Complessità computazionale

Cosa si intende per *complessità*?

È l'impiego, in termini di impegno di risorse, dell'intero sistema di calcolo necessario per la risoluzione di un particolare tipo di problema.
Può essere valutata secondo diverse dimensioni, in accordo con le diverse componenti che cooperano in un sistema di calcolo:

- complessità temporale;
- complessità spaziale;
- complessità di input/output;
- complessità di trasmissione.

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.

Tipi di complessità

Complessità Temporale

Quanto tempo richiede l'esecuzione di un algoritmo?
Ininfluenza della velocità della macchina per molte importanti classi di algoritmi.

Complessità Spaziale

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

Complessità di I/O (Input/Output)

Quanto tempo richiede l'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: nanosecondi (~ 10^{-9} sec)
- Memoria di massa: millisecondi (~ 10^{-3} sec)

Complessità di Trasmissione

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

☞Considereremo principalmente la complessità temporale.

Complessità spaziale

Indica il quantitativo di memoria principale necessario per l'esecuzione di un programma.
La domanda che ci si pone è: *quale occupazione di memoria richiede l'esecuzione di un dato programma*?
La valutazione deve tenere conto dei due termini che contribuiscono al calcolo complessivo:

- occupazione media
- occupazione massima

Occupazione massima

È il picco di occupazione istantaneo raggiunto, indipendentemente dalla durata dell'intervallo temporale per cui tale memoria è allocata.

Occupazione media

Può essere calcolata tenendo conto della stima dei valori di occupazione della memoria ottenuti nel corso dell'esecuzione del programma e facendone una media nel tempo.
Supponendo di campionare l'occupazione con periodo t ed indicando con M_i il quantitativo di memoria occupato all'istante i-esimo e con n la durata (in periodi) del programma, otteniamo:

$$Mem = \frac{1}{nt}\sum_{i=1}^{n} M_i \cdot t = \frac{1}{n}\sum_{i=1}^{n} M_i$$

$$Mem_{\max} = \max_{i=1}^{n}\{M_i\}$$

Scopo dello studio della complessità spaziale

Supponiamo di lavorare con una macchina dedicata e che esista un sistema di tariffazione per l'utilizzo delle varie risorse (tra cui la memoria) del sistema di calcolo.
L'occupazione di memoria contribuirà al conteggio dell'addebito da parte del gestore del sistema.
Il calcolo complessivo della memoria occupata ed il relativo computo del costo da addebitare dovrà essere effettuato tenendo conto di entrambi i fattori precedentemente discussi, usando opportuni pesi.
L'equazione di costo sarà pertanto data come:

$$C(spazio) = p_{\max} * Mem_{\max} + p_{\text{med}}\, Mem$$

Una buona tecnica di programmazione deve cercare di minimizzare tale costo. Per cercare di ridurre lo spazio occupato in memoria centrale occorre:

- **Strutturare opportunamente il programma**, in modo da sfruttare la località delle variabili nelle procedure.
- **Scegliere opportunamente le strutture dati** e dimensionarle sulla base delle necessità effettive, in modo da non sprecare inutilmente spazio.

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.

Esempio:

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).

	$n = 10$	$n = 10000$	$n = 10^6$
n^2 operazioni	0,1 msec	100 sec (1,5 minuti)	10^6 sec (~ 12 giorni)
$n*\log n$ operazioni	23 µsec	92 msec	13.8 sec

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

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	$N_3 + 6.64$	$N_3 + 9.97$

Il modello di costo

Per giungere ad 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 e 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 ed 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

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) e considerare l'architettura di una specifica macchina.

☞In seguito, si indicherà una operazione di costo unitario con il termine passo base.

Calcolo della complessità in numero di passi base

Esempio 1:

```
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 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

Esempio 2:

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

Il corpo del ciclo si compone di due passi base.

Assegnamento esterno:	1
Numero di test:	n+1
Assegnamenti interni:	2*n
Numero totale di passi base:	3*n+2

Esempio 3:

```
i=0;
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.

Assegnamento esterno:	1
Numero di test:	2*n+1
Assegnamenti interni:	2*(2*n)
Numero totale di passi base:	6*n+2

Esempio 4 – Cicli annidati:

```
i=0;
while (i<n) {
  for(j=0; j<n; j++) {
    ...
    printf("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)
Numero totale di passi base:	$2+4*n+3*n^2$

Esempio 5:

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

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

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

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)

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

Dipendenza del costo dal valore dei dati in ingresso

Due tipici esempi sono dati da:

```
if(condizione) { corpo istruzioni }
```

```
if(condizione) { corpo 1 }
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=0; i<n; i++) {
  if(condizione) /* if annidato */
  {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 else> può variare a seconda del valore assunto dalla condizione.

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

Supponiamo di avere il seguente vettore `A` contenente 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 < 14) && (A[i] != 8))
  i=i+1;
```

il cui costo corrisponde a:

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

Se avessimo cercato l'elemento "6"?
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 due casi:

- Si sa che l'elemento cercato è presente nel vettore.
- Non si sa se l'elemento cercato è presente nel vettore.

Prima situazione (elemento presente):

Si possono distinguere tre 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:

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

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 l'elemento è presente):

In questa situazione non si può assegnare un valore di probabilità come nel caso precedente.
Quindi, se ci si trova in questa situazione:

- Il "caso migliore" è la presenza dell'elemento nel vettore.
- 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 vettore.

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

Esempio:

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

main() {
  int n, m, j;
  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 dell'esempio

Complessità della procedura

- 1 istruzione di output
- 1 assegnamento
- ns+1 confronti ed ns incrementi
- ns cicli
- 1 istruzione di output per ogni iterazione

$$3 + 3*ns$$

Complessità del programma principale

- 4 istruzioni di input/output
- 1 assegnamento
- m+1 confronti ed m incrementi
- m cicli
- 1 chiamata di funzione per ogni iterazione

$$4 + 1 + m + 1 + m + m*(3 + 3*n) = 6 + 5*m + 3*m*n$$

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 può dipendere dai parametri che vengono forniti in ingresso.
- Una stessa procedura/funzione può essere chiamata, all'interno del programma principale, con dati diversi (input di dimensione diversa).

Altro esempio

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

main() {
 int m, j;
 printf("Quante righe di stelle?");
 scanf("%d", &m);
 for(j=1; j<=m; j++)
  Stampastelle(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:

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

Altro esempio

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

main() {
  int b, n, esp;
  printf("Quante potenze vuoi
stampare?");
  scanf("%d", &n);
  printf("Scrivi la base");
  scanf("%d", &b);
  for(esp=1; esp<=n; esp++)
    printf("%d", potenza(b, esp));
}
```

Complessità della funzione potenza:

$3 + 3*esp$

Complessità totale:

main:	$6 + 3*n +$
1ª chiamata:	$3 + 3*1 +$
2ª chiamata:	$3 + 3*2 +$
…	…
jª chiamata:	$3 + 3*j +$
…	…
nª chiamata:	$3 + 3*n$
totale:	$6 + 3*n + \sum(3 + 3*j) = 6 + 15*n/2 + 3n^2/2$

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.

	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}	~$4*10^{14}$ 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 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	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.

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 $n \geq 4$
$2*n+5 = O(2^n)$ poichè $2*n+5 \leq 2^n$ per ogni n

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".

Definizione:

$f(n)$ 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)

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	$2\,n^2 + 2n + 2$	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, 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" $m*n$

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"

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

```
i=0;
while(i<n) {
  Stampastelle(i);
  i=i+1;
}
for(i=0; i<2*n; i++)
  scanf("%d", &numero);
```

Sia $g_1(n)$ la complessità del primo blocco e $g_2(n)$ la complessità del secondo blocco. 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++) {
 scanf("%d", &j);
 printf("%d", j*j);
 do {
  scanf("%d", &numero);
  j=j+1;
 } while (j<=n);
```

Sia $g_1(n)$ la complessità del blocco esterno e $g_2(n)$ la complessità di quello 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.

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.

Esempi

Ricerca lineare di un elemento in un vettore

```
int cercaelemento(int *V, n, el) {
 int i;

 for(i=0; i<n; i++)
  if(V[i]==el) return 1;
 return 0;
}
```

Complessità della funzione `cercaelemento`:

- Caso peggiore 3+2n
- Asintotica n

Prodotto di matrici

```
float A[N][M], B[M][P], C[N][P];
int i, j, k;

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

Complessità asintotica del programma: `N*M*P`.

Complessità di Algoritmi Ricorsivi

Sia $S(n)$ il costo incognito dell'algoritmo per un input di dimensione n. Dal codice dell'algoritmo, occorre 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.)

Esempio

```
void scrivi(int n) {
  if(n>1) scrivi(n-1);
  printf("%d", n);
}

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

Equazione Ricorsiva:

- Caso base: $S(1) = 1+1 = 2$
- Caso n: $S(n) = 1+S(n-1)+1 = 2+S(n-1)$

Espansione:

- $S(1) = 2$
- $S(2) = 2 + S(1) = 4$
- $S(3) = 2 + S(2) = 6$
- ...
- $S(n) = 2n$

☞Complessità asintotica: n

Calcolo del fattoriale

```
int fattoriale(unsigned int n)
{
  if(n==0) return 1;
  else return n*fattoriale(n-1);
}

main() {
  int n;
  printf("\nIntrodurre N:\t");
  scanf("%d", &n);
  printf("\nFattoriale di %d:\t%d\n",
    n, fattoriale(n));
}
```

Equazione Ricorsiva:

- Caso base: $S(1) = 1+1= 2$
- Caso n: $S(n) = 1+S(n-1)+1 = 2+S(n-1)$

Espansione:

- $S(1) = 2$
- $S(2) = 2 + S(1) = 4$
- $S(3) = 2 + S(2) = 6$
- ...
- $S(n) = 2n$

☞Complessità asintotica: n