Università Roma La Sapienza
Corsi di Laurea
Informatica/Tecnologie Informatiche

Quanto costa?

Prof. Stefano Guerrini
guerrini@di.uniroma1.it

Programmazione II (can. P-Z)
A.A. 2005-06

Costo di esecuzione

- Cosa contribuisce al costo di esecuzione di un programma?
- Tutte le risorse utilizzate:
  - tempo di occupazione della CPU
  - memoria utilizzata
  - numero di chiamate al sistema operativo
  - numero di accessi alle periferiche
  - ...

Spazio e tempo

- Non consideriamo altri costi come:
  - costi di progettazione;
  - costi di implementazione (tra cui, ad esempio, il numero di ore di lavoro di un programmatore);
  - costi di messa in opera;
  - ...
- Restringiamo l'analisi a due aspetti principali
  - **spazio** o memoria necessaria
  - **tempo** di esecuzione

Spazio

- La quantità di memoria richiesta dal programma.
- Comprende la memoria necessaria alla:
  - acquisizione dei dati di input;
  - memorizzazione dei dati temporanei necessari per l'esecuzione del programma;
  - produzione dei dati di output.
- In un'analisi accurata occorre distinguere tra occupazione della
  - **memoria centrale**
  - **memoria secondaria** o di massa.

Tempo

- Il tempo necessario per completare il programma.
- **Approccio sperimentale**:
  eseguiamo il programma su varie istanze di input e riportiamo i tempi di esecuzione su di un grafico.
- Questioni:
  - su quale computer eseguire il programma?
  - su quali dati?
  - rispetto a quale parametro voglio analizzare la dipendenza del tempo di esecuzione?

Approccio Sperimentale /1

- L'analisi dipende dal computer su cui eseguo i test.
  - Se cambio il computer con uno più veloce, o con una architettura innovativa, devo eseguire nuovamente i test?
- L'analisi dipende fortemente da come ho scelto i dati di input.
  - Se il programma è non banale non posso verificare tutti gli input possibili.
  - Con diverse scelte dei dati di input otterrei gli stessi risultati?

Approccio Sperimentale /2

- Per eseguire l'analisi devo avere a disposizione il sistema su cui verrà eseguito il programma.
- Se il programma (o la funzione) è parte di un programma più grande, dovrei avere già implementato e disponibile l'intero programma.
- L'analisi può essere eseguita solo dopo aver implementato il programma, quindi
  - per verificare l'efficienza di due soluzioni dovrei prima implementarle entrambe e poi confrontare i risultati delle prove sperimentali.

Cosa cerchiamo

- Un modo per analizzare il costo di esecuzione di un programma
  - indipendente dall'efficienza del sistema su cui andrò ad eseguirlo;
  - che mi consenta di poter confrontare l'efficienza di due soluzioni di un problema prima di averle effettivamente implementate e che, di conseguenza, mi permetta di implementare solo la più efficiente;
  - semplice, che trascuri dettagli inutili.

La funzione costo /1

- Per rappresentare su di un grafico i risultati delle prove sperimentali devo fissare:
  - la variabile dipendente (l'ordinata), e.g., il tempo di esecuzione o l'occupazione massima di memoria
  - la variabile indipendente (l'ascissa), ovvero, il parametro rispetto a cui analizzo l'andamento del costo
- Qual è il parametro dei dati di input che devo scegliere per l'ordinata?

La funzione costo /2

- Prendiamo il programma che calcola il fattoriale, la scelta naturale per l'ascissa sembra essere il valore dell'input.
- Se prendiamo un programma che calcola il minimo di un vettore o che cerca un elemento in un vettore, qual è il parametro che ci interessa?
  - La grandezza del valore da cercare?
  - La grandezza dei valori nel vettore?
  - Oppure, la lunghezza del vettore?

La funzione costo /3

- La scelta più naturale per la variabile indipendente sembra essere la
  dimensione dei dati di input
  ad esempio, espressa in termini di celle di memoria o byte occupati.
- **Funzione costo**
  Indicando con x la dimensione dell'input
  f(x) = tempo di esecuzione
  analogamente se misuro lo spazio (memoria)

La scelta dei dati /1

- Per scegliere i dati di input è sufficiente guardare la loro dimensione?
- Esempio: calcolo del minimo di un vettore *V* di lunghezza *N*
  - il tempo di esecuzione è indipendente dagli elementi presenti nel vettore e dall'ordine in cui si trovano
  - per determinare il minimo devo analizzare necessariamente tutti gli elementi di *V*
  - devo eseguire *N* confronti

La scelta dei dati /2

- Esempio: ricerca di un elemento *e* in un vettore *V* di lunghezza *N* (vettore non ordinato)
- Il tempo varia notevolmente a seconda degli elementi presenti nel vettore e di come sono disposti:
  - se *e* non è presente in *V*, per scoprirlo devo sempre analizzare tutti gli *N* elementi di *V*
  - se *e* è il primo elemento di *V* che l'algoritmo va ad analizzare (e.g., l'elemento nella prima cella di *V*) è sufficiente un solo confronto

Caso peggiore

- Supponiamo di voler realizzare un ponte, il progetto della struttura e il calcolo del calcestruzzo saranno fatti tenendo conto degli sforzi massimi a cui il ponte sarà sottoposto.
- Anche per i programmi, una scelta ragionevole per i dati di input è il **caso peggiore**, ovvero, il caso corrispondente al massimo tempo di esecuzione.
- Il costo così ottenuto è un upper-bound al tempo di esecuzione di un'altra istanza della stessa dimensione.

Esplosione esponenziale /1

Confrontiamo il costo della versione di Fibonacci con doppia ricorsione con quello della versione ricorsiva efficiente (o con quella iterativa).

```
int fib (int n) {
  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

$c_1+2t(n-2) \leq t(n) \leq c_1+2t(n-1)$

$(c_1+c_0)2^{n/2}-c_1 \leq t(n) \leq (c_1+c_0)2^{n-1}-c_1$

costo esponenziale

- Proviamo a calcolare il tempo di esecuzione $t(n)$ di fib(n)
- Per $n<2$, la funzione esegue il test e termina nel ramo then. Indichiamo con c_0 il tempo di esecuzione di questi casi.
  $t(0) = t(1) = c_0$
- Per $n \geq 2$, la funzione esegue il test, richiama ricorsivamente fib(n-1) e fib(n-2), somma i risultati ottenuti e termina.
  $t(n) = c_1+t(n-1)+t(n-2)$

Esplosione esponenziale /2

```
void fib_lin (int n, int *p0, int *p1) {
  if (n > 0)  {
    fib_lin(n-1, p0, p1);
    int aux = *p1;
    *p1 = *p0 + *p1;
    *p0 = aux;
  } else
    *p1 = 1, *p0 = 0;
}
```

- Indichiamo con b_0 il costo di esecuzione del caso base $n=0$ e con b_1 il costo delle istruzioni eseguite nel caso ricorsivo $n>0$ senza comprendere la chiamata ricorsiva.

$t(0) = b_0$

$t(n+1) = b_1 + t(n)$

$t(n) = b_1\, n + b0$

costo lineare

Esplosione esponenziale /3

- Nel caso della doppia ricorsione il programma è molto meno efficiente, cresce esponenzialmente, mentre l'altro programma cresce linearmente.
- Per renderci concretamente conto della differenza di efficienza dei programmi, prendiamo
  - c_1+c_0 e b_1 dell'ordine di 10^{-8} *s*
  - pari a soli *40* cicli di una CPU da *4 GHz*.

Esplosione esponenziale /4

n	lineare	doppia ricorsione
10	10^{-7} s	10^{-5} s
20	$2\ 10^{-7}$ s	10^{-2} s
30	$3\ 10^{-7}$ s	10 s
40	$4\ 10^{-7}$ s	3 h
50	$5\ 10^{-7}$ s	130 g
60	$6\ 10^{-7}$ s	366 anni
70	$6\ 10^{-7}$ s	374.000 anni

Le costanti

- In questo caso le costanti erano dello stesso ordine di grandezza.
- Supponiamo però che le costanti fossero diverse, ad esempio, $(c_0 + c_1) \sim 10^{-3}$ $b_0 = 10^{-11}$.
  - per $n = 70$ il programma con costo esponenziale impiega **solo 374 anni** per terminare!
- Il valore delle costanti non è molto significativo per sapere cosa succede quando la dimensione dell'input cresce.
- Anche un incremento di efficienza dei computer di 1000 volte non cambia granché (asintoticamente).

Ricerca in un vettore

- Confrontiamo due differenti algoritmi per la
  - ricerca di un elemento, una chiave, in un vettore,
  - sotto l'ipotesi che sulle chiavi sia definito un ordinamento
  - e il vettore delle chiavi sia ordinato.

Ricerca lineare

- L'algoritmo più semplice per ricercare la chiave nel vettore esegue una scansione lineare.
- Il caso peggiore è quello in cui la chiave è assente.

```
int linearSearch(int array[], int key, int size) {
  int n;

  for (n = 0; n <= size - 1; ++n)
    if (array[n] == key)
      return n;

  return -1;
}
```

La funzione tempo varia con legge lineare

$t(n) = c_1 n + c_0$

Ricerca binaria /1

- La ricerca lineare non sfrutta affatto l'ordinamento del vettore.
- Sarebbe come se, per cercare una parola in un dizionario o un numero in un elenco telefonico, scorressimo ordinatamente tutte le voci a partire dalla prima fino all'ultima o fino a quella cercata.
- Nella realtà, cercheremo di aprire il dizionario o l'elenco del telefono a una pagina *p* in cui pensiamo possa essere l'elemento da cercare e, se non lo troviamo, determinare in base all'ordine alfabetico se la chiave si trova prima o dopo della pagina *p*, applicando nuovamente il procedimento di ricerca da un solo lato di *p*.

Ricerca binaria /2

- Se prendiamo un qualsiasi elemento in posizione *m* del vettore *V* e lo confrontiamo con la chiave *k*, possiamo dire che
  - *k* non è in una posizione $i < m$, se $k > V[m]$
  - *k* non è in una posizione $i > m$, se $k < V[m]$
  - per trovare *k* possiamo procedere applicando ricorsivamente la tecnica di ricerca precedentemente vista ristretta alla parte di *V* in cui non posso escludere che si trovi *k*.

Ricerca binaria /3

- Nel caso del dizionario, la scelta di *m* è fatta in base all'iniziale della parola:
  - se l'iniziale è la lettera s, apriremo più vicino al fondo del dizionario
  - se l'iniziale è la lettera c, apriremo più vicino all'inizio del dizionario
  - se l'iniziale è la lettera m, apriremo circa al centro del dizionario
  - e così via...

Ricerca binaria /4

- Se non abbiamo idea del valore minimo e massimo degli elementi presenti nel vettore,
- né della distribuzione statistica che gli elementi presenti nel vettore hanno nell'intervallo compreso tra i valori minimo e massimo che le chiavi possono assumere,
- la scelta più naturale per il punto *m* in cui fare il confronto con la chiave è il centro del vettore.
- In questo modo, ad ogni confronto, dimezziamo la parte di vettore in cui rimane da eseguire la ricerca.

Ricerca binaria: iterativa

```
int binarySearch(int b[], int key, int low, int high) {
  int middle;

  while (low <= high) {
    middle = (low + high) / 2;
    if (key == b[middle])  /* trovata */
      return middle;
    else if (key < b[middle]) /* cerca nella meta' inferiore */
      high = middle - 1;
    else /* cerca nella meta' superiore */
      low = middle + 1;
   }

  return -1;   /* non trovata */
}
```

Ricerca binaria: ricorsiva

```
int binarySearchRic(int b[], int key, int low, int high) {
  int middle = (low + high) / 2;

  if (low > high) /* intervallo vuoto: chiave non trovata */
    return -1;
  if (key == b[middle])  /* trovata */
    return middle;
  else if (key < b[middle]) /* cerca nella meta' inferiore */
    high = middle - 1;
  else /* cerca nella meta' superiore */
    low = middle + 1;
  /* cerca nel nuovo intervallo [high,low] */

  return binarySearch(b, key, low, high);
}
```

Ricerca binaria: complessità

- Parte intera del logaritmo (base 2)
  $$lg(n) = min \{ i \mid 2^i \leq n < 2^{i+1} \}$$
- Il caso peggiore è ancora quello in cui la chiave è assente.
- Dopo *i* confronti, nel caso peggiore, il vettore da ricercare è ridotto ad una sotto-sequenza del vettore iniziale di lunghezza $n/2^i$.
- L'algoritmo termina al massimo dopo $lg(n)+1$ passi.
- La funzione costo è logaritmica
  $$t(n) = b_1 \, lg(n) + b_0$$

Esempio: controllo ortografico /1

- Supponiamo di voler utilizzare i precedenti algoritmi per eseguire il controllo ortografico
  - di una monografia di circa 1.000 pagine
  - con circa 1.000 parole per pagina
  - usando un dizionario con circa 36.000 lemmi
  - supponendo che ogni confronto tra una parola nel dizionario e quella da cercare costi 10^{-6} s
- Per ogni parola della monografia dovremo eseguire una ricerca all'interno del dizionario. In totale, 10^6 ricerche.

Esempio: controllo ortografico /2

- **Ricerca lineare**
  supponiamo di memorizzare il dizionario in un vettore e di usare la ricerca lineare per cercare le parole nel dizionario.
  - Applicando il caso peggiore dobbiamo eseguire
    10^6 **36.000 confronti**
  - per un tempo totale di
    $10^6\ 36.000\ 10^{-6}$ s = 36.000 s = **10 h**
  - Anche supponendo che ogni ricerca non scorra tutto il dizionario, ma che in media ne scorra solo la metà, il tempo totale rimane di **5 h**.

Esempio: controllo ortografico /3

- **Ricerca binaria**
  Supponiamo di memorizzare il dizionario in un vettore ordinato e di usare la ricerca lineare per cercare le parole nel dizionario.
  - Applicando il caso peggiore dobbiamo eseguire
    $10^6\ lg(36.000) \sim$ **10^6 15 confronti**
  - per un tempo totale di
    $10^6\ 15\ 10^{-6}$ s = **15 s**
- Il tempo è dell'ordine dei secondi. Anche con un miglioramento di un fattore 2, o 10 nella velocità del confronto, nel caso della ricerca lineare, il tempo totale rimane dell'ordine delle ore.

Analisi asintotica /1

- Nei casi precedentemente analizzati l'enorme differenza tra l'andamento delle funzioni da confrontare era immediatamente evidente: esponenziale/lineare o lineare/logaritmico.
- Prendiamo $t_1(n) = c_1 n$ e $t_2(n) = c_2 n^2$ e analizziamo il rapporto $f(n) = t_2(n)/t_1(n)$
  - supponendo $c_1/c_2 = 100$
  - $f(10^k) = 10^{k-2}$
  - $f(1) = 0{,}01 \quad f(100) = 1 \quad f(10.000) = 100$
  - inizialmente t_1 è migliore di t_2, poi t_2 comincia a crescere molto più rapidamente di t_1.

Analisi asintotica /2

- A seconda delle costanti, anche una funzione con andamento esponenziale può avere inizialmente valori più piccoli, ma
- al crescere dell'input, prima o poi supera quella con andamento lineare (o con qualsiasi andamento polinomiale).
- L'**analisi asintotica** analizza cosa succede quando l'input cresce, più precisamente quando la dimensione dell'input tende all'infinito.

Analisi asintotica /3

- Nell'analisi asintotica le costanti possono essere ignorate, conta solo la velocità di crescita della funzione.
- Anzi, conta solo la componente della funzione che cresce più velocemente
  - e.g., di un polinomio, conta solo il suo grado
- Siccome le costanti dipendono dal sistema su cui si esegue il programma, l'indipendenza dalle costanti garantisce l'indipendenza dal calcolatore e dal compilatore.

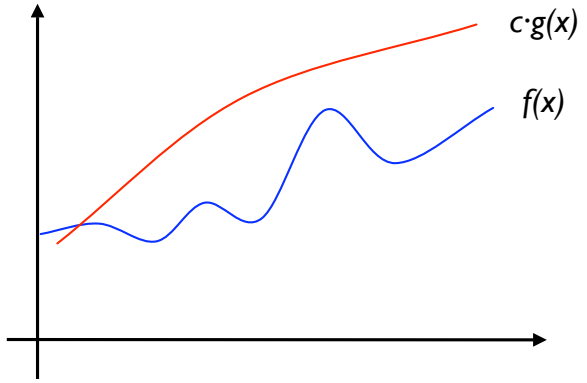
Notazione asintotica

- Spesso non conosciamo l'andamento esatto della funzione da analizzare, ma solo delle funzioni che la approssimano
  - garantendo che la funzione non supera certi valori: **upper-bound**
  - garantendo che la funzione non è inferiore a certi valori: **lower-bound**
  - garantendo che la funzione varia entro un certo intervallo.
- Introduciamo una notazione per dire quando i precedenti bound sono verificati asintoticamente.

Upper-Bound

- La funzione f cresce **al più come** g

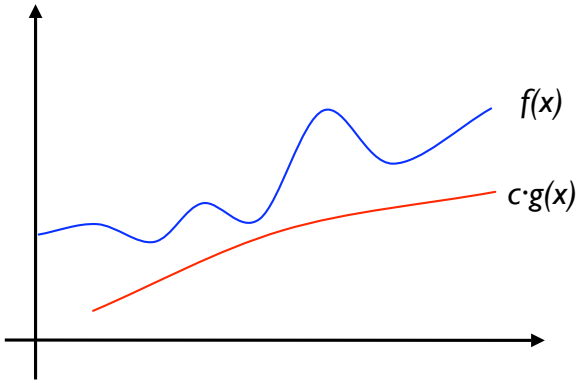
$$f \in O(g) \Leftrightarrow \exists c, k: \forall x > k: f(x) \leq c \cdot g(x)$$



Lower-Bound

- La funzione f cresce **almeno come** g

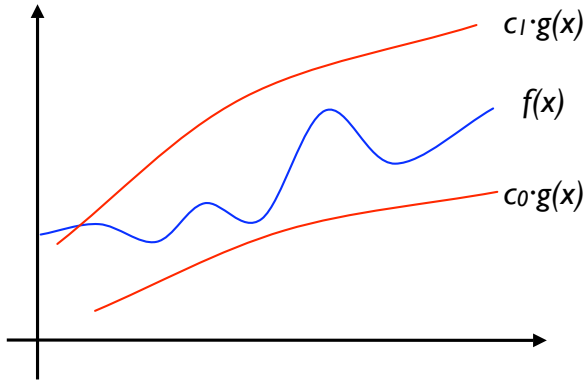
$$f \in \Omega(g) \Leftrightarrow \exists c, k: \forall x > k: c \cdot g(x) \leq f(x)$$



Tight-Bound

- La funzione f cresce **come** g $\quad \Theta(g) = O(g) \cap \Omega(g)$

$$f \in \Theta(g) \Leftrightarrow \exists c_0, c_1, k: \forall x > k: c_0 \cdot g(x) \leq f(x) \leq c_1 \cdot g(x)$$



Operazione dominante /1

- Per determinare l'andamento asintotico del programma non è necessario eseguirlo.
- Dall'analisi del codice del programma o della struttura dell'algoritmo possiamo determinare l'andamento asintotico del tempo di esecuzione senza doverlo compilare ed eseguire.
  - La determinazione delle costanti richiede la conoscenza esatta del compilatore e dell'esecutore del programma (al limite astratti).
- Un ruolo chiave in quest'analisi lo giocano le operazioni principali, il cuore dell'algoritmo, e.g., il confronto tra la chiave e l'elemento nella ricerca.

Operazione dominante /2

- Indichiamo con $N_I(n)$ il numero di volte che viene eseguita l'istruzione I nel caso peggiore su input di dimensione n.
- Se $t(n)$ è la funzione tempo nel caso peggiore
- I è un'**operazione dominante** se $t \in O(N_I)$.
- L' operazione o istruzione dominante non è unica e spesso può essere determinata facilmente analizzando la struttura del programma.

Operazione dominante /3

- In una sequenza di istruzioni elementari ogni istruzione è dominante.
- In un ciclo, che non contiene cicli interni, ogni istruzione non annidata all'interno di costrutti condizionali è dominante.
- In una sequenza di cicli annidati, ogni istruzione dominante del ciclo più interno è dominante per la sequenza di cicli annidati.

Operazione dominante /4

- Nel caso di funzioni ricorsive, ogni istruzione dominante per il corpo della funzione privo delle chiamate ricorsive è dominante.
- Noto il numero di volte che viene eseguita tale istruzione per ogni chiamata ricorsiva
- occorre determinare il numero di volte che viene richiamata la funzione.
  - Non sempre è facile eseguire questo calcolo in modo accurato, perché richiede la soluzione di equazioni di ricorrenza.
  - Quasi sempre si possono però trovare upper e lower-bound.

Dimensione dell'input

- Supponiamo che l'input di un programma sia un vettore di n elementi.
- Se c è la dimensione di un elemento, la dimensione dell'input è cn.
- Nella maggior parte dei casi il valore di c può essere ignorato, e.g., in molti casi
  - posta $h(n) = f(cn)$
  - $h \in O(g)$ quando $f(c) \in O(g)$