1 Costi

In generale, la risoluzione di un problema su un calcolatore potrà essere effettuata e codificata in modi diversi, ossia con istruzioni diverse le quali in conclusione portano allo stesso risultato finale. Tuttavia, prima di codificare un algoritmo oppure un programma, bisogna tenere in considerazione dei fattori importanti che ne condizionano l'efficacia. Essi sono

- ⋆ numero di operazioni richieste – costo computazionale
- ⋆ il tempo di CPU
- ⋆ l'*elapsed time*

1.1 Costo computazionale

Il **costo computazionale** di un algoritmo è il *numero di operazioni aritmetiche* che esso richiede per la sua esecuzione. In genere, sarà necessario solamente quantificarne la grandezza in funzione di un parametro n legato alla dimensione del problema che si sta risolvendo. Questa grandezza determinerà la **complessità** dell'algoritmo.

Più precisamente, diremo che un algoritmo ha una *complessità costante* se richiede un numero di operazioni indipendente da n, cioè se richiede $\mathcal{O}(1)$ operazioni; *lineare* se richiede $\mathcal{O}(n)$ operazioni e, più in generale, *polinomiale* se richiede $\mathcal{O}(n^k)$ operazioni con k intero positivo; *esponenziale* se richiede $\mathcal{O}(c^n)$ operazioni; *fattoriale* se richiede $\mathcal{O}(n!)$ operazioni.

Esempio 1.1 (Prodotto matrice-vettore)**.** *Consideriamo l'algoritmo standard per eseguire il prodotto matrice-vettore* $A\mathbf{v}$ *tra una matrice quadrata* $A = (a_{ij})$, $i, j = 1, \dots, n$ *di dimensione* n *ed un vettore* $\mathbf{v} \in \mathbb{R}^n$ *di componenti* $\mathbf{v} = [v_1, \dots, v_n]^T$*. Vogliamo quantificare il costo computazionale dell'algoritmo in funzione della dimensione* n*. Per calcolare la componente* j*-esima del vettore prodotto*

$$(A\mathbf{v})_j = a_{j1}v_1 + \cdots + a_{jn}v_n, \tag{1}$$

dobbiamo eserguire n *prodotti e* $n-1$ *somme, ovvero* $2n-1$ *operazioni. In tutto dobbiamo calcolare* n *componenti, e dovremo quindi eseguire* $n(2n-1) = 2n^2 - n$ *operazioni aritmetiche, valore che per* n *grandi, si comporta come una costante per* n^2*. In conclusione questo algoritmo richiede* $\mathcal{O}(n^2)$ *operazioni ed ha pertanto complessità quadratica rispetto al parametro* n*.*

Esempio 1.2 (Prodotto matrice-matrice)**.** *Consideriamo l'algoritmo standard per eseguire il prodotto matrice-matrice* AB *tra due matrici quadrate* $A = (a_{ij})$*, e* $B = (a_{ij})$ $i, j = 1, \dots, n$ *di dimensione* n*. Questo algoritmo ripete* n *volte dei prodotti matrice-vettore* (1) *con lo stesso procedimento dell'Esempio 1.1, pertanto richiede* $\mathcal{O}(n^3)$ *operazioni. Esiste tuttavia un algoritmo, detto algoritmo di Strassen, che ne richiede "solo"* $\mathcal{O}(n^{log_2 7})$*, ed un altro, dovuto a Winograd e Coppersmith, che richiede* $\mathcal{O}(n^{2.376})$ *operazioni.*

Esempio 1.3 (Determinante di una matrice quadrata)**.** *L'algoritmo corrispondente alla formula ricorsiva di Laplace per il calcolo del determinante di una matrice quadrata di dimensione* n *ha una complessità fattoriale rispetto a* n*, ovvero richiede* $\mathcal{O}(n!)$ *operazioni.*

Algoritmi con complessità elevate, come ad esempio quelle fattoriali, non possono essere eseguiti neppure sui calcolatori più avanzati oggi disponibili, se non per n piccoli.

Ponendo l'attenzione sui calcolatori, uno degli indicatori della loro velocità è il massimo numero di operazioni *floating-point* che l'elaboratore stesso esegue in un secondo (**flops**). In particolare sono utilizzate le seguenti sigle:

Mega-flops	10^6 flops
Giga-flops	10^9 flops
Tera-flops	10^{12} flops
Peta-flops	10^{15} flops

Il calcolatore più potente che esista (secondo la *top500 supercomputer list* di Novembre 2019: https://www.top500.org/lists/2019/11/) può effettuare circa 148.6 Peta-flops ed è il Summit del Oak Ridge National Laboratory, negli Stai Uniti (cf. Fig. 1).

Figure 1: Il calcolatore più potente che esista: *Summit* del Oak Ridge National Laboratory, negli Stai Uniti.

Ad esempio, se $n = 24$, su un elaboratore in grado di eseguire 1 Peta-flops (cioè 10^{15} operazioni floating-point al secondo) servirebbero circa 20 anni (!) per terminare il calcolo. Come rimedio, esiste un algoritmo di tipo ricorsivo che consente di ridurre il calcolo del determinante a quello del prodotto di matrici, dando così luogo ad una complessità di $\mathcal{O}(n^{log_2 7})$ operazioni se si ricorre all'algoritmo di Strassen.

In conclusione, il numero di operazioni richiesto da un algoritmo è dunque un parametro molto importante da tenere in considerazione nella scrittura e analisi dell'algoritmo stesso.

1.2 CPU ed elapsed time

Un'altra misura importante delle prestazioni di un programma (o di una sequenza di istruzioni) è il cosiddetto **tempo di CPU** (*central processing unit*) ovvero il tempo impiegato dall'unità centrale del calcolatore per eseguire quel determinato programma.

Il tempo di CPU non tiene conto del tempo utilizzato da altri processi (siano essi di sistema o lanciati dall'utente) che sono in esecuzione in contemporanea e che sono gestiti

in multitasking dal processore stesso, ed è quindi diverso dal tempo che intercorre tra il momento in cui un programma è stato mandato in esecuzione ed il suo completamento. Quest'ultimo è noto (in inglese) come **elapsed time**.

I comandi MATLAB (Octave) per misurare (in secondi) il tempo di CPU e l'elapsed time sono rispettivamente `cputime`, `etime`, `tic` e `toc` . Più precisamente,

- ⋆ `cputime` restituisce il tempo totale di CPU (in secondi) usato da MATLAB da quando è stato inizializzato,
- ⋆ `etime` restituisce il tempo (in secondi) che si trascorso tra due vettori dati,
- ⋆ `tic` e `toc` misurano il tempo (in secondi) trascorso tra i momenti in cui `tic` e `toc` sono stati invocati.

Ad esempio, le istruzioni

```
t=cputime; operazioni; cputime-t
```

misurano il tempo CPU necessario a fare girare operazioni, mentre le istruzioni

```
t = clock;  operazioni;  etime(clock,t)
```

restituiscono l'elapsed time per operazioni. Il comando `etime(T1,T0)` restituisce il tempo in secondi che è trascorso tra i vettori $T1$ and $T0$. È importante osservare che i due vettori devono essere lunghi di sei elementi, nel formato restituito dal comando `clock`

```
[year month day hour minute seconds]
```

il quale restituisce il tempo e la data del computer su cui stiamo lavorando, sotto forma di un vettore di sei elementi con rappresentazioni decimali. Il comando `fix(clock)` restituisce lo stesso risultato in rappresentazione intera. Il comando `date` restituisce la data corrente sotto forma di una stringa.

Tuttavia, più preferibili per misurare l'elapsed time rispetto a `etime`, sono i comandi `tic` e `toc`, utilizzati

```
tic;  operazioni;  toc
```

i quali possono essere più affidabili per misurare il tempo richiesto da operazioni rispetto a `etime` e `clock`, in quanto quest'ultimi dipendono dal tempo del sistema.

Esempio 1.4. *Le seguenti istruzioni misurano il tempo di CPU per calcolare il prodotto fra una matrice quadrata ed un vettore di dimensione n, entrambi costruiti con elementi casuali con l'utilizzo del comando* `rand`*, facendo uso del comando* `cputime`*.*

```
t=cputime;
n=1000;
A=rand(n,n);
v=rand(n,1);
b=A*v;
t=cputime-t
```

Esempio 1.5. *Le seguenti istruzioni misurano l'elapsed time per calcolare il prodotto fra una matrice quadrata ed un vettore di dimensione n utilizzando i comandi* `clock` *ed* `etime`*.*

```
t=clock;
n=1000;
A=rand(n,n);
v=rand(n,1);
b=A*v;
e=etime(clock,t)
```

Esempio 1.6. *Le seguenti istruzioni misurano l'elapsed time per calcolare il prodotto fra una matrice quadrata ed un vettore di dimensione n utilizzando i comandi* `tic` *ed* `toc`*.*

```
tic;
n=1000;
A=rand(n,n);
v=rand(n,1);
b=A*v;
toc
```

I tempi che verranno visualizzati come output varieranno in base al modello del computer che viene utilizzato e in base al carico attuale del computer. In questo aspetto, è importante fare notare che ci può essere una variazione considerevole dell'output dopo successive chiamate dello stesso codice. Verificare per gli esempi precedenti.

2 Controllo del flusso

Gli **operatori relazionali** vengono utilizzati per confrontare variabili e array. Quando si confrontano due array, vengono confrontati i singoli elementi del primo con gli elementi corrispondenti del secondo. Quando si confrontano tra uno scalare ed un array, viene confrontato eseguito lo scalare con tutti gli elementi dell'array. Il risultato di una operazione relazionale può essere 1 (vero) oppure 0 (falso). MATLAB rappresenta il vero e il falso tramite gli interi 0 ed 1. Ricordiamo i principali operatori relazionali

operatori relazionali	significato
==	uguale
~=	diverso
<	minore
>	maggiore
<=	minore o uguale
>=	maggiore o uguale

Gli **operatori logici** utilizzati da MATLAB (Octave) sono i seguenti tre

- ~ `not`: `~a` restituisce un array che ha le stesse dimensioni di `a`, i cui elementi valgono 1 se quelli di `a` sono nulli, e valgono 0, altrimenti;
- & `and`: $a\&b$ restituisce un array delle stesse dimensioni di a e b, i cui elementi valgono 1 se i corrispondenti elementi di a e b sono entrambi diversi da 0; e valgono 0 se almeno uno tra i due elementi di a e b è uguale a 0;
- | `or`: $a|b$ restituisce un array delle stesse dimensioni di a e b, i cui elementi valgono 1 se almeno uno tra i due elementi corrispondenti di a e b è diverso da 0 e valgono 0 se entrambi sono uguali a 0.

Le **funzioni logiche** principali implementate in MATLAB sono:

- any: `any(x)` restituisce 1 se almeno un elemento di x è diverso da 0, altrimenti restituisce 0; data una matrice A, `any(A)` agisce per colonna;
- all: `all(x)` restituisce 1 se tutti elementi di x sono diversi da 0; altrimenti restituisce 0; `all(A)` agisce per colonna
- find: `find(x)` restituisce un vettore con gli indici degli elementi non nulli di x; `[i, j, w]=find(A)` restutisce un vettore riga con tre elementi: i contiene gli indici di riga degli elementi non nulli di A, j gli indici di colonna e w il valore degli elementi non nulli;
- xor: `xor(A,B)` restituisce un array delle stesse dimensioni di A e B, i cui elementi sono pari a 1 se soltanto uno dei corrispondenti elementi di A e B è diverso da 0; pari a 0 in tutti gli altri casi

Esempio 2.1. *Dati i seguenti array* `a=[0 0 -2 4 5]` *e* `b=[-1 3 0 10 0]`*, valutare con MATLAB e verificare a mano le seguenti istruzioni*

```
1. c = b<~a
2. c = a&b
3. c = ~(a&b)
4. c = a|b
5. c = a|~b
6. c = xor(a,b)
7. c = xor(~a+b,b)
```

Esercizio 2.1. *Data la matrice* `A=[1 -2 6; 9 -1 8; -3 -3 4]` *si utilizzino gli operatori logici per individuare gli elementi positivi di A, si costruisca una matrice B che contenga solamente gli elementi di A nonnegativi, e abbia entrate nulle in corrispondenza degli elementi di A negativi.*

Soluzione:

```
A=[1 -2 6;9 -1 8;-3 -3 4]
M = A>=0; %costruiamo una matrice ausiliaria
          %che individui le entrate positive di A
B= A.*M
```

2.1 Statement condizionali

Lo statement condizionale `if-elseif-else` ha la seguente forma generale:

```
if <condition 1>
   <statement 1.1>
   <statement 1.2>
    ...
elseif <condition 2>
   <statement 2.1>
   <statement 2.2>
      ...
...
else
   <statement n.1>
   <statement n.2>
    ...
end
```

Le varie condizioni dove `<condition 1>, <condition 2>, ...` rappresentano espressioni logiche che possono assumere i valori 0 o 1 (falso o vero).

Se `<condition k>` è falsa gli statement `<statement k.1>, <statement k.2> , ...` non vengono eseguiti ed il controllo passa oltre.

La costruzione permette l'esecuzione degli statement corrispondenti alla prima condizione che assume valore uguale a 1. Se tutte le condizioni `<condition 1>, <condition 2>, ...` fossero false sarebbero eseguiti gli statement `<statement n.1>, <statement n.2>, ....`

Si faccia riferimento agli Esercizi 2.4-2.5.

Le espressioni logiche che compaiono in una condizione possono essere ottenute tramite gli operatori logici elementari oppure combinando espressioni logiche elementari mediante gli operatori booleani `&, |, &&`, e ||. Più precisamente, l'istruzione

```
(n_iter<=n_max) & (err>tol)
```

calcola il valore logico di entrambe le espressioni nelle parentesi e poi su di esse agisce l'operatore *and*, mentre l'istruzione

```
(n_iter<=n_max) && (err>tol)
```

calcola il valore logico della prima parentesi e, solo se questo è 1 (cioè vero), calcola il valore della seconda. Si dirà che gli operatori && e || implementano la forma *short-circuiting* invece di quella *element-by-element* degli operatori & e |.

2.2 Cicli

In MATLAB (Octave) sono disponibili due tipi di ciclo

- ciclo `for` ripete le istruzioni presenti nel ciclo stesso per tutti i valori dell'indice contenuti in un certo vettore riga di indici;

```
for i=1:n
    istruzioni
end
```

- ciclo `while` viene eseguito intanto che una data espressione logica è vera, ed ha la seguente forma generale

```
while condizione logica
    istruzioni
end
```

A differenza del ciclo `for`, il ciclo `while` viene utilizzato quando non si conosce a priori il numero di passaggi da effettuare. Ad ogni passaggio, MATLAB verifica la condizione: se è vera entra nel ciclo, esegue le istruzioni fino all'end e poi torna nuovamente a verificare la condizione; e così via. Quando MATLAB verifica che la condizione è diventata falsa, esce automaticamente dal ciclo.

Esempio 2.2. *Il seguente script calcola il primo intero positivo n per cui n! (n fattoriale) è un numero di 10 cifre, utilizzando un ciclo* `while`.

```
n=1;
while prod(1:n)<1e10
    n=n+1;
end
n-1
```

Esercizio 2.2. *Scrivere uno script che calcoli il valore più grande di n affinchè la somma*

$$1^3 + 2^3 + \cdots + n^3 < 500. \tag{2}$$

Soluzione:

```
S=1; n=2;
while S+n^3<500
  S=S+n^3;n=n+1;
end
[n-1,S]
```

Facciamo notare che è molto importante che la variabile di ciclo (n in questo caso; in generale quella che compare all'interno della condizione) venga modificata durante l'esecuzione delle istruzioni ($n = n+1$ in questo caso). In caso contrario, se la condizione è **sempre** vera MATLAB non uscirà **mai** dal ciclo, generando un ciclo infinito.

Esercizio 2.3. *Si consideri la successione di Fibonacci $f_i = f_{i-1} + f_{i-2}$, $i \geq 3$ con $f_1 = 0$ e $f_2 = 1$. Si calcolino i primi otto elementi della successione in due modi diversi: prima utilizzando un ciclo* `for` *poi utilizzando un ciclo* `while`*.*

Soluzione:

```
f(1)=0; f(2)=1;
for i=3:8 %alternativamente for i=[3 4 5 6 7 8]
  f(i)=f(i-1)+f(i-2);
end
f
```

```
f(1)=0; f(2)=1; i=3;
while i <= 8
    f(i)=f(i-1)+f(i-2); i=i+1;
end
f
```

Esercizio 2.4. *Dato il seguente array*

$$a = [3.021 - 0.003 - 1.443 - 0.073\ 0.009 - 0.103 - 0.016], \tag{3}$$

si crei uno script che sostituisca tutti gli elementi di a per cui che $-0.2 \leq a_i \leq 0.2$ *con il valore* 0. *Si utilizzino le istruzioni condizionali* `if-else` *annidate con un ciclo* `for`.

Soluzione:

```
a=[3.021 -0.003 -1.443 -0.073 0.009 -0.103 -0.016];
y=[ ]; %inizializziamo y come un vettore vuoto
for i=1:length(a)
    if abs(a(i))>=0.2
            y=[y,a(i)];
    else
            y=[y,0];
    end
end
y
```

Esercizio 2.5. *Si consideri la successione di Fibonacci definita nell'Esercizio 2.3. Si crei un* M-file *di tipo* function *per definirla il maniera ricorsiva, facendo utilizzo dello statement condizionale* `if-elseif-else`.

Soluzione:

```
function y = fun(n)
    if n==1
        y=0;
    elseif n==2
        y=1;
    else
        y=fun(n-1)+fun(n-2);
    end
return
```

Ultima versione aggiornata: 29 Aprile 2020

Link alle lezioni precedenti:
Lezioni 3-4
Lezioni 1-2

Referenze

1. MATLAB® The language of technical computing: computation, visualization, programming, The MathWorks Inc
2. MATLAB: An introduction with applications, A. Gilat, Wiley
3. Scientific Computing with MATLAB and Octave, A. Quarteroni, , F. Saleri, P. Gervasio, Springer
4. MATLAB Guide1 D. J. Higham and N. J. Higham, SIAM
5. Numerical Computing with MATLAB, C. Moler, SIAM