La complessità di calcolo

Valutazione dell'efficienza dei programmi

L'analisi della complessità di calcolo è lo studio dell'**efficienza** di algoritmi e programmi.

Obiettivo: trovare algoritmi (e quindi programmi) più efficienti per un dato problema.

Risorse considerate:

- tempo di elaborazione (è la risorsa più importante e l'unica che consideriamo)
- quantità di memoria occupata

Prima definizione (intuitiva) di efficienza: Un programma è più efficiente di un altro se la sua esecuzione richiede meno tempo di calcolo.

Problema: misurare il tempo (ad es. in secondi) non è attendibile. Bisogna tenere conto delle **condizioni** in cui si effettuano le prove per confrontare i due programmi:

- elaboratore (e sistema operativo)
- compilatore utilizzato
- dati in ingresso
- significatività dei dati in ingresso (più esecuzioni con dati diversi)

Non otteniamo un criterio oggettivo (persone diverse giungono a conclusioni diverse).

Funzione di costo

- Per ottenere un criterio oggettivo cerchiamo di esprimere il costo di un programma attraverso una funzione.
- Il costo dipende (quasi) sempre dai dati in ingresso.
- Il fattore determinante è la **dimensione dei dati in ingresso** (intesa come la memoria necessaria per memorizzarli).

  Esempio: Ordinamento di un vettore di 10 o 1000 elementi.

$\Longrightarrow$ L'argomento della funzione di costo è la dimensione dei dati in ingresso.

N.B. Tipicamente il costo di esecuzione dipende non solo dalla dimensione dei dati, ma anche dai particolari dati.

Assunzioni per calcolare il costo delle singole istruzioni:

- istruzione semplice (assegnazione, lettura, scrittura):
  costo 1 (*)
- istruzione composta:
  somma dei costi delle istruzioni componenti
- istruzione di ciclo:
  costo totale della condizione $+$ costo totale del corpo
  $=$ numero-iterazioni $*$ ($1+$ costo di una esecuzione del corpo)
- istruzione condizionale:
  costo della condizione $+ \begin{cases} \text{costo del ramo then, se la condizione è vera} \\ \text{costo del ramo else, se la condizione è falsa} \end{cases}$
- attivazione di funzione:
  costo di tutte le istruzioni che la compongono (eventualmente tenendo conto di attivazioni al suo interno)

L'ipotesi (*) è **semplificativa**.

Esempio: sia t_1 il costo di `x = x+1;`
sia t_2 il costo di `x = 2*(x+y) / (z*3*q);`

Non è realistico assumere che $t_1 = t_2 = 1$.
Però esiste una costante c tale che $c \cdot t_1 > t_2$.

Quindi, **(*) è vera a meno di un fattore moltiplicativo costante** (ovvero, indipendente dalla dimensione dei dati in ingresso).

Otteniamo una valutazione:

- indipendente dal calcolatore, compilatore, ...
- approssimata per un fattore moltiplicativo costante
- indipendente dal linguaggio di programmazione (si può fare riferimento direttamente all'algoritmo)

Esempio: Ricerca esaustiva di un elemento in un vettore

```
bool RicercaEsaustiva(TipoVettore A, TipoElemVettore elem, int n, int *posiz)
{ int i = -1;     bool trovato = FALSE;

  do i++; while (i < n-1 && A[i] != elem);
  if (A[i] == elem) { *posiz = i;
                      trovato = TRUE; }
  return trovato;
}
```

Costo di esecuzione del programma dipende dalla posizione dell'elemento cercato:

se è il 1^o:	1 iterazione	$\Longrightarrow$ costo	$= 2 + 2 \cdot 1 + 1 + 2 + 1 = 8$
se è il 2^o:	2 iterazioni	$\Longrightarrow$ costo	$= 2 + 2 \cdot 2 + 1 + 2 + 1 = 10$
se è l'i^o:	i iterazioni	$\Longrightarrow$ costo	$= 2 + 2 \cdot i + 1 + 2 + 1 = 6 + 2 \cdot i$
se non c'è	n iterazioni	$\Longrightarrow$ costo	$= 2 + 2 \cdot n + 1 + 1 \quad = 4 + 2 \cdot n$

Si distinguono:

- il caso migliore (meno costoso) (Es.: è il 1^o)
- il caso peggiore (più costoso) (Es.: è l'ultimo o non c'è)

Per ottenere una valutazione del costo che dipende solo dalla dimensione dell'input e non dai particolari dati in ingresso **si fa riferimento al caso peggiore**.

Riassumendo, esprimiamo il **costo di un algoritmo** (o di un programma)

- come funzione della dimensione dell'input
- a meno di un fattore moltiplicativo costante
- facendo riferimento al caso peggiore

Classificazione degli algoritmi secondo il loro costo

- algoritmo **costante**: costo è c_0
- algoritmo **lineare**: costo è $c_1 \cdot n + c_0$
- algoritmo **quadratico**: costo è $c_2 \cdot n^2 + c_1 \cdot n + c_0$
- algoritmo **polinomiale**: costo è $c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \cdots + c_1 \cdot n + c_0$
- algoritmo **logaritmico**: costo è $c \cdot \log_2 n$
- algoritmo **esponenziale**: costo è $c \cdot 2^n$

dove: n è la dimensione dell'input

c, k, c_0, c_1, ..., c_k, sono costanti indipendenti da n

La notazione O

Definizione Un algoritmo (o un programma) ha **costo** (o **complessità**) $O(f(n))$ se il numero di istruzioni $t(n)$ che devono essere eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) < c_1 \cdot f(n) + c_2$$

per ogni n e per opportune costanti c_1 e c_2 (indipendenti da n).

Quindi:

$O(1)$	... costo costante (indipendente dalla dimensione dell'input)
$O(n)$	... costo lineare
$O(n^2)$	... costo quadratico
$O(n^k)$	... costo polinomiale (k è una costante indipendente da n)
$O(\log_2 n)$	... costo logaritmico
$O(2^n)$	... costo esponenziale

Attenzione: **Non teniamo conto di costanti moltiplicative.**

Esempio: $O(n)$, sia se il costo è $2 \cdot n$ che se il costo è $1000 \cdot n$.

Quindi, stiamo facendo una **valutazione semplificata** che però funziona bene.

- motivo teorico: per input sufficientemente grandi la dimensione dell'input prevale sempre sulle costanti moltiplicative
- motivo pratico: le costanti moltiplicative sono quasi sempre piccole.

Questo significa che in pratica, per quanto riguarda il costo di un algoritmo che risolve un certo problema:

$O(\log_2 n)$	è molto meglio di $O(n)$
$O(n)$	è meglio di $O(n \cdot \log_2 n)$
$O(n \cdot \log_2 n)$	è meglio di $O(n^2)$
$O(n^2)$	è meglio di $O(n^3)$
...	
$O(n^k)$	è molto molto molto meglio di $O(2^n)$.

Istruzione dominante

Il fatto di trascurare le costanti moltiplicative permette di semplificare ulteriormente la valutazione del costo di un algoritmo.

Dato un algoritmo (o un programma) con costo $O(f(n))$, un'**istruzione** è **dominante** se, per ogni intero n, viene ripetuta nel caso peggiore di input di dimensione n, un numero di volte $d(n)$ tale che

$$c \cdot d(n) > f(n)$$

per un'opportuna costante c (che non dipende da n).

Intuitivamente, un'istruzione dominante è una di quelle che vengono eseguite più volte tra tutte le istruzioni dell'algoritmo (tipicamente, le istruzioni nei cicli più interni).

Esempio: Nella ricerca esaustiva (file `riceordi/ricesau.c`): `i++`

$\Longrightarrow$ Per valutare il costo di un algoritmo è sufficiente identificare un'istruzione dominante e valutare il costo di esecuzione di tale istruzione.