessità spaziale (risorsa = spazio)

Complessità di Input/Output (risorsa = tempo per l'accesso alle periferiche)

Tempo Accesso RAM (core i9): 2.4 GHz > 4,2 10^{-10} sec

Tempo Accesso SSD: 1,0 10^{-5} sec (fino a 540 MB/s in lettura)

LA COMPLESSITÀ TEMPORALE

Complessità temporale

Costo di un algoritmo in termini di quantità di tempo richiesto per il calcolo

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

- Il primo algoritmo ordina gli n numeri con n^2 istruzioni
- il secondo ordina gli n numeri con $n*\log_2 n$ istruzioni.

Supponiamo di avere un processore i9 a 2.9 GHz > $3.4\ 10^{-10}$ sec.

	$n= 10^4$	$n= 10^6$
n^2	34 msec	340 sec
$n*\log_2 n$	45 µsec	6.7 msec

FATTORI PER LA VALUTAZIONE DEL TEMPO

Scelto un algoritmo per risolvere il problema, i fattori che influenzano il tempo richiesto per il calcolo sono:

- la dimensione dell'input
- la configurazione dell'input
- la velocità della macchina usata
- il linguaggio di programmazione

Vogliamo un modello di calcolo per il calcolo della complessità temporale che consenta di confrontare algoritmi a prescindere

- dal linguaggio con cui sono implementati
- dalla potenza della macchina su cui vengono eseguiti

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

Indichiamo con "n" la dimensione dell'input.

MODELLO DI COSTO

Sono operazioni di **costo unitario** le istruzioni semplici:

- Lettura da file. Es: scanf();
- Scrittura su file. Es: printf();
- Assegnamento. Es. x=x+y;

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

- Un blocco ha un costo pari alla somma dei costi delle istruzioni che contiene.
- Le espressioni condizionali hanno un costo pari alla somma del costo di valutazione della condizione più il costo del ramo selezionato
- Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni
- Le chiamate di funzioni hanno un costo pari al costo del blocco. Il passaggio di parametri ha un costo nullo.

CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE

ESEMPIO 1 (somma fino a n)

```
int i=1, s=0;
while (i <= n) {
    s=s+i;
    i=i+1;
}
```

Assegnamento esterno (i=1, s=0)	2
Numero di test (i <= n)	n+1
Assegnamento interno (s=s+i, i=i+1)	n*2
Numero totale di passi base:	3+3*n

(n è il limite della somma)

CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE

ESEMPIO 2 (potenza m^{2*n})

```
int i=1, p=1;
while (i <= 2*n) {
    p=p*m;
    i=i+1;
}
```

Assegnamento esterno (i=1, p=1)	2
Numero di test (i <= 2*n)	2*n+1
Assegnamento interno (p=p*m, i=i+1)	2*(2*n)
Numero totale di passi base:	3+6*n

(2*n è l'esponente)

CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE

ESEMPIO 3 (ciclo condizionale)

```
int i=0;
while(i < n){
    for (int j=0;j<n;j++)
        if(i<j) printf("Ciao");
        else printf("a tutti");
    i=i+1;}
```

Assegnamento esterno (i=0)	1 +
Test while (i < n)	n+1 +
Numero blocchi while (	n*(
Test nel for (j<n)	n + 1 +
Corpo for ((i<j), printf()e j++):	3*n +
Assegnamenti (j=0; e i=i+1;):	2)
Numero totale di passi base:	$2+4*n+4*n^2$

CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE

ESEMPIO 4 (chiamata procedura)

```
void Stampastelle(int ns){
   for(int i=0;i<ns;i++)
     printf("*");
}

void main(){
int n,m;
   printf("Quante stelle per riga?");
   scanf("%d",&n);
   printf("Quante righe di stelle?");
   scanf("%d",&m);
   for(j=0;j<m;j++) Stampastelle(n);}
```

Complessità della procedura:

$2 + 3*ns$

Complessità del main:

$6+m+m*((2+3*n)+1) =$

$6+4*m+3*m*n$

Nel calcolo della complessità di un algoritmo bisogna tener conto di tutti i parametri che definiscono la dimensione dell'input. Il costo di esecuzione di una procedura o funzione può dipendere dai parametri che le vengono dati in ingresso.

CALCOLO DELLA COMPLESSITÀ IN NUMERO DI PASSI BASE

ESEMPIO 5 (variante chiamata procedura)

```
void Stampastelle(int ns){
   for(int i=0;i<ns;i++)
     printf("*");
}

void main(){
int m;
    printf("Quante righe di
                stelle?");
   scanf("%d",&m);
   for(j=0;j<m;j++)
Stampastelle(j);}
```

Complessità della procedura:

$2 + 3*ns$

Complessità del main:

$4+m+\sum_{j=0}^{j=m-1}((2+3*j)+1) =$

$4+m+3*m+3*m*(m-1)/2 =$

$4+5/2*m+3/2*m^2$

Una stessa procedura/funzione può essere chiamata con dati diversi (input di dimensione diversa).

COSTO IN FUNZIONE DELLA CONFIGURAZIONE DELL'INPUT

```
int ricercaInVettore(int V[], int e){
    for(int i=0;i<n;i++)
        if(V[i]==e) return i;
    return -1;}
```

ESEMPIO 6

Supponiamo che V abbia la seguente configurazione:

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

Cerchiamo l'elemento 8 (e=8):

1 ass + 8 test + 8 test if + 7 incr. + 1 return = 2+2*8+(8-1) = 25

Cerchiamo l'elemento 6 (e=6):

Poiché tale elemento non è presente, dobbiamo scorrere l'array per intero, con complessità:

1 ass. + (n+1) test + n test if + n incr. +1 return =

3+3*n passi base (3+3*14 passi nell'esempio)

CASO MIGLIORE, MEDIO, PEGGIORE

Questo è un classico esempio in cui il costo del programma dipende non solo dalla dimensione dell'input ma anche dai **particolari valori dei dati stessi**.

Nel caso in cui il costo di esecuzione del programma dipende dalla configurazione dell'input, è possibile distinguere diversi casi:

- **Caso migliore**: si fa riferimento alla configurazione che comporta il costo minore
- **Caso peggiore**: si fa riferimento alla configurazione che comporta il costo maggiore
- **Caso medio**: si calcola la somma dei costi delle esecuzioni rispetto a tutti i possibili dati di ingresso

CASO MIGLIORE, MEDIO, PEGGIORE (2)

Caso migliore: l'elemento cercato è il primo. Il costo in numero di passi base è 4.

Caso peggiore: l'elemento cercato non appartiene all'array.

Il costo in numero di passi base è 3+3*n.

Caso medio:

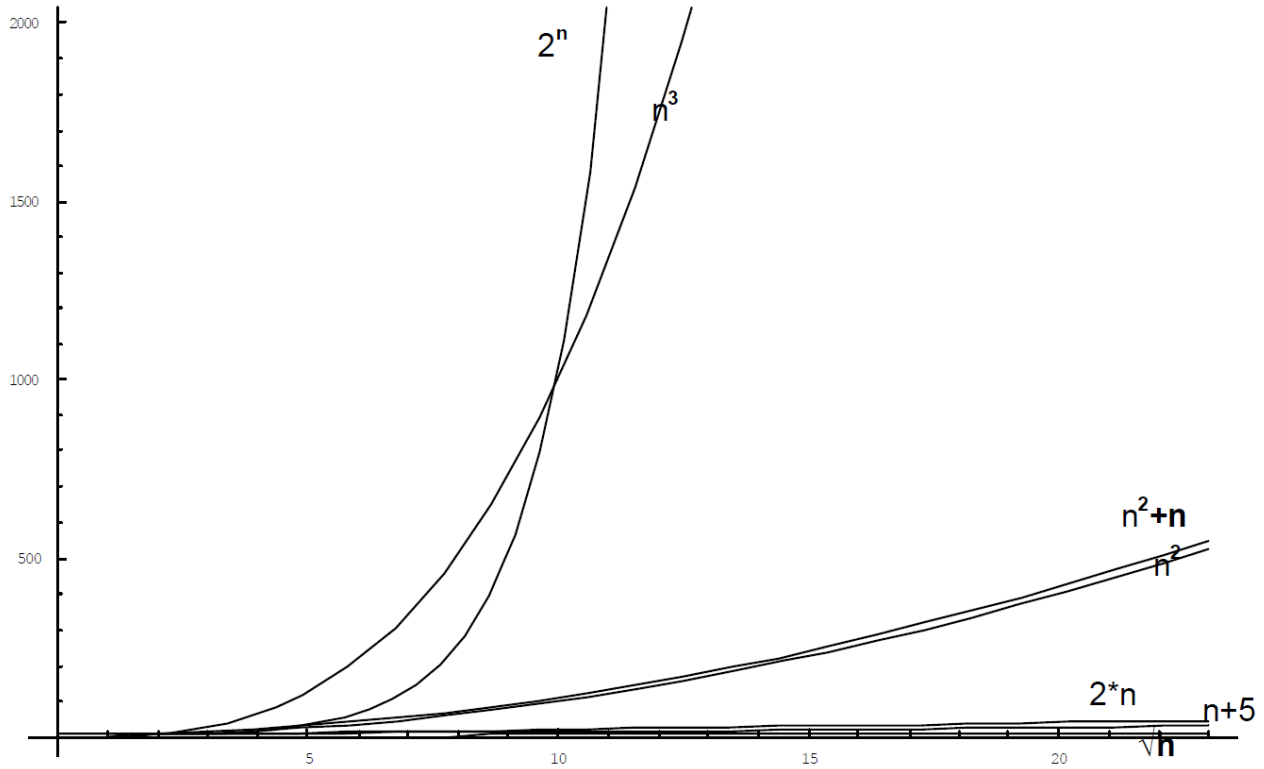
Ipotesi di distribuzione uniforme dei valori nell'array. La probabilità di trovare l'elemento e cercato in posizione i-esima è: **P(V[i]=e) = 1/n**

Se l'elemento e è nella posizione i-esima, Il costo in numero di passi base è: C(V[i]=e)=2+2*i+(i-1)=1+3*i

Il numero medio di confronti da effettuare è:

$$\sum_{i=1}^{j=n} \frac{1}{n} * (1+3*i) = 1 + \frac{3}{n} * \frac{n*(n+1)}{2} = \frac{5+3*n}{2}$$

COMPORTAMENTO ASINTOTICO DI UNA FUNZIONE



Per n che tende all'infinito le funzioni 2*n e n+5 hanno un comportamento simile. Lo stesso vale per n^2 e n^2+n.

La funzione 2^n diverge molto rapidamente

COMPORTAMENTO ASINTOTICO DI UNA FUNZIONE (2)

Non è sempre facile quantificare con esattezza la complessità di un algoritmo in numero di passi base.

Per l'analisi della complessità

dei problemi e degli algoritmi che li risolvono

si fa spesso riferimento al comportamento asintotico.

Comportamento asintotico di una funzione

Comportamento al crescere della dimensione n all'infinito trascurando le costanti moltiplicative ed additive e tutti i termini di ordine inferiore

La notazione asintotica si basa su 2 notazioni principali:

- **Notazione O e algebra degli O**
- **Notazione Ω**

NOTAZIONE O

Algoritmi diversi per la soluzione dello stesso problema possono essere confrontati valutando il numero di operazioni in ordine di grandezza.

Notazione O

f(n) è di ordine g(n), O(g(n)), se esistono due costanti positive C e n_0 tali che:

f(n) <= c * g(n) per ogni n >= n_0

Applicata alla funzione di complessità f(n), la notazione O ne limita superiormente il comportamento asintotico e fornisce quindi una indicazione sulla **bontà dell'algoritmo**.

2n+5 è O(n) poiché 2n+5<=7n per ogni n

2n+5 è O(n^2) poiché 2n+5<=n^2 per n>= 4

2n+5 è O(2^n) poiché 2n+5<=2^n per ogni n

COMPLESSITA' ASINTOTICA

La notazione O(f(n)) consente di **dividere gli algoritmi in classi**, ponendo nella medesima classe tutti quelli la cui complessità è dello **stesso ordine di grandezza**.

Complessità asintotica

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)

costante	O(1)	**polinomiale**	$O(n^k)$ con k>1
sublineare	O(log n), $O(n^k)$ con k<1	**esponenziale**	$O(c^n)$ n^n
lineare	O(n)	**fattoriale**	O(n!)
pseudo-lineare	O(n*log n)		

DAL NUMERO DI PASSI BASE ALLA COMPLESSITÀ ASINTOTICA

Numero Passi Base	Complessità Asintotica
$2*n + 2$	n (lineare)
$4*n^2 + 4*n + 5$	n^2 (quadratico)
$4*2^n$	2^n (esponenziale)
$3^n + 2*n + 5^n$	5^n (esponenziale)
$3^n + 2*n + n! + 5^n$	n! (fattoriale)

Il calcolo della complessità asintotica porta ad una valutazione approssimata della complessità

ESEMPIO: Gli algoritmi A1 e A2 con complessità in numero di passi base

$f_{A1}(n)=2*n$ e $f_{A2}(n)=1000*n$

hanno entrambi complessità lineare ma tempi di esecuzione diversi

Consente di confrontare algoritmi:

Dato un problema P e due algoritmi A1 e A2, diciamo che A1 è migliore di A2 nel risolvere P se**: $O(f_{A1}(n)) < O(f_{A2}(n))$**

ALGEBRA DEGLI "O" GRANDI

Tramite l'algebra degli "O" grandi è possibile calcolare la complessità asintotica di un programma strutturato.

In un programma a blocchi possono presentarsi due situazioni

Blocchi in sequenza

Primo blocco P

Secondo blocco Q

```
i=0;
while (i<n){
    stampastelle(i);
     i++;}
for(i=0;i<2*n;i++)
    printf("ciao");
```

$f_P(n)$: complessità del primo blocco

$f_Q(n)$: complessità del secondo

$$O(f_P(n)+ f_Q(n)) = O(\max\{ f_P(n), f_Q(n)\})$$

ALGEBRA DEGLI "O" GRANDI

Blocchi annidati

Primo blocco P

Secondo blocco Q

```
for(i=0;i<n;i++){
    s[i]=0;
    for(j=0;j<m;j++)
      s[i]=s[i]+j;
}
```

$f_P(n)$: complessità del primo blocco

$f_Q(n)$: complessità del secondo

$$O(f_P(n) * f_Q(n)) = O(f_P(n)) * O(f_Q(n))$$

ALGEBRA DEGLI "O" GRANDI: ESEMPIO

```
#define N 7
int ricercaInVettore(int V[], int e){
   for(int i=0;i<n;i++)
        if(V[i]==e) return i;
   return -1;}
void main(){
    int vett[N];
    int elem;
    for(int i=0;i<N;i++)
         scanf("%d",&vett[i]);
    printf("Elemento cercato");
    scanf("%d",&elem)
    if(ricercaInVettore(vett,elem)!=-1)
         printf("trovato!");
    else printf("non trovato!");
}
```

Due blocchi in sequenza

Acquisizione vettore

$f_P(n) = n$

Ricerca elemento

$f_Q(n) = 3 + 3*n$

$O(f_P(n) + f_Q(n)) =$

$O(f_Q(n)) = n$

NOTAZIONE Ω

Esiste un **limite inferiore alla complessità di un algoritmo**, che dipende solo dal problema in esame? In altri termini, esiste una soglia invalicabile alla complessità di ogni algoritmo non noto?

Notazione Ω

f(n) è omega di g(n), Ω(g(n)), se esistono due costanti positive C e n_0 tali che:

f(n) >= c * g(n) per ogni n >= n_0

Applicata alla funzione di complessità f(n), la notazione Ω ne limita inferiormente il comportamento asintotico. Se si riesce a dimostrare che

- un problema ha delimitazione inferiore Ω(f(n)) ovvero che ogni algoritmo per il problema in esame deve essere Ω(f(n))
- un algoritmo A è O(f(n)) allora **A è un algoritmo ottimo**