Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Costo computazionale

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Criterio della misura di costo uniforme

Se è soddisfatta l'ipotesi che ogni dato in input sia minore di un valore

$$k = 2^{\text{numero di bit della parola di memoria}}$$

ciascuna operazione elementare sui dati del problema verrà eseguita in un tempo costante.

In tal caso si parla di ***misura di costo uniforme***.

Tale criterio **non è sempre realistico** perché, se un dato del problema è **più grande di k**, esso deve comunque essere memorizzato, ed in tal caso si useranno più parole di memoria.

Di conseguenza, anche le operazioni elementari su di esso dovranno essere reiterate per tutte le parole di memoria che lo contengono, e quindi richiederanno un tempo che non è più costante.

Criterio della misura di costo logaritmico

Questo criterio, più realistico, risolve il problema sopra esposto assumendo che il **costo delle operazioni elementari** sia **funzione della dimensione degli operandi** (ossia dei dati).

Poiché il numero di bit necessari per memorizzare un valore ***n*** è proporzionale a ***log n***, si parla di ***misura di costo logaritmico***.

Essa però implica **rilevanti complicazioni** nei calcoli dell'efficienza di un algoritmo, per cui sia in letteratura che nella pratica si sceglie di usare la misura di costo uniforme, che si rivela adatta alla maggior parte dei problemi reali.

Esempio di applicazione dei due criteri (1)

Analizziamo informalmente un semplicissimo programma applicando entrambi i criteri al fine di evidenziarne le differenze.

Il programma consiste di un **ciclo, reiterato *n* volte, che calcola il valore 2^n**:

```
x ← 1;
for i = 1 to n do
  x ← x*2;
```

Con la misura di costo uniforme si vede facilmente che il tempo di esecuzione totale è ***proporzionale ad n***, poiché:

- si tratta di un **ciclo eseguito *n* volte**;
- ad ogni iterazione del ciclo si compiono **due operazioni**, ciascuna delle quali ha costo unitario:
  1. l'incremento del contatore
  2. il calcolo del nuovo valore di *x*.

Esempio di applicazione dei due criteri (2)

Con la misura di costo logaritmico le cose diventano subito più complicate, perché **sia l'incremento di *i* che il raddoppio di *x* non hanno più costo costante**.

In particolare, per ogni singola iterazione il costo complessivo è dato dalla somma dei seguenti due fattori:

1. *log i*, per l'incremento del contatore;
2. $\log x = \log 2^i = i$ per il calcolo del nuovo valore di *x*.

Il costo totale diviene dunque:

$$\sum_{i=1}^{n} (i + \log i)$$

Esempio di applicazione dei due criteri (3)

Ora, considerando che:

$$\sum_{i=1}^{n}(i + \log i) \geq \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

e che:

$$\sum_{i=1}^{n}(i + \log i) \leq \sum_{i=1}^{n} 2i = n(n+1)$$

si vede che il tempo di esecuzione totale è proporzionale ad n^2.

Valutazione del costo computazionale (1)

Vediamo ora come calcolare effettivamente il costo computazionale di un algoritmo, adottando il **criterio della misura di costo uniforme**.

Prima di procedere, facciamo due importanti considerazioni.

Innanzi tutto, osserviamo che è ragionevole pensare che il costo computazionale, inteso come funzione che rappresenta il tempo di esecuzione di un algoritmo, sia una **funzione monotona non decrescente** della **dimensione dell'input**.

Questa osservazione ci conduce a constatare che, prima di passare al calcolo del costo, bisogna **definire quale sia la dimensione dell'input**.
Trovare questo parametro è, di solito, abbastanza semplice:

- in un algoritmo di ordinamento esso sarà il numero di dati;
- in un algoritmo che lavora su una matrice sarà il numero di righe e di colonne;
- in un algoritmo che opera su alberi sarà il numero di nodi, ecc.

Valutazione del costo computazionale (2)

Tuttavia, vi sono casi in cui l'individuazione del parametro non è banale; in ogni caso, **è necessario stabilire quale sia la variabile (o le variabili) di riferimento** prima di accingersi a calcolare il costo.

In secondo luogo, vogliamo sottolineare che la notazione asintotica viene sfruttata pesantemente per il calcolo del costo computazionale degli algoritmi, quindi - in base alla definizione stessa – tale costo computazionale potrà essere ritenuto **valido solo asintoticamente**.

In effetti, esistono degli algoritmi che per dimensioni dell'input relativamente piccole hanno un certo comportamento, mentre per dimensioni maggiori un altro.

Per poter valutare il tempo computazionale di un algoritmo, esso deve essere **formulato in un modo che sia chiaro, sintetico e non ambiguo**.

Pseudocodice (1)

Si adotta il cosiddetto ***pseudocodice***, che è una sorta di linguaggio di programmazione “informale” nell’ambito del quale:

- si impiegano, come nei linguaggi di programmazione, i costrutti di controllo (***for, if then else, while***, ecc.);
- si può impiegare il linguaggio naturale per specificare alcune operazioni;
- si omettono dettagli quali ad esempio la gestione degli errori, al fine di esprimere solo l’essenza della soluzione.

Non esiste una notazione universalmente accettata per lo pseudocodice.

Pseudocodice (2)

In questo corso (come del resto nel libo di testo) useremo le seguenti convenzioni nella scrittura di pseudocodice:

- si utilizza ***l'indentazione*** per rappresentare i diversi livelli dei blocchi di istruzioni
- il simbolo ← indica un'**assegnazione**
- il simbolo **=** si utilizza per verificare che il contenuto di 2 variabili sia lo stesso
- il simbolo **≠** si utilizza per verificare che il contenuto di 2 variabili sia differente.

Costo delle istruzioni (1)

- le ***istruzioni elementari*** (operazioni aritmetiche, lettura del valore di una variabile, assegnazione di un valore a una variabile, valutazione di una condizione logica su un numero costante di operandi, stampa del valore di una variabile, ecc.) **hanno costo *Θ(1)***;

- l'istruzione

```
if (condizione)
   then istruzione1
   else istruzione2
```

ha costo pari a:

1. il costo di verifica della condizione (di solito **costante**)

2. più il **massimo** dei costi di **`istruzione1`** e **`istruzione2`**; (analisi di **caso pessimo**)

Costo delle istruzioni (2)

- le ***istruzioni iterative*** (o ***iterazioni***: ***cicli for, while e repeat***) hanno un costo pari alla **somma dei costi massimi di ciascuna delle iterazioni** (compreso il costo di verifica della condizione).
  - Se tutte le iterazioni hanno **lo stesso costo massimo**, allora il costo dell'iterazione è pari al **prodotto** del costo massimo di una singola iterazione per il numero di iterazioni.

- In entrambi i casi è comunque **necessario stimare il numero delle iterazioni**.

- Si osservi che la **condizione viene valutata una volta in più rispetto al numero delle iterazioni**, poiché l'ultima valutazione, che dà esito negativo, è quella che fa terminare l'iterazione.

Il costo dell'algoritmo nel suo complesso è pari alla **somma dei costi delle istruzioni che lo compongono**.

Dipendenza del costo dall'input

Un dato algoritmo potrebbe avere tempi di esecuzione (e quindi costo computazionale) **diversi a seconda dell'input** (della stessa taglia). In tal caso, per fare uno studio esauriente del suo tempo computazionale, bisogna valutare prima quali siano i cosiddetti ***casi migliore e peggiore***, cioè cercare di capire quale input sia particolarmente vantaggioso, ai fini del costo computazionale dell'algoritmo, e quale invece svantaggioso.
Esempio: vettore già ordinato in un algoritmo di ordinamento.

Per avere un'idea di quale sia il tempo di esecuzione atteso di un algoritmo, **a prescindere dall'input**, è chiaro che è necessario prendere in considerazione il **caso peggiore**, cioè la situazione che porta alla computazione più onerosa.

Nel contempo, però, vorremmo essere il più precisi possibile e quindi, **nel contesto del caso peggiore**, cerchiamo di calcolare il costo in termini di notazione asintotica Θ.

Laddove questo non sia possibile, essa dovrà essere approssimata per difetto (tramite la notazione Ω) e per eccesso (tramite la notazione O).

Esempio: calcolo del massimo (1)

Calcolo del massimo in un vettore disordinato contenente *n* valori.

Nota: ***Θ(1)*** indica un ***valore costante***.

```
Funzione Trova_Max (A: vettore)

1  max ← A[1]                    Θ(1)

2  for i = 2 to n do             (n-1) iterazioni più Θ(1)

3     if A[i] > max              Θ(1)

4        then max ← A[i]         Θ(1)

5  stampa max                    Θ(1)
```

Esempio: calcolo del massimo (2)

Il costo dell'istruzione 1 è $\Theta(1)$.

L'iterazione viene eseguita $(n - 1)$ **volte**, e ciascuna iterazione (costituita dalle istruzioni 2, 3 e 4) ha costo

$$\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$$

poiché l'incremento del contatore (istruzione 2), la valutazione della condizione (istruzione 3) e l'assegnazione (istruzione 4) sono **istruzioni elementari**.

Il costo dell'istruzione 5 è $\Theta(1)$.

Quindi, detto $T(n)$ il costo computazionale di questo algoritmo, esso vale:

$$T(n) = \Theta(1) + [(n - 1)\ \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

Esempio: calcolo del massimo (3)

Attenzione

La correttezza di questa relazione:

$$T(n) = \Theta(1) + [(n - 1)\Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

Non deriva dal mettere a fattor comune i singoli $\Theta(1)$, il che **non è corretto** perché essendo relativi a istruzioni diverse possono nascondere costanti diverse, ma dal fatto che:

$$(n - 1)\Theta(1) = \Theta(n)$$

Infatti:

$$c'n \leq (n - 1)\, c \leq c''n$$

per $\mathbf{n_0 = 2,\ c' \leq c/2,\ c'' \geq c}$.

Esempio: somma dei primi n interi (1)

Calcolo della somma di tutti i numeri da 1 a *n*.

```
Funzione Calcola_Somma(n: intero)
1  somma ← 0                    Θ(1)
2  for i = 1 to n do            (n) iterazioni più Θ(1)
3     aggiungi i a somma        Θ(1)
4  stampa somma                 Θ(1)
```

Esempio: somma dei primi n interi (2)

Il costo dell'istruzione 1 è $\boldsymbol{\Theta(1)}$.

L'iterazione viene eseguita n volte, e ciascuna iterazione (costituita dalle istruzioni 2 e 3) ha costo $\boldsymbol{\Theta(1) + \Theta(1) = \Theta(1)}$.

Il costo dell'istruzione 4 è $\boldsymbol{\Theta(1)}$.

Quindi, detto $T(n)$ il costo computazionale di questo algoritmo, esso vale:

$$T(n) = \Theta(1) + [n\ \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

Attenzione alla dimensione dell'input!

Benché il costo di questo programma sia $\theta(n)$, si tratta di un algoritmo di **costo esponenziale** in quanto l'input è il numero n che può essere rappresentato con una sequenza di $\log_2 n$ simboli.

Differente sarebbe il programma analogo che somma n valori memorizzati in un vettore, nel qual caso la dimensione dell'input è proporzionale ad n.

Esempio: somma dei primi n interi (3)

Osserviamo, tuttavia, che lo stesso problema può essere risolto in modo ben più efficiente come segue:

```
Funzione Calcola_Somma(n: intero)
1    somma ← n (n+1)/2          Θ(1)
2    stampa somma               Θ(1)
```

Il costo della funzione è, ovviamente, *Θ(1),* costo decisamente migliore rispetto al *Θ(n)* precedente.

Questo è vero, ripetiamo, sotto l'ipotesi di **costo uniforme**.

Esempio: valutazione di un polinomio (1)

Valutazione del polinomio $\sum_{i=0}^{n} a_i x^i$ nel punto x = c.

```
Funzione Calcola_Polinomio(n:intero, A:vettore, c:reale)
sum ← A[0]                          Θ(1)
for i = 1 to n do                   n iterazioni più Θ(1)
   potenza ← 1                      Θ(1)
   for j = 1 to i do                i iterazioni più Θ(1)
      Potenza ← c*potenza           Θ(1)
       sum ← sum+A[i]*potenza       Θ(1)
stampa somma                        Θ(1)
```

Esempio: valutazione di un polinomio (2)

Il costo computazionale dell'istruzione 1 è ***Θ(1)***.

La prima iterazione (istruzione 2) viene eseguita ***n* volte**; ciascuna iterazione contiene le 4 istruzioni 3-6 il cui costo è globalmente ***Θ(i)*** poiché **vi è un ciclo eseguito *i* volte** con, all'interno, operazioni costanti.

Il costo computazionale dell'istruzione 7 è, infine, ***Θ(1)***.

Quindi, detto *T(n)* il costo computazionale di questo algoritmo, esso vale:

$$T(n) = \Theta(1) + \left[\sum_{i=1}^{n} (\Theta(1) + \Theta(i)) + \Theta(1) \right] + \Theta(1) = \Theta(1) + \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Istruzioni	1	2,3,6,fin.4	4,5	fin.2	7

Operazioni Elementari Reloaded

Ancora una volta, osservo che un algoritmo è più chiaro usando **operazioni complesse**: nel nostro esempio, l'elevamento a potenza può essere considerato (**in fase di progetto**) un'operazione elementare.

In fase di **analisi di complessità computazionale** occorre ricordare che **non si tratta di un'operazione elementare** che si esegue in $\Theta(1)$! (in questo caso *potenza* è $\Theta(i)$).

```
def calcolaPolinomio(array p, int n, float c):

    v ← 0;
    for i ← 1 to n do
        v ← v + p[i] * potenza(c, i);

    return v,
```

Esempio: valutazione di un polinomio (3)

Osserviamo tuttavia che, solo riscrivendo in modo più oculato lo pseudocodice dello stesso algoritmo, il medesimo problema può essere risolto in modo più efficiente come segue:

```
Funzione Calcola_Polinomio(n:intero, A:vettore, c:reale)
 1  somma ← A[0]                          Θ(1)
 2  potenza ← 1                           Θ(1)
 3  for i = 1 to n do                     n iterazioni più Θ(1)
 4     potenza ← c*potenza                Θ(1)
 5     somma ← somma + A[i]*potenza       Θ(1)
 6  stampa somma                          Θ(1)
```

Esempio: valutazione di un polinomio (4)

In questa versione della funzione abbiamo semplicemente evitato di ricalcolare più volte le potenze di c, sfruttando quelle calcolate precedentemente.

Il costo computazionale di questa funzione è, ovviamente, **$\Theta(n)$**, costo decisamente migliore rispetto al **$\Theta(n^2)$** precedente.

Costi computazionali e tempi di esecuzione (1)

Vogliamo ora mostrare come variano i tempi di esecuzione di un algoritmo in funzione del suo costo computazionale.

Ipotizziamo di disporre di un sistema di calcolo in grado di effettuare una operazione elementare in un nanosecondo (**10^9 operazioni al secondo**), e supponiamo che la dimensione del problema sia **$n = 10^6$** (un milione):

- costo ***O(log n)*** - tempo di esecuzione: ***6 miliardesimi di secondo***;
- costo ***O(n)*** - tempo di esecuzione: ***1 millesimo di secondo***;
- costo ***O(n log n)*** - tempo di esecuzione: ***20 millesimi di secondo***;
- costo ***$O(n^2)$*** - tempo di esecuzione: ***1000 secondi = 16 minuti e 40 secondi***.

Costi computazionali e tempi di esecuzione (2)

C'è un'altra situazione interessante da considerare: che succede se il costo computazionale cresce esponenzialmente, ad esempio quando è $O(2^n)$?

E' abbastanza ovvio che i tempi di esecuzione diventano rapidamente proibitivi: un tale tipo di problema su un input di dimensione $n = 100$ richiede per la sua soluzione mediante il sistema di calcolo di cui sopra ben $1{,}26*10^{21}$ secondi, cioè circa $3*10^{13}$ anni.

Si potrebbe ipotizzare che l'avanzamento tecnologico, magari formidabile, possa prima o poi rendere abbordabile un tale problema, ma purtroppo non è così.

Costi computazionali e tempi di esecuzione (3)

Infatti, poniamoci la seguente domanda: supponendo di avere un calcolatore estremamente potente che riesce a risolvere un problema di dimensione $n = 1000$, avente costo computazionale $O(2^n)$, **in un determinato tempo *T***, quale dimensione ***n' = n + x*** del problema riusciremmo a risolvere nello stesso tempo utilizzando un calcolatore **mille volte più veloce**?

Possiamo scrivere la seguente uguaglianza:

$$T = \frac{2^{1000}\ operazioni}{10^k\ operazioni\ al\ secondo} = \frac{2^{1000+x}\ operazioni}{10^{k+3}\ operazioni\ al\ secondo}$$

Si ha quindi:

$$\frac{2^{1000+x}}{2^{1000}} = 2^x = \frac{10^{k+3}}{10^k} = 10^3 = 1000$$

Ossia ***x = log 1000 ≈ 10***

Costi computazionali e tempi di esecuzione (4)

Dunque, con un calcolatore mille volte più veloce riusciremmo solo a risolvere, nello stesso tempo, **un problema di dimensione 1010 anziché di dimensione 1000**!

In effetti esiste un'importantissima branca della teoria della complessità che si occupa proprio di caratterizzare i cosiddetti problemi ***intrattabili***, ossia quei problemi il cui costo computazionale è tale per cui essi non sono né saranno mai risolubili per dimensioni realistiche dell'input.

Noi ci concentreremo su problemi decisamente più semplici e perseguiremo l'***efficienza***: non ci limiteremo a risolvere un problema, ma cercheremo di risolverlo in modo efficiente, progettando cioè un algoritmo che, tra tutti quelli che risolvono quel problema, **abbia costo computazionale minore possibile**.

Esercizi (1)

Calcolare il costo del seguente algoritmo scritto in pseudocodice, distinguendo tra caso migliore e caso peggiore se necessario:

```
Funzione Insertion_Sort(A[1..n])
1    for j = 2 to n do
2         x ← A[j]
3         i ← j - 1
4         while ((i > 0)and(A[i] > x))
5              A[i+1] ← A[i]
6              i ← i - 1
7         A[i+1] ← x
```

Riscrivere lo stesso algoritmo considerando una funzione insert(A, n, x) che inserisce x al posto giusto in vettore ordinato nelle posizioni 1..n

Esercizi (2)

Calcolare il costo del seguente algoritmo scritto in pseudocodice, distinguendo tra caso migliore e caso peggiore se necessario:

```
Funzione Selection_Sort(A[1..n])
1    for i = 1 to n - 1 do
2         m ← i
3         for j = i + 1 to n do
4              if (A[j] < A[m])
5                   m ← j
6         Scambia A[m] e A[i]
```

Riscrivere lo stesso algoritmo considerando una funzione minimo(A, n) che restituisce il minimo di in vettore di n elementi.

Esercizi (3)

Calcolare il costo del seguente algoritmo scritto in pseudocodice, distinguendo tra caso migliore e caso peggiore se necessario:

```
Funzione Bubble_Sort(A[1..n])
1    for i = 1 to n do
2         for j = n downto i + 1 do
3              if (A[j] < A[j - 1])
4                   Scambia A[j] e A[j - 1]
```

Questo algoritmo (a differenza di Selection_Sort e meglio di Insertion_Sort) potrebbe trarre vantaggio dal fatto che il vettore sia già o quasi ordinato (migliorando quindi il caso ottimo e medio ma non quello pessimo).

Come si può modificare questo algoritmo?