Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2023/24

Costo computazionale

Tiziana Calamoneri

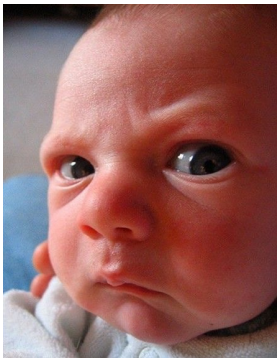
SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Valutazione del costo computazionale (1)

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

Nota: 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.



Valutazione del costo computazionale (2)

Trovare questo parametro è, di solito, abbastanza semplice:

- in un algoritmo di ordinamento esso sarà il numero di dati da ordinare;
- in un algoritmo che lavora su una matrice sarà il numero di righe e di colonne (quindi, 2 parametri);
- in un algoritmo che opera su alberi sarà il numero di nodi;
- ecc.

Valutazione del costo computazionale (3)

In altri casi, invece, 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.

Valutazione del costo computazionale (4)

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.

Valutazione del costo computazionale (5)

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

Pseudocodice (1)

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

Si adotta il cosiddetto pseudocodice, una sorta di linguaggio di programmazione "informale":

- si usano, come nei linguaggi di programmazione, i costrutti di controllo (for, if then else, while, ecc.);
- si può usare il linguaggio naturale per specificare alcune operazioni;
- si omettono dettagli (ad es. la gestione degli errori), per esprimere solo l'essenza della soluzione.

Pseudocodice (2)

Non esiste una notazione universalmente accettata per lo pseudocodice.

In questo corso useremo spesso i costrutti del Python perché sono particolarmente intuitivi ma, a volte, potremo usare altre convenzioni, pure intuitive, ad esempio il simbolo ≠ per verificare che il contenuto di 2 variabili sia differente...

Costo delle istruzioni (1)

- le istruzioni elementari, tra cui:
  - 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

  hanno costo $\Theta(1)$;

Costo delle istruzioni (2)

- l'istruzione

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

  ha costo pari a:
  1. il costo di verifica della condizione (di solito costante, ma NON sempre)
  2. più il max tra i costi di **`istruzione1`** e **`istruzione2`**;

Costo delle istruzioni (3)

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

 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 (ma se il suo costo è costante, possiamo ignorarlo…).

Valutazione del costo computazionale (5)

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

Un dato algoritmo potrebbe avere tempi di esecuzione (e quindi costo computazionale) diversi a seconda dell'input

casi migliore e peggiore (FISSATA UNA DIMENSIONE, input particolarmente vantaggioso e, rispettivamente, svantaggioso, ai fini del costo computazionale dell'algoritmo)

Dipendenza del costo dall'input

Per avere un'idea di quale sia il tempo di esecuzione di un algoritmo, a prescindere dall'input: **caso peggiore** (cioè la situazione che porta alla computazione più onerosa).

Ma vorremmo anche essere il più precisi possibile e quindi calcoliamo il costo nel caso peggiore, in termini di notazione asintotica Θ.

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

Esempio: calcolo del massimo

Calcolo del max in un array disordinato con n valori.

Θ(1) indica un valore costante.

```
def Trova_Max (A):
n=len(A)                    Θ(1)
max=A[0]                    Θ(1)
for i in range (1,n):       (n-1) iteraz. + Θ(1)
     if A[i] > max:            Θ(1)
        max=A[i]               Θ(1)
return max
```

Detto $T(n)$ il costo computazionale di questo algoritmo:

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

Esempio: somma dei primi n interi (1)

Calcolo della somma dei primi n interi.

```
def Calcola_Somma_1(n):
somma = 0                  Θ(1)

for i in range (1,n+1):    n iterazioni + Θ(1)

    somma += i                 Θ(1)

return somma               Θ(1)
```

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

qui potremmo disquisire su quale sia la dimensione dell'input...

Esempio: somma dei primi n interi (2)

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

```
def Calcola_Somma_2(n):
  somma = n*(n+1)/2      Θ(1)
  return somma           Θ(1)
```

Il costo della funzione è, ovviamente, $\Theta(1)$, costo decisamente migliore rispetto al $\Theta(n)$ precedente.

Esempio: valutazione di un polinomio (1)

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

```
def Calcola_Polinomio_1(A, c):
 somma=A[0]                          Θ(1)
 for i in range(1,len(a)):           n iterazioni + Θ(1)
     potenza = 1                       Θ(1)
     for j in range(i):                i iterazioni + Θ(1)
         potenza = c*potenza              Θ(1)
     somma = somma+A[i]*potenza        Θ(1)
 return somma                        Θ(1)
```

$T(n)= \Theta(1)+\Sigma_{i=1..n} \left(\Theta(1)+ \Theta(i)+\Theta(1)\right)+\Theta(1)=...$

Esempio: valutazione di un polinomio (2)

$T(n)= \Theta(1)+\Sigma_{i=1..n} \left(\Theta(1)+ \Theta(i)+ \Theta(1)\right)+ \Theta(1)=$

$= \Theta(1)+\Sigma_{i=1..n} \left(\Theta(1)+ \Theta(i)\right)=$

$= \Theta(1)+\Sigma_{i=1..n} \Theta(1)+ \Sigma_{i=1..n} \Theta(i) =$

$= \Theta(1)+\Theta(n)+\Theta(n^2)$

avendo usato che:

$$\Sigma_{i=1..n} \Theta(i) = \Theta\left(\Sigma_{i=1..n} i\right) = \Theta(n\ (n+1)/2)= \Theta(n^2)$$

Esempio: valutazione di un polinomio (3)

Riscrivendo in modo più oculato lo pseudocodice, il medesimo problema può essere risolto in modo più efficiente come segue:

```
def Calcola_Polinomio(A,c):
  somma = A[0]                          Θ(1)
  potenza = 1                           Θ(1)
  for i in range(1,len(A)):             n iterazioni + Θ(1)
       potenza = c*potenza                 Θ(1)
       somma=somma+A[i]*potenza            Θ(1)
  return somma                          Θ(1)
```

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)

Cerchiamo di capire la relazione tra i tempi di esecuzione di un algoritmo ed il suo costo computazionale.

Ipotizziamo di disporre di un sistema di calcolo in grado di effettuare una operazione elementare in un nanosecondo (10^9 operaz./sec.), e supponiamo che la dim. del problema sia $n = 10^6$ (un milione):

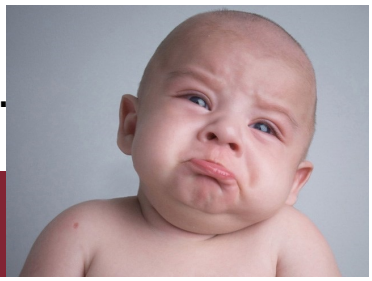
- costo $O(n)$: tempo di esec.: 1 millesimo di sec.;
- costo $O(n \log n)$: tempo di esec.: 20 millesimi di sec.;
- costo $O(n^2)$: tempo di esec.: 1000 sec. = 16 min. e 40 sec.

Costi computazionali e tempi di esecuzione (2)

Che succede se il costo computazionale cresce esponenzialmente, cioè ad esempio è in $O(2^n)$?

Anche solo su un input di dimensione n = 100, l'eventuale algoritmo richiederebbe per la sua soluzione ben $1{,}26 \cdot 10^{21}$ sec., cioè circa $3 \cdot 10^{13}$ anni.

Un algoritmo con costo esponenziale serve a poco, infatti l'avanzamento tecnologico, seppur formidabile, non è in grado di rendere abbordabile un tale problema.



Costi computazionali e tempi di esecuzione (3)

Infatti, supponiamo di avere un calcolatore 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?

$$T = \frac{2^{1000}\ operaz.}{10^k\ operaz.\ al\ sec.} = \frac{2^{1000+x}\ operaz.}{10^{k+3}\ operaz.al\ sec.}$$

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 1000!

Cioè: un algoritmo con costo esponenziale serve a poco oggi e servirà a poco domani.

Costi computazionali e tempi di esecuzione (5)

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.

Esercizi per casa

Esercizi per casa (1)

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

```
def Insertion_Sort(A):
    for j in range (1,len(A)):
        x = A[j]
        i = j - 1
        while (i>=0)and(A[i]>x):
            A[i+1] = A[i]
            i=i-1
        A[i+1]=x
```

Esercizi per casa (2)

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

```
def Selection_Sort(A):
    for i in range(len(A)-1):
        m=i
        for j in range(i+1,len(A)):
            if A[j] < A[m]:
                m = j
        A[m],A[i]=A[i],A[m]
```


Esercizi per casa (3)

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

```
def Bubble_Sort(A)
    for i in range(len(A)-1):
        for j in range(len[A]-i-1):
            if (A[j] > A[j+1]):
                A[j],A[j+1]=A[j+1],A[j]
```