Complessità computazionale

Introduzione alla complessità computazionale

Un problema spesso può essere risolto utilizzando algoritmi diversi

- Come scegliere il “migliore”?

La **bontà** o **efficienza** di un algoritmo si misura in base alla quantità minima di **risorse** sufficienti per il calcolo:

- Il **tempo** richiesto per eseguire le azioni elementari
- Lo **spazio** necessario per memorizzare e manipolare i dati

La risorsa di maggior interesse è il **tempo**

Misurare il tempo di calcolo

Non è significativo misurare il tempo di calcolo di un algoritmo in base al numero di secondi richiesti per l'esecuzione su un elaboratore di un programma che lo descriva

Infatti, tale tempo dipende da:

- i dati di ingresso
- il linguaggio in cui è scritto il programma
- la qualità della traduzione in sequenze di bit
- la velocità dell'elaboratore

Una buona misura dell'efficienza di un algoritmo deve prescindere dal calcolatore utilizzato

Occorre una misura "astratta" che tenga conto del "metodo di risoluzione" con cui l'algoritmo effettua la computazione

Misurare il tempo di calcolo

La complessità dei problemi da risolvere dipende dalla dimensione dei dati in ingresso

Il tempo di calcolo si può quindi esprimere come:

il numero complessivo di operazioni elementari in funzione della dimensione n dei dati in ingresso

Le operazioni elementari sono quelle:

- aritmetiche
- logiche
- di confronto
- di assegnamento

Solitamente, il numero di operazioni considerate è valutato nel **caso peggiore**, ossia nel caso di dati in ingresso più sfavorevoli tra tutti quelli di dimensione n

Calcolo dell'efficienza: un esempio

Trovare l'elemento minimo in un insieme di **n** numeri $\{x_1, x_2, \ldots, x_n\}$

Soluzione:

1. considero x_1 come candidato ad essere il minimo
2. confronto il candidato con tutti gli altri elementi e se trovo un elemento più piccolo lo faccio diventare il nuovo candidato
3. al termine dei confronti, il candidato corrente è sicuramente il minimo

Complessivamente sono stati eseguiti **n** confronti

L'efficienza dell'algoritmo è quindi **direttamente proporzionale** alla dimensione dell'input

Calcolo dell'efficienza: un altro esempio

Ordinare in ordine crescente un insieme di **n** numeri $\{x_1, x_2, \ldots, x_n\}$

Soluzione:

1. per i = 1,2,…,n ripeti le seguenti operazioni:
2. trova l'elemento minimo nel sottoinsieme $\{x_i, x_{i+1}, \ldots, x_n\}$
3. scambialo con x_i

Per **n** volte bisogna trovare il minimo su insiemi sempre più piccoli

Si eseguono $n + (n-1) + (n-2) + \ldots + 2 + 1 \approx$ **n^2 operazioni**

Efficienza degli algoritmi

L'efficienza di un algoritmo si può esprimere mediante una funzione f(n) della variabile n:

- esprime il numero di operazioni compiute per un problema di dimensione n
- rappresenta la **complessità computazionale** dell'algoritmo

Se A e B sono due algoritmi che risolvono lo stesso problema e se $f_A(n)$ e $f_B(n)$ esprimono la complessità dei due algoritmi allora:

A è migliore di B se, al crescere di n, risulta
$f_A(n) \leq f_B(n)$

Efficienza degli algoritmi: esempio

Supponiamo di avere 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)

Tempo di esecuzione:

	n=10	n=10000	$n=10^6$
n^2 op.	0,1 msec	100 sec (1,5 min)	10^6 sec (~12 gg.)
n*log n op.	23 μsec	92 msec	13,8 sec

Quali fattori influenzano il tempo di 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

Dipendenza dalla tecnologia

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

- Algoritmi di Ricerca (*n* operazioni)
- Algoritmi di Ordinamento (n^2 operazioni)
- Algoritmi Decisionali (2^n operazioni)

- Esempio:

Complessità	Tecnologia attuale	100 volte più veloce	1000 volte più veloce
n	N_1	$N_1 * 100$	$N_1 * 1000$
n^2	N_2	$N_2 * 10$	$N_2 * 31,6$
2^n	N_3	$N_3 + 6,64$	$N_3 + 9,97$

N = numero dati gestiti in un lasso di tempo t (dipendente dalla complessità)

Ad es.: 2^N operazioni (op.) nel tempo t. Con tecnologia 100 volte più veloce 2^N op. in t/100, quindi $2^N * 100$ op. nel tempo t, $2^N * 2^{6,64} = 2^{N+6,64}$ op. nel tempo t.

Il modello di costo

Per giungere a un modello di costo è necessario definire:

- Dimensione dell'input
- Istruzione di costo unitario (passo base)
- Calcolo della complessità in numero di passi base
- Complessità
  - ✓ Nel caso migliore
  - ✓ Nel caso medio
  - ✓ Nel caso peggiore
- Complessità di programmi strutturati
- Complessità asintotica

Dimensione dell'input

- 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

Operazione di costo unitario

- È un'operazione la cui esecuzione non dipende dai valori e dai tipi delle variabili
  - Assegnamento e operazioni aritmetiche di base
  - Accesso ad un elemento qualsiasi di un vettore residente in memoria centrale
  - Valutazione di un'espressione booleana qualunque
  - Istruzioni di input/output

- In seguito, si indicherà un'operazione di costo unitario con il termine passo base

Modello semplificato

- Il costo del test di una condizione booleana composta di più condizioni booleane semplici è sempre minore o uguale a *K* volte il costo del test di una condizione semplice, dove *K* è una opportuna costante numerica, quindi, semplificando, consideriamo comunque un costo unitario
- Lo stesso vale per le *operazioni aritmetiche composte*
- Non si considerano, in questa sede, operazioni di accesso ai file
- Un modello più sofisticato dovrebbe inoltre:
  - differenziare tra aritmetica intera e reale (floating point)
  - tenere conto di come il compilatore traduce le istruzioni di alto livello (compilatori ottimizzanti)
  - considerare l'architettura di una specifica macchina

Calcolo della complessità in numero di passi base

```
i = 0;
while(i < n)
          i = i+1;
```

- Assegnamento esterno al ciclo: i=0;
- Ciclo while: si compone di un test (i<n) e di un corpo costituito da un'operazione di assegnamento (i=i+1)
- Quindi:
  - Assegnamento esterno: 1
  - Numero di test: n+1
  - Assegnamenti interni: 1*n

  - Numero totale di passi base: 2*n+2

Calcolo della complessità in numero di passi base (2)

```
i = 0;
while(i < n){
          i = i+1;
          j = j*3 + 42;
}
```

- Il corpo del ciclo si compone di 2 passi base:
  - Assegnamento esterno: 1
  - Numero di test: n+1
  - Assegnamenti interni: 2*n

  - Numero totale di passi base: 3*n+2

Calcolo della complessità in numero di passi base (3)

```
i = 0;
while(i < 2*n){
          i = i+1;
          j = j*3 + 4367;
}
```

- Il test viene eseguito 2*n+1 volte e il corpo è costituito da due passi base:
  - Assegnamento esterno: 1
  - Numero di test: 2*n+1
  - Assegnamenti interni: 2*(2*n)

  - Numero totale di passi base: 6*n+2

Cicli annidati

```
i=0;
while(i<n){
   for(j=0; j<n; j++){
       . . .

System.out.println("Ciao!");
       . . .
   }
   i=i+1;
}
```

- n+1 test
- un ciclo for per ogni while
- una scrittura per ogni for
- un assegnamento per ogni while

- Assegnamento esterno 1+
- Test while: n+1+
- Cicli while: n*(
- Assegnamento iniziale for: 1+
- Controlli ciclo for: n+1+
- Incrementi ciclo for: n+
- Scritture: n+
- Assegnamento: 1)

- Totale passi base $2+4*n+3*n^2$

Il costo può dipendere dal *valore* dei dati in ingresso!

- Attenzione: esprimere la complessità in funzione dei dati in ingresso!

```
i = 0;
while(i*i < n)
          i = i+1;
```

  - Quanti test vengono eseguiti?
  - Quante volte viene eseguito il ciclo?

Il costo può dipendere dal *valore* dei dati in ingresso!

- Supponiamo n=9:
  - i=0
  - i^2=0 <9 ? sì i=1
  - i^2=1 <9 ? sì i=2
  - i^2=4 <9 ? sì i=3
  - i^2=9 <9 ? no fine ciclo

```
i = 0;
while(i*i < n)
        i = i+1;
```

  - Il ciclo viene eseguito 3 = √9 volte, eseguendo 4 = √9+1 test
  - La complessità in passi base di questo blocco è dunque:

$$1 + (\sqrt{n}+1) + \sqrt{n} = 2 + 2^* \sqrt{n}$$

(con arrotondamento all'intero superiore)

Dipendenza del costo dal valore dei dati in ingresso

- Due tipici esempi sono dati da:
  1. if(condizione){ corpo istruzioni }
  2. if(condizione) { corpo1 } else { corpo2 }

- Nel caso 1) il costo è dato dal costo di un test più:
  - Il costo di esecuzione del corpo istruzioni se la condizione è vera
  - Nessun costo aggiuntivo se la condizione è falsa
- Nel caso 2) il costo è dato dal costo di un test più:
  - Il costo di esecuzione del corpo1 se la condizione è vera
  - Il costo di esecuzione del corpo2 se la condizione è falsa

Dipendenza del costo dal valore dei dati in ingresso

```
for(i=0; i<n; i++){
   if(condizione){ /* if annidato */
     <corpo1>
   }
   else{
     <corpo2>
   }
}
```

- 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 else> può variare a seconda del valore assunto dalla condizione

Complessità nei casi migliore, peggiore e medio

- Supponiamo di avere il seguente vettore A contenente 14 elementi:

```
A = {3, -2, 0, 7, 5, 4, 0, 6, -3, -1, 9, 12, 20, 5}
```

- La dimensione dell'input è la lunghezza dell'array (n=14 nell'esempio)
- Cerchiamo l'elemento "6" con un algoritmo di ricerca lineare:

```
i = 0;
while(i<14 && A[i] != 6)
        i = i+1;
```

- Il cui costo corrisponde a:
  - 1 assegnamento + 8 test + 7 cicli = 16 passi base

Complessità nei casi migliore, peggiore e medio

```
A = {3, -2, 0, 7, 5, 4, 0, 6, -3, -1, 9, 12, 20, 5}
```

- Se avessimo cercato l'elemento "13"?
- Poiché tale elemento non è presente nel vettore, avremmo dovuto scorrere interamente il vettore con una complessità:

$1 + (n+1) + n = 2 + 2*n$ passi base
(30 passi nell'esempio)

Ricerca di un elemento in un vettore

- Per il problema della ricerca in un vettore *non ordinato,* bisogna distinguere 2 casi:
  - Si sa che l'elemento cercato è presente nel vettore
  - Non si sa se l'elemento cercato sia presente nel vettore

Prima situazione (elemento presente)

- Si possono distinguere 3 casi:
  - Caso migliore: l'elemento cercato è in prima posizione. Il costo effettivo dell'algoritmo è 2
  - Caso peggiore: l'elemento cercato è in ultima posizione. Il costo effettivo è 2*n
  - Caso medio: consideriamo un'ipotesi di distribuzione uniforme, ovvero che i valori siano disposti casualmente nel vettore
    - Poiché i valori sono distribuiti uniformemente, la probabilità di trovare l'elemento cercato in posizione i nel vettore (di dimensione n) è:

$$Pr(i) = 1/n$$

Prima situazione (elemento presente)

- Il numero di operazioni di test da eseguire per trovare l'elemento cercato nella posizione *i* è:

$$C(i) = i$$

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

$$C_{avg} = \sum_{i=1}^{n} C(i) \cdot \Pr(i) = \sum_{i=1}^{n} \frac{i}{n} = \frac{n+1}{2}$$

Seconda situazione (non si sa se elemento sia presente)

- In questa situazione non si può assegnare un valore di probabilità come nel caso precedente
- Quindi, in questa situazione:
  - Caso migliore: è la presenza dell'elemento nel vettore
  - Caso peggiore: è l'assenza. Implica il costo di scansione dell'intero vettore
  - 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

Complessità nei casi migliore, peggiore e medio

- 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, salvo diversa indicazione, verrà considerata la complessità nel caso peggiore

Programmi strutturati

- Nel calcolo della complessità di un programma strutturato si deve:
  - Calcolare la complessità di ogni procedura/funzione/metodo
  - Per ogni chiamata a procedura/funzione/metodo, aggiungere la complessità della/o stessa/o al costo globale del programma

Complessità di programmi strutturati

```
public static void main(String[] args) {
	Scanner scanner=new Scanner(System.in);
	System.out.println("Quante stelle per riga? ");
	int n=scanner.nextInt();
	System.out.println("Quante righe di stelle? ");
	int m=scanner.nextInt();
	for(int j=0; j<m; j++) stars(n);
}

public static void stars(int ns) {
	System.out.println();
	for(int i=0; i<ns; i++)
		System.out.print('*');
}
```

Complessità dell'esempio (1)

```
public static void stars(int ns) {
        System.out.println();
        for(int i=0; i<ns; i++)
                System.out.print('*');
}
```

- Complessità di stars:
  - 1 istruzione di output
  - 1 assegnamento
  - ns+1 confronti e ns incrementi
  - ns cicli
  - 1 istruzione di output per ogni iterazione
- Complessità totale: 3 + 3*ns

Complessità dell'esempio (2)

```
public static void main(String[] args) {
        Scanner scanner=new Scanner(System.in);
        System.out.println("Quante stelle per riga? ");
        int n=scanner.nextInt();
        System.out.println("Quante righe di stelle? ");
        int m=scanner.nextInt();
        for(int j=0; j<m; j++) stars(n);
}
```

- Complessità del main:
  - 4 istruzione di input/output
  - 1 assegnamento
  - m+1 confronti e m incrementi
  - m cicli
  - 1 chiamata di metodo per ogni iterazione

- Complessità totale: $4 + 1 + m + 1 + m + m*(3 + 3*n)$

$$= 6 + 5*m + 3*m*n$$

Complessità dell'esempio (3)

- Note:
  - Se in un algoritmo la dimensione dell'input è definita da più parametri, come nell'esempio precedente, bisogna tenere conto di tutti
  - Il costo di esecuzione di una procedura o funzione (metodo) può dipendere dai parametri che vengono forniti in ingresso
  - Una stessa procedura/funzione/metodo può essere chiamata, all'interno del programma principale, con dati diversi (input di dimensione diversa)

Altro esempio

```
public static void main(String[] args) {
        Scanner scanner=new Scanner(System.in);
        System.out.println("Quante righe di stelle? ");
        int m=scanner.nextInt();
        for(int j=1; j<=m; j++) stars(j);
}
```

- In questo caso, la lunghezza della fila di asterischi si incrementa ad ogni iterazione
- Calcolo della complessità:
  - Alla j-esima iterazione, il costo della funzione è 3+3*j, quindi:

$$3 + (m + 1) + m + \sum_{j=1}^{m} (3 + 3 * j) = 4 + \frac{13 * m}{2} + \frac{3 * m^2}{2}$$

Altro esempio

```
public static void main(String[] args) {
	Scanner scanner=new Scanner(System.in);
	System.out.println("Quante potenze vuoi stampare? ");
	int n=scanner.nextInt();
	System.out.println("Base delle potenze? ");
	int b=scanner.nextInt();
	for(int exp=0; exp<n; exp++)
		potenza(b, exp);
}

public static int potenza(int base, int exp) {
	int ris=1;
	for(int i=0; i<exp; i++)
		ris*=base;
	return ris;
}
```

Complessità dell'esempio (1)

```
public static int potenza(int base, int exp) {
        int ris=1;
        for(int i=0; i<exp; i++)
                ris*=base;
        return ris;
}
```

- Complessità della funzione potenza: 3 + 3*exp

Complessità dell'esempio (2)

```
public static void main(String[] args) {
        Scanner scanner=new Scanner(System.in);
        System.out.println("Quante potenze vuoi stampare? ");
        int n=scanner.nextInt();
        System.out.println("Base delle potenze? ");
        int b=scanner.nextInt();
        for(int exp=0; exp<n; exp++)
                potenza(b, exp);
}
```

- Complessità del main:
  - main: 6 + 2*n +
  - 1ª chiamata: 3 + 3*0 +
  - 2ª chiamata: 3 + 3*1 +
  - …
  - jª chiamata: 3 + 3*(j-1) +
  - …
  - nª chiamata: 3 + 3*(n-1)
- Totale: 6 + 2*n + $\sum_{j=0}^{n-1}(3 + 3 * j)$ = 6 + 7*n/2 + 3*n^2/2

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*

Complessità	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 (~12 gg.)
n^2+n	10^{-4}	10^{-2}	1 sec	10^6 (~12 gg.)
n^3	10^{-3}	1 sec	10^5 (~ 1 g)	10^{12} (~300 secoli)
2^n	10^{-3}	~ $4*10^{14}$ secoli	~ $3*10^{287}$ secoli	~ $3*10^{301016}$ 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.)
- Per grandi dimensioni dell'input (n=10^6), i sette algoritmi di 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 — miliardi di secoli…

Gli «O grandi»

- È 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:

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

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

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

- Se *g(n)* non è identicamente nulla (fatto assolutamente certo nell'esecuzione di un algoritmo), la condizione precedente può essere espressa come:

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

ovvero la funzione f(n)/g(n) è limitata

Gli «O grandi»

- Esempi
  - $2*n+5 = O(n)$ poiché $2*n+5 \leq 7*n$ per ogni n
  - $2*n+5 = O(n^2)$ poiché $2*n+5 \leq n^2$ per $n \geq 4$
  - $2*n+5 = O(2^n)$ poiché $2*n+5 \leq 2^n$ per $n \geq 4$
- Esempi inversi:
  - $n = O(2*n+5)$? sì
  - $n^2 = O(n)$? no, $n^2/n = n$, non limitata
  - $e^n = O(n)$? no, $e^n/n \geq n^2/n = n$, non limitata

Ordinamento fra gli «O grandi»

- È possibile definire un criterio di ordinamento fra gli «O grandi»
- *f(n)* è più piccola di (di ordine inferiore a) *g(n)* se valgono le due condizioni seguenti:
  - $f(n) = O(g(n))$
  - $g(n)$ non è $O(f(n))$
- Esempi:
  - $\sqrt{n}$ è di ordine inferiore a $2n$ +5 (e quindi a n)
    - $(2n+5)/\sqrt{n} = 2\sqrt{n} + 5/\sqrt{n} \leq 2\sqrt{n} + 5$ che non è limitata
  - n è di ordine inferiore a e^n
    - $n = O(e^n)$, e^n non è $O(n)$

Complessità asintotica

- Si dice che *f(n)* ha complessità asintotica *g(n)* se valgono le seguenti condizioni:
  - $f(n) = O(g(n))$
  - *g(n)* è la più piccola di tutte le funzioni che soddisfano la prima condizione
- 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)
  - $3n + 2^n + 5^n$ — complessità asintotica 5^n (esponenziale di base 5)
  - $2^n + 5n + n! + 5^n$ — complessità asintotica $n^n \sqrt{n}$ (fattoriale)

Dalla complessità in passi base alla complessità asintotica

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

- In pratica, la complessità asintotica è definita dal blocco di complessità maggiore
- Non contano le costanti, né additive, né 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» $m*n$

- Osservazioni:
  - Le ipotesi semplificative del modello di costo introdotto e 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 e 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»

- In precedenza, 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 strutturato a blocchi si possono presentare due situazioni:
  - blocchi in sequenza
  - blocchi annidati

Blocchi in sequenza

```
i=0;
while(i<n) {
    stampastelle(i);
    i=i+1;
}
for(i=0; i<2*n; i++)
    numero = scanner.nextInt();
```

- Sia $g_1(n)$ la complessità del primo blocco e $g_2(n)$ la complessità del secondo. La complessità globale è:

$$O(g_1(n)+g_2(n)) = O(\max\{g_1(n),g_2(n)\})$$

- La complessità di un blocco costituito da più blocchi in sequenza è quella del blocco di complessità maggiore

Blocchi annidati

```
for(i=0; i<n; i++)
    j = scanner.nextInt();
    System.out.print(j*j);
    do{
       n = scanner.nextInt();
       j=j+1;
    }while(j<=n);
}
```

- Sia $g_1(n)$ la complessità del blocco esterno e $g_2(n)$ la complessità del blocco interno. La complessità globale è:

$$O(g_1(n)*g_2(n)) = O(g_1(n))* O(g_2(n))$$

- La complessità di un blocco costituito da più blocchi annidati è data dal prodotto delle complessità dei blocchi componenti

Utilità dell'algebra degli «O grandi»

- Con il concetto di «O grande» e le due operazioni definite dall'algebra degli «O grandi» si può calcolare la complessità di qualsiasi algoritmo strutturato senza conoscerne preventivamente la complessità in passi base
- Esempio: ricerca lineare in un vettore

```
public boolean cercaElemento(int[] V, int n, int el){
   for(int i=0; i<n; i++)
        if(V[i]==el)
           return true;
   return false;
}
```

- Complessità di cercaElemento:
  - Caso peggiore: 2 + 3n
  - Asintotica: n

Prodotto di matrici

```
float A[]=new float[N][M], B[]=new float[M][P], C[]=new float[N][P];
for(int i=0; i<N; i++)
   for(int j=0; j<P; j++){
      C[i][j]=0;
      for(int k=0; k<M; k++)
         C[i][j] += A[i][k]*B[k][j];
   }
```

- Complessità asintotica del programma: N*M*P