# ANALISI DEGLI ALGORITMI



ALGORITMO: un algoritmo è una descrizione precisa e non ambigua di una sequenza di passi che possono essere eseguiti meccanicamente.

In informatica i programmi non sono altro che la traduzione in un LINGUAGGIO di PROGRAMMAZIONE di un algoritmo.

È importante conoscere l'EFFICIENZA di un programma, cioè in pratica:

-1) Tempo di esecuzione
-2) Memoria occupata dalle sue variabili

Ci sono due metodi per misurare l'efficienza di un programma:

1) benchmarking
2) Analisi

Il primo metodo consiste nel provare i programmi su alcuni esempi significativi. Il secondo, più interessante, consiste nello studio teorico dell'algoritmo in qui sue fasi.

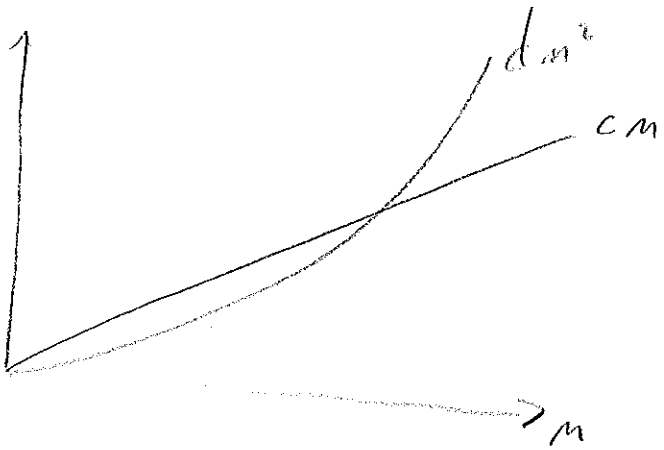
Per l'analisi degli algoritmi è innanzitutto importante conoscere la dimensione caratteristica del problema affrontato. Per esempio:

1) Soluzione sistemi lineari $\Rightarrow$ dimensione caratteristica $n$
2) Ordinamento di un vettore $\Rightarrow$ dimensione caratteristica $n$

In generale esprimeremo il tempo di esecuzione $T$ come funzione di $n$: $T = T(n)$

Tempo di esecuzione lineare: $T(n) = c \cdot n$
Tempo di esecuzione quadratica: $T(n) = d \cdot n^2$

Nei nostri modi non hanno grande importanza le costanti moltiplicative, ma più importanza la potenza
Se mettiamo in grafico:




Prima o poi $d n^2$ diventerà maggiore di $c n$ qualunque siano $c$ e $d$.

Il tempo di esecuzione di un programma si calcola considerando tutte le istruzioni che esso compie e attribuendo un certo tempo di esecuzione a ciascuna istruzione.
In generale non siamo interessati a conoscere esattamente il tempo di esecuzione di un programma, ma piuttosto ci avviene un'idea di come varia il tempo rispetto ad $n$.

NOTAZIONE O-GRANDE

Informalmente dire che $T(n) = O(f(n))$ significa dire che $T(n) = K \cdot f(n)$ con $K$ costante.
Più precisamente:
Si dice che $T(n) = O(f(n))$ se esistono $n_0$ e $K > 0$ tali che $T(n) \leq K f(n)$ $\forall n \geq n_0$

REGOLETTE:

1) Nei fattori costanti non contano: $T(n)$ è $O(dT(n))$ per qualsiasi costante $d$.

2) Nei termini di grado inferiore non contano:

$T(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ possiamo dire che $T(n)$ è $O(n^k)$

Tempi di esecuzione comuni:

| | |
|---|---|
| $O(1)$ | costante |
| $O(\log n)$ | logaritmico |
| $O(n)$ | lineare |
| $O(n \log n)$ | $n \log n$ |
| $O(n^2)$ | quadratico |
| $O(n^3)$ | cubico |
| $O(2^n)$ | esponenziale |

---

ESEMPI:

Prodotto scalare:

```
DDOT = 0.d0
DO i = 1, n
   DDOT = DDOT + V(i)*W(i)
ENDDO
```

TEMPO ASSEGNAZIONE $T_1$
TEMPO INIZIO CICLO $T_2$
TEMPO INCREMENTO CICLO $T_3$
TEMPO MOLTIPLICAZIONE $T_4$
TEMPO ADDIZIONE $T_5$

TEMPO TOTALE: $T(n) = T_1 + T_2 + n(T_3 + T_4 + T_5 + T_1)$

Posso chiamare $T_1 + T_2 = a$ e $(T_3 + T_4 + T_5 + T_1) = b$

$T(n) = a + b \cdot n$ e quindi $T(n) = O(n)$

PRODOTTO MATRICE VETTORE

1. DO i = 1, m
2. W(i) = 0.d0
3. DO j = 1, m
4. W(i) = W(i) + A(i,j) * V(j)
5. ENDDO
6. ENDDO

$T_1$ = Tempo assegnazione
$T_2$ = Tempo inizio ciclo
$T_3$ = Tempo incremento ciclo
$T_4$ = Tempo moltiplicazione
$T_5$ = Tempo addizione

$$T(m) = T_2 + m\left[T_1 + T_3 + T_2 + m\left(T_3 + T_4 + T_5 + T_1\right)\right] = a + b \cdot m + c \cdot m^2$$

$$\Rightarrow T(m) = \mathcal{O}(m^2)$$

ANALISI SEMPLIFICATA se ho dei cicli annidati comincio ad analizzare i cicli più interni!
Poi passo ad analizzare un ciclo quando il suo contenuto è determinato il suo tempo di esecuzione!!

Quando:

3. DO j = 1, m
4. W(i) = W(i) + A(i,j) * V(j)
5. ENDDO

la riga 4. ha tempo di esecuzione costante (non dipende da m) è quindi $O(1)$

Il ciclo da 3. a 5. ripete m volte la riga 4. e quindi ha tempo di esecuzione $m\,O(1)$ e cioè $O(m)$.

La riga 2. $O(1)$ e se la sommo al tempo di 3.-5. ottengo che 2.-5. è $O(1)+O(m)=O(m)$.

Infine noto che il ciclo 1.-6. ripete m-volte le istruzioni 2.-5. e quindi è $m\,O(m) = O(m^2)$ C.V.D.

# PRODOTTO MATRICE-MATRICE (QUADRATE)

```
1. DO i = 1, m
2.    DO j = 1, m
3.       C(i,j) = 0.d0
4.       DO k = 1, m
5.          C(i,j) = C(i,j) + A(i,k) * B(k,j)
6.       ENDDO
7.    ENDDO
8. ENDDO
```

Applico l'ordine semplificato:

1) Noto che le righe 2.-7. non sono altro che il prodotto matrice vettore e così sono $O(m^2)$.

2) Il prodotto matrice-matrice richiede m-volte il prodotto matrice-vettore cioè quindi in $O(m^2) = O(m^3)$

ESERCIZI:

Determinare il tempo di calcolo e complessità di:

- NORMA EUCLIDEA DI UN VETTORE
- NORMA ASSOLUTA E NORMA EUCLIDEA DI UNA MATRICE
- TRACCIA DI UNA MATRICE
- PRODOTTO MATRICE-VETTORE CON MATRICE RETTANGOLARE
- PRODOTTO MATRICE-MATRICE CON MAT. RETTANGOLARI

N.B.: per gli ultimi due esercizi la dimensione del problema può caratterizzarla da più di una costante (per es. m, n e m, n, l)

# Analisi del Metodo di Gauss

## Fase 1 - Triangolarizzazione

```
1. DO i = 1, m-1
2.    DO j = i+1, m
3.       PIVOT = -a(j,i)/a(i,i)
4.       DO k = i, m
5.          a(j,k) = a(j,k) + PIVOT * a(i,k)
6.       ENDDO
7.    ENDDO
8. ENDDO
```

Ciclo 4.6. è dimensione $O(m-i)$

Ciclo 2.7. è $(m-i-1)O(m-i) = O((m-i)^2)$

Ciclo totale è $O\left(\sum_{i=1}^{m-1}(m-i)^2\right)$

Vediamo la sommatoria:

$$\sum_{i=1}^{m-1}(m-i)^2 = \sum_{i=1}^{m-1}(m^2 - 2mi + i^2) =$$

$$\sum_{i=1}^{m-1} m^2 = (m-1)m^2$$

$$\sum_{i=1}^{m-1} 2mi = 2m\frac{(m-1)\cdot m}{2} = (m-1)m^2$$

$$\sum_{i=1}^{m-1} i^2 = \frac{(m-1)\cdot m\cdot(2(m-1)+1)}{6} = 2m^3 - 3m^2 + m$$

$$\Rightarrow \text{Ciclo totale è } O(m^3)$$

FASE 2 (SOSTITUZIONE INDIETRO)

1. Do $i = n, 1, -1$
2. SOM = 0.d0
3. DO $j = i+1, n$
4. SOM = SOM + $a(i,j) * x(j)$
5. ENDDO
6. $x(i) = (b(i) - SOM)/a(i,i)$
7. ENDDO

Il ciclo 3.-5. è dominante $O(n-i)$ e quindi la fase 2 risulta di complessità $O\left(\sum_{i=1}^{n}(n-i)\right)$

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

$\Rightarrow$ la sostituzione indietro è $O(n^2)$

$\Rightarrow$ IL METODO DI GAUSS È $O(n^3) + O(n^2) = O(n^3)$

---

METODO DI JACOBI

Finora abbiamo visto algoritmi con tempi di esecuzione predeterminati. Con Jacobi (e Seidel) non siamo in grado di determinare a priori la complessità perché non sappiamo quante iterazioni servono!! $\Rightarrow$ Possiamo determinare il costo x iterazione.

Scriviamo solo la parte interna

Per "i" che va da 1 a n
SOM1 = 0.d0
Per "j" che va da 1 a "i-1"
SOM1 = SOM1 + $a_{ij} x_j^{(k)}$
Fine Per
SOM2 = 0.d0
Per "j"

Costo per iterazione

1. Per "i" che va da 1 a m
2. SOM1 = 0.d0
3. Per "j" che va da 1 a "i-1"
4. SOM1 = SOM1 + $a_{ij} * x_k^{(s)}$
5. Fine Per
6. SOM2 = 0.d0
7. Per "j" che va da "i+1" a m
8. SOM2 = SOM2 + $a_{ij} * x_k^{(s)}$
9. Fine per
10. $x_{k+1}^{(i)} = \frac{1}{a_{ii}} (b_i - SOM1 - SOM2)$
11. $s_{k+1}^{(i)} = x_{k+1}^{(i)} - x_k^{(i)}$
12. Fine per
13. Scarto = $\|s_{k+1}\|$
14. Controllo convergenza

Analizzo

$T(2.) = O(1)$

$T(3.-5.) = O(i-1)$

$T(6.) = O(1)$

$T(7.-9.) = O(m-i)$

$T(10.) = O(1)$

$T(11.) = O(1)$

$T(2.-11.)$ è dato dalla somma di $T(3.-5.)$ e $T(7.-9.)$ che sono complementari e insieme hanno costo $O(m)$

$\Rightarrow T(2.-11.) = O(m)$

$T(1.-12.) = O(m^2)$

$T(13.) = O(m)$

$T(14.) = O(1)$

Ogni iterazione di Jacobi avrà costo $O(m^2) + O(m) + O(1) = O(m^2)$

# ORDINAMENTO PER SELEZIONE - SELECTION SORT (5) - BUBBLE SORT

Sia dato un array (di interi, di reali o anche di caratteri) disordinato.

Scopo: vogliamo ordinare i suoi termini dal più piccolo al più grande

Idea: parto dal primo elemento dell'array e lo confronto con tutti quelli che seguono e appena trovo un elemento più piccolo lo scambio di posizione (come nel calcolo del massimo e del minimo). Alla fine di questo passo so che l'elemento più piccolo è salito in cima all'array (come una bollicina). Ripeto lo stesso operazione a partire dal secondo termine del vettore e così via

ESEMPIO CON UN ARRAY INTERO

1) Ho bisogno di una SUBROUTINE di supporto per fare lo scambio

```
SUBROUTINE SWAP_I(I1,I2)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: I1,I2
INTEGER                :: TMP

TMP=I1
I1=I2
I2=TMP
END SUBROUTINE SWAP_I
```

```
SUBROUTINE BSORT (N, VI)
IMPLICIT NONE
INTEGER, INTENT(IN) :: N
INTEGER, INTENT(INOUT) :: VI(N)
INTEGER :: I, J

! CICLO PRINCIPALE
1. DO I = 1, N-1
     ! CONFRONTO L'ELEMENTO I-ESIMO CON QUELLI CHE SEGUONO
     ! E SE NE TROVO UNO + PICCOLO LI SCAMBIO
     !
2.   DO J = I+1, N
3.     IF (VI(J) .LT. VI(I)) CALL SWAP_I (VI(I), VI(J))
4.   ENDDO
5. ENDDO
END SUBROUTINE BSORT
```

ANALISI DELL'ALGORITMO

Se tolgo 3. il ciclo rimane $O(1)$

Il ciclo 2.-4. è $O(N-I)$

Il costo dell'algoritmo varierà $O\left(\sum_{I=1}^{N-1}(N-I)\right)$

$$\sum_{I=1}^{N-1}(N-I) = N(N-1) - \frac{N(N-1)}{2} = \frac{1}{2}N(N-1) = \frac{1}{2}N^2 - \frac{1}{2}N = O(N^2)$$

Complessità quadratica

ESERCIZI:

Scrivere un programma principale che legga un vettore di interi e lo riordini usando la BSORT

- MODIFICARE BSORT in modo che ordini un array di real o di caratteri. NB: i caratteri sono ordinati secondo la tabella ASCII (cioè il FORTRAN confronta i caratteri secondo la loro posizione in tabella) e le MAIUSCOLE VENGONO TUTTE PRIMA DELLE MINUSCOLE

- MODIFICARE BSORT in modo che faccia l'ordinamento inverso, dal più grande al più piccolo.

---

È possibile utilizzare BSORT anche per ordinare più vettori contemporaneamente $V1, V2, \ldots, V_M$ in modo che tutti vicino riordinati effettuando le stesse permutazioni

ESEMPIO:

Ho una classe di alliev cinsanno con un'età diversa.

| V1 | V2 |
|---|---|
| MARIO | 21 |
| CARLO | 8 |
| GIOVANNA | 16 |
| PIPPO | 3 |
| FRANCESCA | 5 |
| GUSTAVO | 30 |

ORDINO PER ETÀ

| | |
|---|---|
| PIPPO | 3 |
| FRANCESCA | 5 |
| CARLO | 8 |
| GIOVANNA | 16 |
| MARIO | 21 |
| GUSTAVO | 30 |

ORDINO PER ALFABETO

| | |
|---|---|
| CARLO | 8 |
| FRANCESCA | 5 |
| GIOVANNA | 16 |
| GUSTAVO | 30 |
| MARIO | 21 |
| PIPPO | 3 |

Come si fa?

Basta effettuare tra le posizioni I e SV gli stessi scambi che si effettuano sul vettore che comanda. del vettore aggiuntivo

Quando ordino per età il vettore che comanda è il vettore delle età, quando ordino per alfabeto il vettore che comanda è quello dei nomi.

# Ricerca Binaria

Supponiamo di avere un vettore ordinato e di voler trovare la posizione occupata da un certo valore.
Per esempio abbiamo un vettore ordinato di interi e vogliamo trovare in che posizione si trova il valore "3". Dobbiamo fare una ricerca, esplorare il vettore e trovare fuori la posizione che ci interessa

## Utilizzo?

Per esempio abbiamo due vettori: NOMI ed ETÀ (il primo è un array di caratteri, il secondo di interi) che rappresentano i nomi e le corrispondenti età di un gruppo di persone

| | |
|---|---|
| CARLO | 8 |
| FRANCESCA | 5 |
| GIOVANNA | 16 |
| GUSTAVO | 31 |
| MARIO | 21 |
| PIPPO | 3 |

Se voglio sapere quanti bravi ha MARIO, cercherò nell'array di caratteri la posizione occupata dalla stringa "MARIO" e poi andrò nella corrispondente posizione dell'array d'interi per conoscerne l'età.

## Che algoritmo uso?

Il più semplice è scorrere la lista di caratteri dalla prima posizione all'ultima e fermarmi quando trovo il nome che mi interessa. Potrebbe anche succedere che non trovo il nome che mi interessa ed esco con una condizione di non trovato.

```
FUNCTION TROVAPOS(N,VI,II) RESULT(POS)
! la function restituisce POS=0 se non trova l'intero
! cercato
IMPLICIT NONE
! INPUT
INTEGER, INTENT(IN) :: N, VI(N), II
! OUTPUT
INTEGER :: POS
! LOCALI
INTEGER :: K
POS=0
DO KK = 1, N
   IF(VI(KK) .EQ. II) THEN
      POS = KK
      EXIT <------ RETURN
   ENDIF
ENDDO
END FUNCTION TROVAPOS
```

COMPLESSITÀ DI TROVAPOS

La complessità di questo algoritmo non è "deterministica", cioè il numero di operazioni che compie dipende dai dati che riceve.

- CASO MIGLIORE : II è in prima posizione, l'algoritmo è $O(1)$
- CASO PEGGIORE : II non è presente, deve scorrere tutto il vettore per non trovare nulla $O(n)$.

Bisogna valutare il CASO MEDIO: in generale eseguirà un numero di confronti che sarà proporzionale alla lunghezza di VI e quindi l'algoritmo è $O(n)$.

Si può fare meglio!!! ⇒ RICERCA DINARIA

Sfrutta il fatto che l'array in cui effettuare la ricerca è ordinato e utilizza un algoritmo analogo a quello del metodo di bisezione.

! Inizializzazione
istart = 1
iend = n
POS = 0
! Controllo che il valore cercato sia uno tra
! gli estremi
Se (ii < v(1)) o (ii > v(n)) esci

Fino a che (iend - istart) > 1 :
imid = (istart + iend)/2
Se (v(imid) > ii) allora
iend = imid
altrimenti
istart = imid
Fine se
Fine

Se (v(istart) = ii) pos = istart
Se (v(iend) = ii) pos = iend

COMPLESSITÀ : l'algoritmo divide l'intervallo in 2 fino a che questo non ha dimensione 1.
Dopo k cicli l'intervallo avrà dimensione : $\frac{n}{2^k}$
Avrà quindi bisogno di un k t.c. $\frac{n}{2^k} = 1 \iff 2^k = n \Rightarrow$

$\Rightarrow k = \log_2 n$

Il mio algoritmo di ricerca binaria ha complessità ottimale: $O(\log_2 n)$ : logaritmica

# RICORSIONE

Una funzionalità importante di molti linguaggi di programmazione è la possibilità di scrivere SOTTOPROGRAMMI RICORSIVI, cioè sotto-programmi che CHIAMANO SE STESSI e funzionano in maniera INDUTTIVA.

- CHIAMANO SE STESSI: Cosa significa?

Significa che nel corpo delle SUBROUTINE o delle FUNCTION c'è una chiamata alla SUBROUTINE o alla FUNCTION stessa.

- Funzionamento: "INDUTTIVO": quando vogliamo definire il principio di induzione si definisce una BASE, cioè a cosa deve valere la proprietà nel caso $n = n_0$ e poi si definisce una REGOLA secondo cui a questa proprietà nel caso $n$ allora dovremo dimostrare la proprietà per $n+1$. Per questo motivo è importante che la data proprietà valga per ogni $n$ finito!

Esempio: il fattoriale ha una definizione INDUTTIVA:

BASE: $0! = 1$

REGOLA: $(n+1)! = (n+1)\,n!$

Lo stesso vale dobbiamo fare per molti sottoprogrammi ricorsivi:

1) Definiamo una BASE, cioè un comportamento standard quando l'argomento è uguale a $n_0$

2) Definiamo una REGOLA con un valore da $n$ a $n+1$

NB: È molto importante definire una base, altrimenti il programma può cadere in un LOOP INFINITO, continua a chiamarsi e chiamarsi all'infinito.

Il FORTRAN 77 non prevedeva l'uso di programmi ricorsivi. Per fortuna con il FORTRAN 90 sono stati introdotti !!!

Per scrivere una FUNCTION o SUBROUTINE ricorsiva è sufficiente inserire la parola RECURSIVE prima di FUNCTION o SUBROUTINE.

Esempio:

```
RECURSIVE FUNCTION Rfatt(n) RESULT(fatt)

IMPLICIT NONE
INTEGER, INTENT(IN) :: n
INTEGER             :: fatt

IF (n .eq. 0) THEN
   ! Base
   fatt = 1
ELSE
   ! Regola
   fatt = n * Rfatt(n-1)
ENDIF
END FUNCTION Rfatt
```

Questa funzione funziona quasi sempre bene, ma ha un problema: quale?