Complessità dei metodi e degli algoritmi

Carla Binucci e Walter Didimo

Efficienza di un metodo

Un metodo è tanto più efficiente quanto minori sono le risorse di calcolo necessarie per la sua esecuzione

Le risorse del calcolatore necessarie per l'esecuzione di un metodo sono tipicamente:

- tempo di esecuzione
- quantità di memoria
- accessi a dischi e alla rete

Complessità di un metodo

L'analisi di complessità di un metodo è la valutazione dell'efficienza del metodo

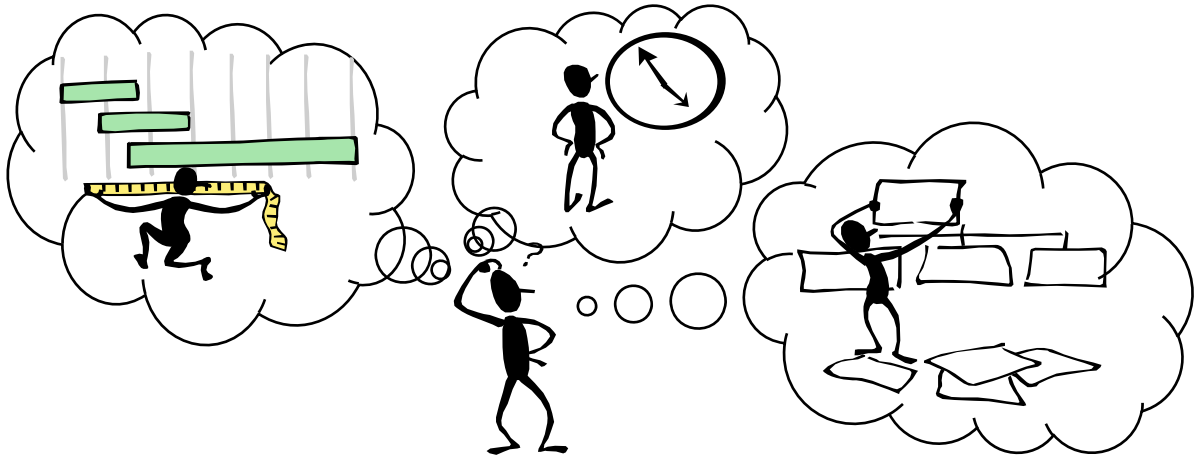
La complessità di un metodo è una quantificazione delle risorse necessarie al metodo per risolvere un determinato problema

Complessità temporale

Ci concentriamo sull'analisi di complessità temporale di un metodo

- un metodo è tanto più efficiente quanto minore è il tempo richiesto per la sua esecuzione

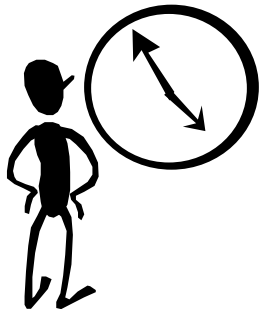
Come si misura la complessità temporale di un metodo?



Idea del cronometro

Possiamo valutare sperimentalmente la complessità temporale di un metodo usando un cronometro per misurare il tempo di esecuzione del metodo su un *insieme di dati di ingresso (input)*

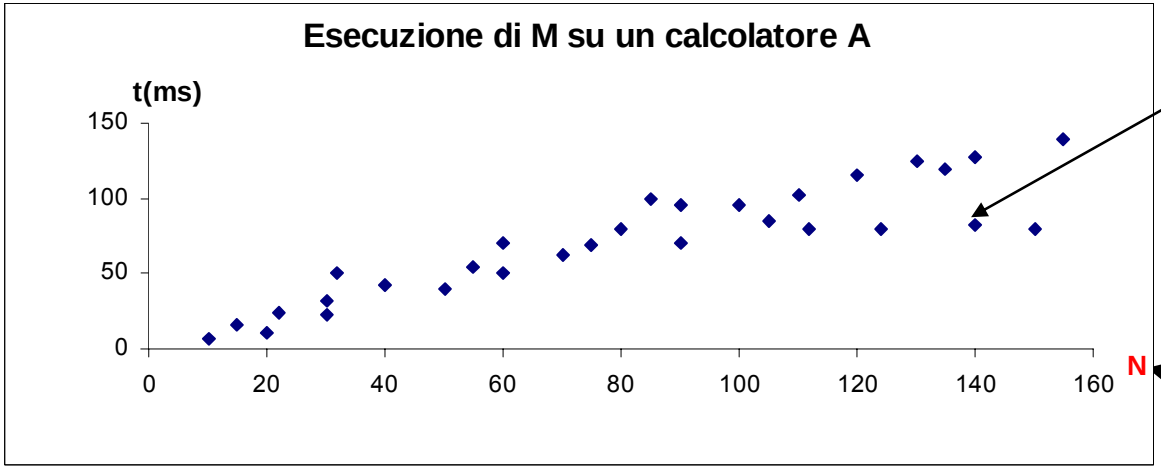
- il metodo viene eseguito più volte, su vari input, e mediante un cronometro viene misurato il tempo impiegato per ciascuna esecuzione del metodo



Misurazione mediante cronometro

Il tempo di esecuzione del metodo viene determinato in funzione della dimensione *N* dell'input

- la dimensione dell'input è una misura dello spazio di memoria necessario a memorizzare l'input



ogni punto rappresenta una esecuzione — su un diverso insieme di dati di ingresso

la dimensione dell'input

Limiti del metodo del cronometro

Il metodo del cronometro ci fornisce una valutazione poco oggettiva perché:

- dipende dalla scelta dell'input
- dipende dal calcolatore utilizzato
- la misurazione è possibile solo se il metodo è stato implementato e si ha un calcolatore a disposizione per eseguirlo

Funzione di costo

Vogliamo definire la complessità temporale di un metodo mediante una sua funzione di costo che esprima il tempo di esecuzione del metodo in modo:

- *indipendente* dal calcolatore e dai valori dell'input
- *dipendente* dalla dimensione dell'input

L'approccio utilizzato per valutare la funzione di costo si chiama analisi di complessità (non è un approccio sperimentale, ma analitico)

Dimensione dell'input

Come si misura la dimensione dell'input?

- la misurazione esatta è quella espressa dal *numero di bit* necessari per codificarlo

- tuttavia, dal punto di vista teorico, possiamo trascurare i bit e concentrarci sul numero di oggetti "elementari" (item) che formano l'input (numeri, caratteri, ecc.)

- nella maggior parte dei casi (tutti quelli che analizzeremo) questa esemplificazione non modifica la rilevanza dell'analisi dal punto di vista asintotico

Modello di costo

Dato un metodo *M*, indichiamo con $t_M(N)$ la sua funzione di costo; per esprimere $t_M(N)$ occorre definire un modello di costo:

- un modello di costo associa un costo ad ogni tipologia di istruzione
  - vengono definite delle istruzioni di costo unitario, dette operazioni elementari
  - Il costo di una operazione (non elementare) è la somma delle operazioni elementari che la compongono

Operazioni elementari

Nell'analisi di complessità teorica vengono tipicamente definite come istruzioni elementari le seguenti:

- assegnazione
- operazione aritmetica
- confronto
- istruzione *return*
- istruzione *new*

Funzione di costo - esempio

```
/* somma gli elementi di un array di N interi */
public static int somma(int[] dati) {
    int i;
    int s;
    s = 0;
    for (i=0; i<dati.length; i++)
        s = s + dati[i];
    return s;
}
```

- Indichiamo con $t_{somma}(N)$ la funzione di costo del metodo *somma*;
- *N* indica la dimensione dell'input, che in questo caso è espressa come la dimensione dell'array *dati,* ossia come il numero di item (numeri interi) da sommare

Funzione di costo - esempio

```
/* somma gli elementi di un array di N interi */
public static int somma(int[] dati) {
    int i;
    int s;
    s = 0;
    for (i=0; i<dati.length; i++)
        s = s + dati[i];
    return s;
}
```

- *s = 0* è una operazione elementare - viene eseguita una volta
- *return s* è una operazione elementare - viene eseguita una volta

Quale è il costo computazionale dell'istruzione *for*?

Funzione di costo - esempio

```
for (i=0; i<dati.length; i++)
        s = s + dati[i];
```

Durante l'esecuzione di una istruzione *for*

- l'*inizializzazione* viene eseguita una sola volta
- il *corpo* viene eseguito *N* volte,
- la *condizione* viene valutata *N+1* volte
- l'*incremento* viene eseguito *N* volte

In questo caso:

- *i = 0* è una operazione elementare eseguita una volta
- *i < dati.length* è una operazione elementare eseguita *N+1* volte
- *i++* è una operazione elementare eseguita *N* volte
- *s = s + dati[i]* sono due operazioni elementari eseguite *N* volte

Funzione di costo - esempio

```
/* somma gli elementi di un array di N interi */
public static int somma(int[] dati) {
    int i;
    int s;
    s = 0;
    for (i=0; i<dati.length; i++)
        s = s + dati[i];
    return s;
}
```

La funzione di costo $t_{somma}(N)$ del metodo *somma* è quindi:

$$t_{somma}(N) = t_{s=0} + t_{i=0} + (N+1)\, t_{i<dati.length} + N\, t_{s= s + dati[i]} + N\, t_{i++} + \\ + t_{return\ s}$$

Funzione di costo - esempio

Quindi, in base al nostro modello di costo, si ha:

$$t_{somma}(N) = t_{s=0} + t_{i=0} + (N+1)\, t_{i<dati.length} + N\, t_{s= s + dati[i]} + N\, t_{i++} + \\ + t_{return\ s}$$

$$t_{somma}(N) = 1 + 1 + (N+1) + 2N + N + 1 = 4N + 4$$

Modello di costo e approssimazioni

Ogni modello di costo introduce delle approssimazioni (ad esempio, nella pratica la somma 1000 + 2000 è più costosa di 1 + 2)

Le approssimazioni introdotte sono comunque accettate nella teoria

Funzione di costo – un altro esempio

Vogliamo determinare la funzione di costo del seguente metodo che verifica se l'array *dati* di *N* elementi interi contiene un elemento di valore *chiave*

```
/* ricerca in un array di N interi */
public static boolean ricerca(int[] dati, int chiave) {
    int i;                  // per la scansione di dati
    boolean trovato;        // esito della ricerca
    trovato = false;
    for (i=0; i<dati.length; i++)
        if (dati[i]==chiave)
            trovato = true;
    return trovato;
}
```

Funzione di costo – un altro esempio

```
/* ricerca in un array di N interi */
public static boolean ricerca(int[] dati, int chiave) {
    int i;
    boolean trovato;
    trovato = false;
    for (i=0; i<dati.length; i++)
        if (dati[i]==chiave)
            trovato = true;
    return trovato;
}
```

eseguita 1 volta

eseguita N+1 volte

eseguita N volte

eseguita N volte

eseguita K volte (0≤K≤ N)

eseguita 1 volta

In questo caso, la funzione di costo del metodo *ricerca* è

$$t_{ricerca}(N) = 3N + K + 4 \qquad (0 \leq K \leq N)$$

dove K è il numero di elementi di *dati* che sono uguali a *chiave*

Caso migliore e caso peggiore

$$t_{ricerca}(N) = 3N + K + 4 \qquad (0 \leq K \leq N)$$

- $K = 0$ se nessun elemento di *dati* è uguale a *chiave*
  - in questo caso $t_{ricerca}(N) = 3N + 4$ è il costo del metodo *ricerca* nel caso migliore (nel senso di caso che conduce al costo minimo)

- $K = N$ se tutti gli elementi di *dati* sono uguali a *chiave*
  - in questo caso $t_{ricerca}(N) = 3N + N + 4 = 4N + 4$ è il costo del metodo *ricerca* nel caso peggiore (nel senso di caso che conduce al costo massimo)

Dipendenza dai valori dell'input

Con questa analisi, la complessità di un metodo può dipendere, quindi, non solo dalla dimensione dei dati di ingresso, ma anche dai valori

$$t_{ricerca}(N) = \begin{cases} 3N + 4 & \text{caso migliore} \\ 4N + 4 & \text{caso peggiore} \end{cases}$$

Analisi di caso peggiore

Per eliminare la dipendenza dal valore dei dati, ci si concentra tipicamente sulla sola
analisi del caso peggiore

- questo perché si vuol valutare quanto può costare al massimo l'esecuzione di un metodo

Calcolo esatto della funzione di costo

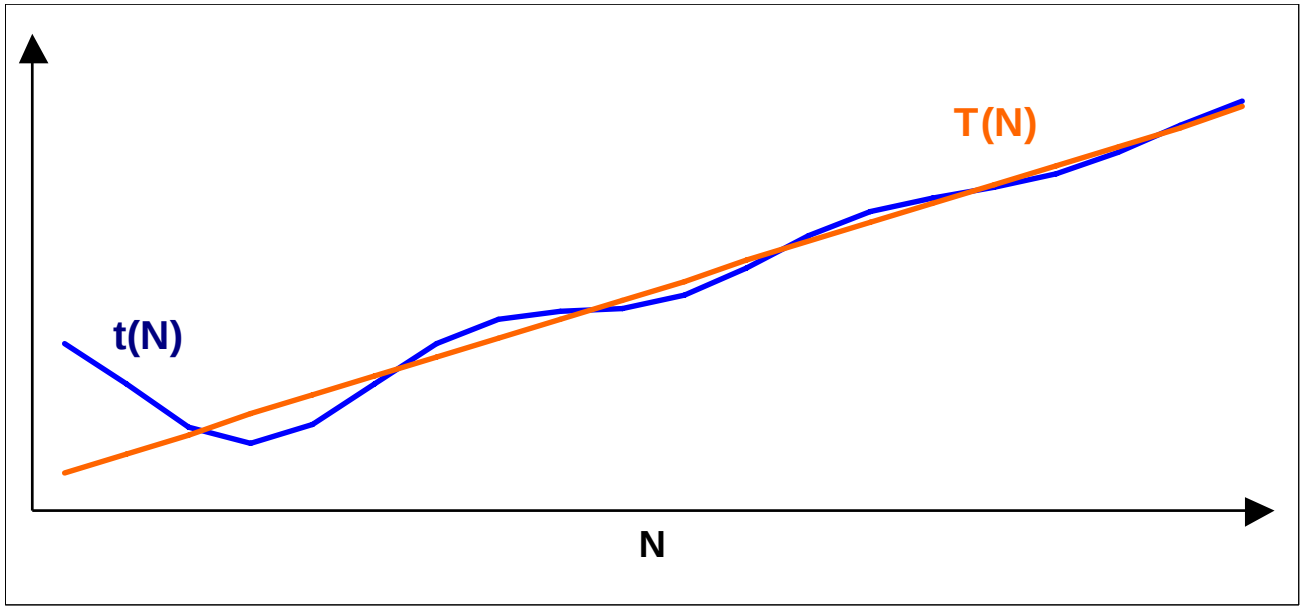
Calcolare in modo esatto una funzione di costo $t_M(N)$ può risultare difficile:

- il livello di dettaglio descritto dalla funzione di costo è veramente necessario?
- quanto è importante stabilire con precisione quante operazioni elementari vengono eseguite ?
- è possibile fare delle ulteriori semplificazioni sul modello di costo, senza alterare l'analisi in modo significativo?

Comportamento asintotico della funzione costo

Nella teoria, il calcolo esatto di una funzione di costo $t_M(N)$ viene semplificato trascurando i fattori costanti e i termini di ordine inferiore per $N \rightarrow +\infty$

In questo modo si valuta il comportamento asintotico $T_M(N)$ di $t_M(N)$ al crescere della dimensione dell'input



Analisi asintotica di complessità

L'analisi asintotica di complessità è una tecnica di analisi *approssimata*

- studia il comportamento asintotico della funzione di costo di un metodo
- i risultati sono approssimati, ma comunque significativi nella maggior parte dei casi

Perché l'analisi asintotica?

Perché ci interessa l'andamento asintotico della funzione di costo?

- l'andamento asintotico ci dice come peggiorano le prestazioni del metodo al crescere dell'input
- se la funzione di costo cresce molto velocemente allora le prestazioni diventeranno inaccettabili per valori relativamente piccoli dell'input
- se la funzione di costo cresce molto lentamente allora le prestazioni rimarranno accettabili anche per dimensioni dell'input molto grandi

Analisi asintotica

Consideriamo due metodi *M* ed *M'* con le seguenti funzioni di costo $t_M(N)=3N$ e $t_{M'}(N)=8N$

N	$t_M(N)$	$t_{M'}(N)$
10	30	80
20	60	160
40	120	320
80	240	640

Se la dimensione dell'input raddoppia il costo raddoppia in entrambi i casi

Analisi asintotica

Consideriamo un terzo metodo *M”* la cui funzione di costo è $t_{M”}(N)=2N^2$

N	$t_M(N)$	$t_{M'}(N)$	$t_{M”}(N)$
10	30	80	200
20	60	160	800
40	120	320	3200
80	240	640	12800

In questo caso se la dimensione dell’input raddoppia il costo quadruplica; la funzione di costo di *M”* quindi cresce più velocemente delle altre due al crescere di *N*

Analisi asintotica

Consideriamo due metodi *M* ed *M'* con le seguenti funzioni di costo $t_M(N)=N+3$ e $t_{M'}(N)=N+20$

N	$t_M(N)$	$t_{M'}(N)$
10	13	30
100	103	120
1000	1003	1020
10000	10003	10020
100000	100003	100020
1000000	1000003	1000020

Al crescere di *N* il contributo delle costanti additive diventa trascurabile

Notazione *O* grande

La notazione *O* grande è il principale strumento matematico per l'analisi asintotica

• Date le funzioni $f(N)$ e $g(N)$, diciamo che $f(N)$ *è* $O(g(N))$, se esistono due costanti positive C ed N_0 tali che:
$f(N) \leq C \bullet g(N) \quad \forall N \geq N_0$
(si scrive anche $f(N) = O(g(N))$ e si legge "*f è O grande di g*")

• Se $f(N) = O(g(N))$ allora la funzione f cresce (in modo asintotico) al più quanto la funzione g, a meno di fattori costanti e termini di ordine inferiore

Notazione *O* grande

La semplice regola per calcolare la funzione $g(N)$ è quella di trascurare i fattori costanti e i termini di ordine inferiore

$7N - 3$ è $O(N)$

$3N^2 + 25N - 37$ è $O(N^2)$

$\log_{10} N = (\log_2 N) / (\log_2 10) = O(\log_2 N)$

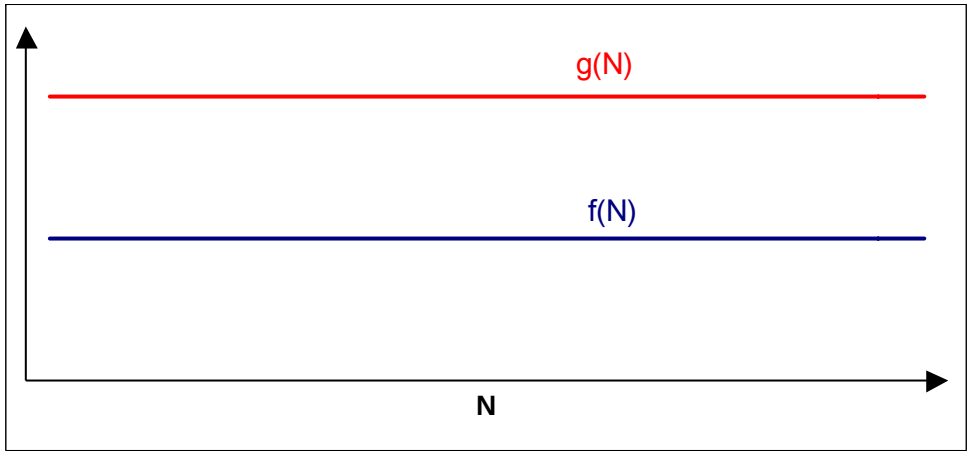
- possiamo quindi ignorare la base dei logaritmi e scrivere $\log N$

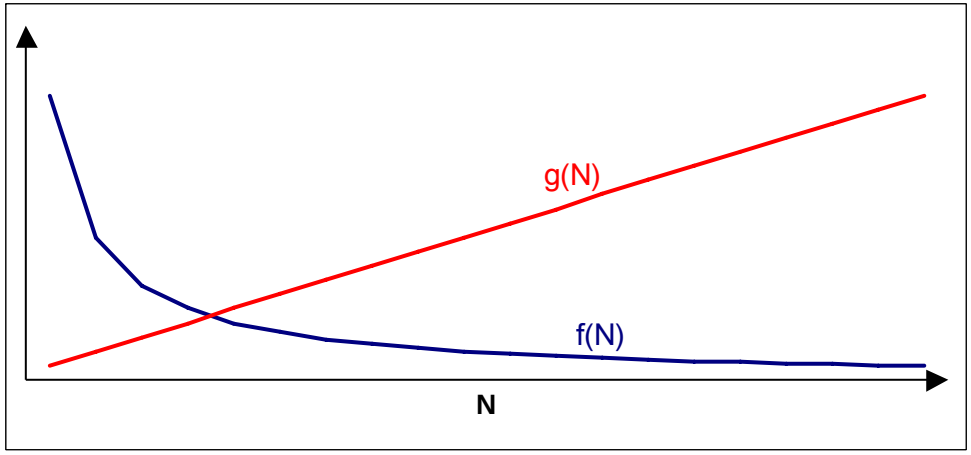
$8N^2 \log N + 5N^2 + 3N$ è $O(N^2 \log N)$

- osservazione — anche se $7N - 3$ è $O(N^k)$ per ogni $k \geq 1$ cerchiamo sempre di esprimere la *migliore approssimazione*, dicendo che $7N - 3$ è $O(N)$

Notazione *O* grande - esempi

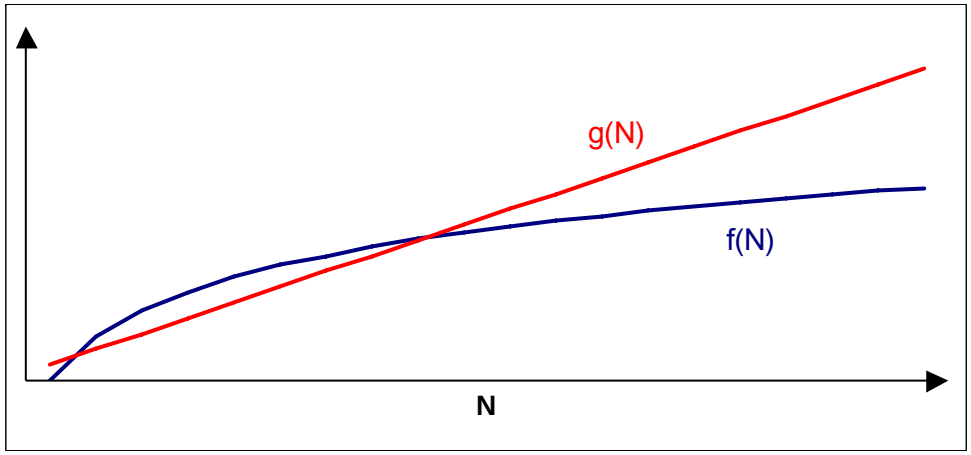
Esempi di funzioni *f* che sono *O(g)*

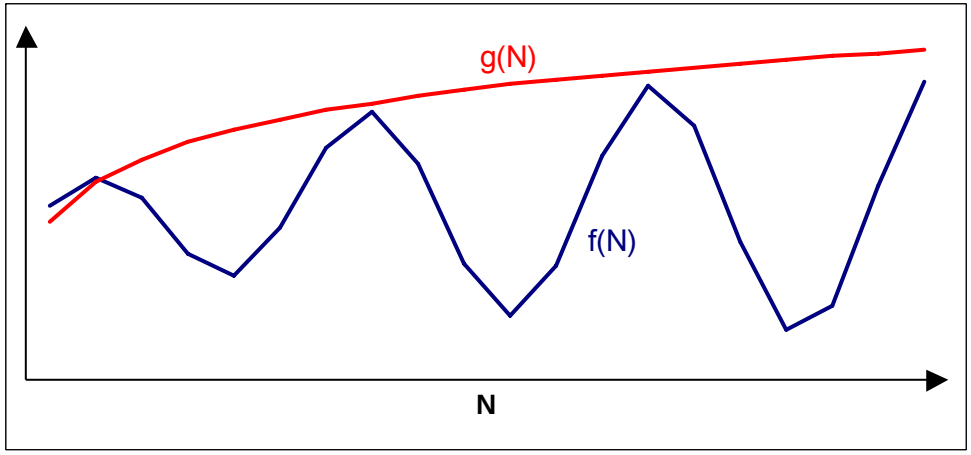




Notazione *O* grande - esempi

Esempi di funzioni *f* che sono *O(g)*





Altre notazioni

Nell'analisi asintotica, oltre alla notazione *O*, esistono altre notazioni molto utilizzate : Θ (teta grande) ed Ω (omega grande)

Per motivi di tempo, in questo corso ci concentriamo soltanto sull'uso della notazione *O*

Funzioni usate nell'analisi asintotica

Nell'analisi asintotica si fa riferimento ad alcune funzioni che hanno una forma particolarmente "semplice" e che vengono usate come "pietre di paragone"

- In genere, la funzione di costo asintotico di un metodo viene espressa come una di queste funzioni o come prodotto di queste funzioni

Pietre di paragone

Ecco le principali “pietre di paragone”

- *1* - funzione costante
  - se un metodo ha questa funzione di costo asintotico si dice che ha complessità costante
- *log N* - logaritmo in base 2
  - se un metodo ha questa funzione di costo asintotico si dice che ha complessità logaritmica
- *N* - funzione lineare
  - se un metodo ha questa funzione di costo asintotico si dice che ha complessità lineare

Pietre di paragone

- N^2 - funzione quadratica
  - se un metodo ha questa funzione di costo asintotico si dice che ha complessità quadratica
- 2^N - funzione esponenziale
  - se un metodo ha questa funzione di costo asintotico si dice che ha complessità esponenziale

Pietre di paragone - osservazioni

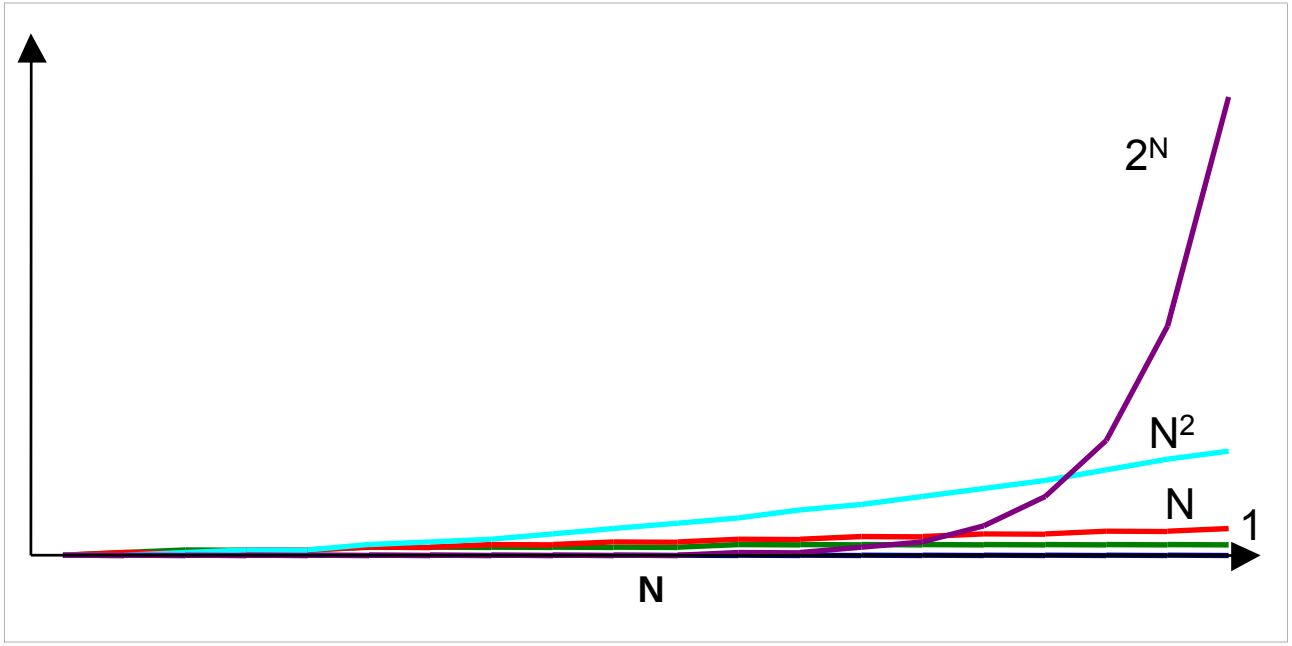
• queste funzioni sono state elencate in modo "crescente", ovvero ciascuna di queste funzioni è *O* grande della successiva e cresce asintoticamente meno della successiva

  – ad esempio, $\log N$ è $O(N)$

• le funzioni N e N^2 sono casi particolari delle funzioni polinomiali N^k (che crescono meno di 2^N, qualunque sia il valore di k)

Comportamento asintotico delle pietre di paragone

Ecco un grafico che mostra il comportamento asintotico delle funzioni “pietre di paragone”



Esempio di crescita dei Tempi

Passi

1-10ns

	n=2	n=5	n=10	n=100	n=1000
n	2	5	10	100	1000
n^2	4	25	100	10^4	10^6
n^3	8	125	1000	10^6	10^9
2^n	4	32	$\approx 10^3$	$\approx 10^{30}$	$\approx 10^{300}$
2^{n^2}	16	$\approx 3 \times 10^7$	$\approx 10^{30}$	$\approx 10^{3000}$	$\approx 10^{300000}$

Tempi

	n=2	n=5	n=10	n=100	n=1000
n	20 ns	50 ns	100 ns	1 μs	10 μs
n^2	40 ns	250 ns	1 μs	100 μs	10 ms
n^3	80 ns	1.25μs	10 μs	10 ms	10 sec
2^n	40 ns	320 ns	$\approx$ 10 μs	$\approx 10^{14}$ anni	$\approx 10^{284}$ anni
2^{n^2}	160ns	0.3 s	$\approx 10^{14}$anni	$\approx 10^{3000}$ anni	$\approx 10^{300000}$ anni

Analisi asintotica di complessità - esempio

Considera questo metodo per verificare se gli elementi di un array *dati* di interi sono tutti distinti

```
/* verifica se gli elementi di dati sono tutti distinti*/
public static boolean distinti(int[] dati) {
    int i,j;                  // per la scansione di dati
    boolean esisteCoppia;  // coppia di elementi uguali
    esisteCoppia = false;
    for (i=0; i<dati.length; i++)
        for (j=i+1; j<dati.length; j++)
            if (dati[i]==dati[j])
                esisteCoppia = true;
    return !esisteCoppia;
}
```

– il *caso peggiore* per questo metodo si verifica quando tutti gli elementi sono uguali

Analisi asintotica di complessità - esempio

Ipotesi — la lunghezza di *dati* è *N*, gli elementi sono *tutti uguali*

```
/* verifica se gli elementi di dati sono tutti distinti*/
public static boolean distinti(int[] dati) {
    int i,j;                    // per la scansione di dati
    boolean esisteCoppia;  // coppia di elementi uguali
    esisteCoppia = false;              // 1 volta
    for (i=0; i<dati.length; i++)   // il corpo N volte
        for (j=i+1; j<dati.length; j++)   //N-(i+1) volte
            if (dati[i]==dati[j])          //N-(i+1) volte
                esisteCoppia = true;       //N-(i+1) volte
    return !esisteCoppia;          // 1 volta
}
```

La funzione di costo è

$$t(N) = 4 + 2N + \sum_{i=0}^{N-1} (4N - 2 - 4i) = 2\,N^2 + 2N + 4$$

L'andamento asintotico della funzione di costo è

$$T(N) = O(N^2)$$

Analisi asintotica di complessità

La notazione *O* è solo una notazione matematica usata per trasformare una funzione di costo nella funzione più semplice che la approssima in modo asintotico e a meno di fattori costanti

- è chiaramente inutile fare un'analisi asintotica a partire dalla funzione di costo calcolata in modo preciso
- si può considerare un modello di costo che permette di calcolare direttamente il comportamento asintotico della funzione di costo

Un modello di costo asintotico

Istruzione	**Costo asintotico dell'istruz.**
Operazione elementare	1
blocco — *{ s_1 s_2 ... s_K }*	max (T_{s1} , T_{s2} , …., T_{sk})
istruzione condizionale — *if (cond) s_1 else s_2*	max(T_{cond} , T_{s1}) – se *cond* è *true* max(T_{cond} , T_{s2}) – se *cond* è *false*
istruzione ripetitiva — *while (cond) s* — nell'ipotesi che il blocco *s* sia eseguito *N* volte	$T_{cond}^{(1)} + T_s^{(1)} + T_{cond}^{(2)} + \; + T_s^{(2)} + \ldots + T_{cond}^{(N)} + T_s^{(N)} + T_{cond}^{(N+1)}$
invocazione di metodo o costruttore — *r.M(p_1, ..., p_K)*	max(T_r , T_{p1} , ….. , T_{pk} , T_M)

Operazione dominante

Sia *M* un metodo la cui funzione di costo è $t_M(N)$

– una operazione o istruzione d di *M* è una operazione dominante se risulta $t_M(N) = O(t_M^d(N))$, dove $t_M^d(N)$ è il costo della sola operazione d nell'ambito di *M* (cioè il costo di una singola esecuzione di d per il numero di volte in cui d viene eseguita in *M*)

– nell'ambito dell'esecuzione di *M* un metodo può avere una o più operazioni dominanti, ma anche nessuna

Operazione dominante

Il costo asintotico di un metodo *M* che ha un'operazione dominante *d* è pari al costo asintotico di *d* nell'ambito di *M*

– trovare una operazione dominante di un metodo aiuta a calcolare il costo asintotico del metodo

– solitamente, le operazioni dominanti di un metodo appartengono alle *istruzioni di controllo più nidificate del metodo*

Operazione dominante - esempio

Considera nuovamente questo metodo il cui costo è $2N^2 + 2N + 4$

```
/* verifica se gli elementi di dati sono tutti distinti*/
public static boolean distinti(int[] dati) {
    int i,j;                    // per la scansione di dati
    boolean esisteCoppia;  // coppia di elementi uguali
    esisteCoppia = false;           // 1 volta
    for (i=0; i<dati.length; i++)   // il corpo N volte
        for (j=i+1; j<dati.length; j++)   //N-(i+1) volte
            if (dati[i]==dati[j])         //N-(i+1) volte
                esisteCoppia = true;      //N-(i+1) volte
    return !esisteCoppia;         // 1 volta
}
```

La condizione dati[i]==dati[j] è una operazione dominante

- viene valutata $\sum_{i=0}^{N-1} (N - i - 1) = (N^2 - N)/2 = O(N^2)$ volte
- ogni valutazione ha costo unitario

Complessità di algoritmi

Le tecniche di analisi di complessità, che sono state introdotte per l'analisi dei metodi, possono essere usate anche per l'analisi di complessità degli algoritmi

- un algoritmo è un procedimento per risolvere un problema
- ogni metodo implementa un algoritmo, ed uno stesso algoritmo può essere implementato da metodi diversi
- il metodo più efficiente per implementare un algoritmo definisce la complessità dell'algoritmo

Complessità di algoritmi

In genere viene studiata la complessità degli algoritmi

- viene considerato un modello di costo generico per un linguaggio di programmazione astratto
- non sono considerati tutti i dettagli della programmazione in uno specifico linguaggio
- le approssimazioni fatte sono solitamente paragonabili al trascurare fattori costanti o termini di ordine inferiore

Complessità di algoritmi - esempio

Considera ad esempio il problema di decidere se gli elementi di un array di interi sono tutti uguali

- primo algoritmo
  - ipotizzo che gli elementi dell'array sono tutti uguali
  - confronto ogni elemento con ogni altro elemento di indice maggiore; se ho trovato almeno una coppia di elementi diversi, allora decido che l'ipotesi è falsa altrimenti decido che è vera
- secondo algoritmo
  - ipotizzo che gli elementi dell'array sono tutti uguali
  - confronto ogni elemento con l'elemento successivo; se ho trovato almeno una coppia di elementi consecutivi diversi, allora decido che l'ipotesi è falsa altrimenti decido che è vera

Qual è la complessità dei due algoritmi ?

Complessità di algoritmi - esempio

- primo algoritmo
  - l'operazione dominante è il confronto tra coppie di elementi
  - devo confrontare ciascuno elemento dell'array con ciascun altro elemento di indice maggiore
  - la complessità è quadratica
- secondo algoritmo
  - l'operazione dominante è il confronto tra coppie di elementi
  - devo confrontare ciascun elemento con il successivo
  - la complessità è lineare

Posso concludere che il secondo algoritmo è asintoticamente migliore del primo

Glossario dei termini principali

Termine	Significato
Tempo di esecuzione di un metodo	Il tempo richiesto per la sua esecuzione
Analisi di complessità di un metodo	Valutazione del tempo di esecuzione di un metodo
Funzione di costo di un metodo	Funzione che esprime il tempo di esecuzione di un metodo in termini del numero di operazioni elementari che vanno svolte durante l'esecuzione del metodo, e in modo parametrico rispetto alla dimensione dei dati di ingresso
Modello di costo	Modello in cui a ciascuna istruzione del linguaggio di programmazione usato viene assegnato un costo. In genere a ciascuna operazione elementare viene assegnato costo unitario
Analisi di caso peggiore	Determinazione della funzione di costo nel caso peggiore, ovvero con riferimento a quei valori per l'insieme dei dati di ingresso per cui il costo di esecuzione, a parità di dimensioni, è massimo
Funzione di costo asintotica	Funzione che approssima la funzione di costo di un metodo quando la dimensione dei dati cresce in modo asintotico
Analisi asintotica di complessità	Tecnica di analisi della complessità che studia il comportamento asintotico della funzione di costo di un metodo
Operazione dominante di un metodo	Operazione il cui costo è dello stesso ordine di grandezza del costo complessivo del metodo