Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Il costo di un algoritmo

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Costo computazionale

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

- E' 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.

- Quindi, 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 da ordinare, in un algoritmo di ricerca sarà il numero di elementi in cui ricercare, 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..

- 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.

Pseudocodice

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***, che è 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.

Non esiste una notazione universalmente accettata per lo pseudocodice.

In questo corso (come nel libo di testo) useremo spesso i costrutti del Python ma, a volte, potremo usare altre convenzioni intuitive, ad esempio il simbolo ≠ per verificare che il contenuto di 2 variabili sia differente

Regole generali per valutare la complessità computazionale di un algoritmo:

Costo delle istruzioni

• 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 $\Theta(1)$;

• l'istruzione

```
if condizione:
    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;

Regole generali per valutare la complessità computazionale di un algoritmo:

Costo delle istruzioni

•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. 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.

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 di una lista contenente n valori.

```
def TrovaMax(lista):
    mass=lista[0]                    Θ(1)
    for i in range(1,len(lista)):    n iterazioni
        if lista[i]>mass:            Θ(1)
            mass=lista[i]            Θ(1)
    print(mass)                      Θ(1)
```

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

Esempio. Calcolo della somma dei primi n interi.

```
Funzione CalcolaSomma(n):
    somma=0                          Θ(1)
    for i in range (1,n+1):          n iterazioni
        somma+=i                     Θ(1)
    print(somma)                     Θ(1)
```

$$T(n) = \Theta(1) + n \cdot \Theta(1) + \Theta(1) = \Theta(n)$$

Esempio. Calcolo della somma dei primi n interi.

Nota che $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

```
def CalcolaSomma1(n):
    somma=n*(n+1)//2     Θ(1)
    print(somma)         Θ(1)
```

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

ATTENZIONE: Non confondere la complessità dell'algoritmo con quella del problema.

ATTENZIONE: perseguire l'efficienza.

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

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

```
def potenza(c,i):
    r=1                              Θ(1)
    for j in range(i):               i iterazioni
        r=c*r                        Θ(1)
    return r                         Θ(1)
```

```
>>>CalcolaPolinomio([1,-3,0,2],2)
11
```

$2x^3 - 3x + 1$ **calcolato nel punto** $c = 2$ **vale** 11

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

Dove abbiamo usato che $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$

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

```
def valuta(A,c):
    potenza=1
    p=0
    for i in range(len(A)):
        p+=A[i]*potenza
        potenza=potenza*c
    return p
```

$\Theta(1)$
$\Theta(1)$
n **iterazioni**
$\Theta(1)$
$\Theta(1)$
$\Theta(1)$

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

Studiamo ora 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 $n = 10^6$:

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

Cosa succede se il costo cresce esponenzialmente, ad esempio quando è $O(2^n)$?

Se anche $n = 100$ con il precedente sistema di calcolo, il tempo computazionale è di $1{,}26 * 10^{21}$ secondi, cioè circa $3 * 10^{13}$ anni!

ATTENZIONE: un algoritmo con complessità esponenziale serve a poco.

Si potrebbe pensare che l'avanzamento tecnologico possa prima o poi rendere accettabili tali algoritmi, ma purtroppo non è così:

•supponiamo di avere un calcolatore estremamente potente che riesce a risolvere un problema di dimensione *n = 1000*, avente complessità $O(2^n)$, 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?

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

deve aversi $\frac{10^{k+3}}{10^k} = \frac{2^{1000+x}}{2^{1000}}$ **da cui ricaviamo** $2^x = 1000$ **vale a dire** $x = \log_2 1000$

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

ATTENZIONE: un algoritmo con complessità esponenziale serve a poco oggi e servirà a poco domani.

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 esponenziale 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**.

Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio 1

Calcolare il costo del seguente algoritmo, 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
```

Esercizio 2

Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

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

Esercizio 3

Calcolare il costo del seguente algoritmo, distinguendo tra caso migliore e caso peggiore se necessario.

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