# Problemi, algoritmi e programmi di matematica elementare

Note per il corso di **Laboratorio di Programmazione e Calcolo**
A.A. 2013/14

a cura di Stefano FINZI VITA

(dicembre 2013)

## 1 Introduzione

In queste note vogliamo affrontare semplici problemi che non richiedono particolari conoscenze matematiche oltre a quelle già possedute da uno studente appena uscito dal liceo, e che possono essere usati come 'palestra' per l'avviamento alla programmazione in C++ e alla risoluzione di problemi più complessi.

Ogni problema verrà analizzato in dettaglio, richiamando quanto serve della teoria (introducendo nuove definizioni ove necessario) per pervenire ad un algoritmo in grado di risolverlo. In qualche caso si confronteranno più algoritmi per lo stesso scopo, al fine di valutare quale sia più efficiente od 'economico' (contando ad esempio il numero di operazioni necessarie per arrivare al risultato cercato). Perché in generale non esiste LA soluzione del problema, ma diverse soluzioni possono condurre allo stesso scopo. Perciò sentitevi autorizzati a proporre e testare voi stessi altre varianti risolutive.

Alla fine di ogni paragrafo si discuteranno i problemi da affrontare per tradurre gli algoritmi in C++. Il più delle volte non si scriveranno dei listati completi ma solo le istruzioni chiave, perché lo scopo è di stimolarvi a ragionare e di abituarvi a risolvere problemi via via più complicati, non quello di trasformarvi in passivi 'copiatori' di programmi.

Qui diamo per note le istruzioni base del linguaggio C++ (in particolare quelle che riguardano i cicli, le alternative, la gestione delle variabili e degli array), che del resto si possono trovare in un qualunque manuale. In generale invece non faremo uso delle funzioni: è stata una scelta consapevole, e data la semplicità della maggior parte dei programmi considerati non particolarmente penalizzante. Tutti gli algoritmi analizzati possono del resto facilmente essere trasformati in funzioni da utilizzare in programmi più complessi come quelli che vedremo nella seconda parte del corso. Solo alla fine si è fatta un'eccezione per mostrare come grazie alla ricorsività (la possibilità che in C++ hanno le funzioni di richiamare sé stesse) alcuni degli algoritmi precedentemente esaminati possono assumere una forma molto snella ed efficiente.

Queste note sono un 'work in progress': sono pertanto gradite osservazioni, richieste di approfondimenti o segnalazioni di errori.

## 2 Lettura/scrittura di un numero

Poiché in ogni problema vanno introdotti dei dati, iniziamo con alcune semplici considerazioni al riguardo. Nell'introdurre le funzioni di input/output si è preferito partire da quelle ereditate nel C++ dal precedente linguaggio C, cioè le istruzioni *printf, scanf* della libreria < *stdio.h* >. Il motivo è che per essere usate queste richiedono di riflettere attentamente sul tipo delle variabili in gioco, e ciò riteniamo sia molto opportuno per chi si accinga a programmare in un linguaggio dove ogni variabile va dichiarata. Ovviamente, una volta aquisita un po' di esperienza, potete passare alle più snelle istruzioni del C++, cioè *cin* e *cout*, includendo la libreria < *iostream* >. A voi la scelta finale.

Leggere un numero (da tastiera) richiede dunque di aver prima dichiarato il tipo di variabile destinata a contenerlo (per esempio *int, float* o *double*). Il compilatore così saprà quanti bytes riservare in memoria per il suo stockaggio. Il comando di lettura (di C ma come detto ereditato nel C++) è *scanf*, e richiede di precisare (tra virgolette) il formato della variabile, seguito dall'indirizzo della stessa. Il simbolo & che infatti precede la variabile ha il significato di puntatore all'indirizzo della posizione di memoria che andrà a contenere il numero letto.

**Esempio 1**. Leggiamo un numero intero. Il comando di stampa *printf* che precede la lettura vera e propria non sarebbe necessario, ma è fortemente consigliato per permettere a chi lancia il programma di sapere con precisione quale dato deve inserire (in genere chi usa un programma non è necessariamente la stessa persona che lo ha scritto, e comunque se il programma è complicato e prevede la lettura di molti dati, diverrebbe impossibile ricordare l'esatta sequenza del loro inserimento).

```
//  leggi_intero.cpp
#include <stdio.h>
main()
  {int n;
   printf("Dammi un numero intero: n= ");
   scanf("%d",&n);
   return 0;}
```

**Esempio 2.** Per leggere invece un numero reale basterà sostituire nelle istruzioni precedenti la corretta dichiarazione di tipo (*float* o *double*) e il formato corrispondente sia in *printf* che in *scanf*. Quindi ad esempio:

```
//  leggi_reale.cpp
#include <stdio.h>
main()
{float x;
  printf("Dammi un numero reale: x= ");
  scanf("%f",&x);
  return 0;}
```

**Esempio 3.** (Lettura controllata)
Spesso accade che il dato da inserire debba soddisfare alcune condizioni (positivo, minore di un numero prefissato, appartenente a un certo intervallo, ecc.). Ovviamente si può fare un controllo a posteriori dopo la lettura, segnalando l'errore eventuale ed uscendo dal programma. Ad esempio:

```
 main()
```

```
{ ...
 printf("Dammi un numero intero positivo: n= ");
  scanf("%d",&n);
if (n<=0) {printf("ERRORE: il numero inserito non va bene.\n"); return 0;}
 ...
 return 0;}
```

È però più efficiente inserire il controllo durante la lettura stessa, in modo da permettere l'inserimento di un nuovo dato se il primo non soddisfa la condizione voluta. Lo consente l'istruzione *do .. while*, e la condizione da mettere tra parentesi deve ovviamente essere contraria a quella richiesta. Nello stesso esempio precedente:

```
 main()
{ ...
do
   {printf("Dammi un numero intero positivo: n= ");
    scanf("%d",&n);}
while (n<=0);
 ...
 return 0;}
```

Si uscirà dal ciclo appena la condizione tra parentesi diventerà falsa, cioè appena il dato inserito sarà positivo come voluto. Altrimenti la richiesta verr ripetuta. Analogamente, volendo leggere un numero reale $x \in [2, 10[$, scriveremo:

```
do
   {printf("Dammi un numero x, con 2<=x<10 : ");
    scanf("%f",&x);}
while (x<2 || x>=10);
```

Attenzione: si può anche usare l'istruzione *while* al posto di *do while*, ma in tal caso occorre premettere l'assegnazione fittizia di un valore scorretto alla variabile da leggere. Esempio:

```
n=0;
while (n<=0)
    {printf("Dammi un numero intero positivo: n= ");
    scanf("%d",&n);}
```

# 3 I numeri macchina

Abbiamo discusso a lungo nelle lezioni introduttive sul fatto che mentre i numeri reali costituiscono un continuo, quelli macchina, cioè i numeri effettivamente rappresentabili nella memoria di un calcolatore, formano un 'colabrodo'. Si è visto come ogni numero reale venga approssimato dal più vicino numero macchina, come nell'algebra dei numeri macchina diventino false alcune identità matematiche, e come gli errori di queste approssimazioni possano venire propagati e amplificati da alcune operazioni. In questa sezione vogliamo rendere evidenti questi fenomeni attraverso alcuni semplici programmi dimostrativi, ma li sottolineeremo anche in seguito ogni volta che si presenteranno.

## 3.1 Integer overflow

Abbiamo visto che in C++ gli interi possono essere rappresentati mediante due bytes (*short int*) o quattro bytes (*int*). Questo significa che esiste un massimo numero intero rappresentabile per ognuno dei due tipi. Ricordando che i numeri macchina sono rappresentati in base 2, non è difficile caratterizzare questi numeri: tenendo conto che un bit è riservato al segno, l'ordine di grandezza sarà rispettivamente $2^{15}$ e $2^{31}$. Nel programma seguente l'abbiamo voluto verificare, generando la successione $2^n$ per variabili *short int* ($a_n$) e *int* ($b_n$). Come si è detto, l'*integer overflow* non viene segnalato, ma la successione fornisce in tal caso valori errati (prima rientra 'dal fondo', indicando il più grande numero intero negativo, poi diventa nulla).

```
main (void) {
short int a; int i, b;
a=1; b=1;
printf("SHORT INT\n");
for (i=1;i<17;i++)
     {a=a*2; printf("a_%d = %d \n",i,a);}
printf("\nINT\n");
for (i=1;i<33;i++)
     {b=b*2; printf("b_%d = %d \n",i,b);}
```

Ecco una porzione dell'output relativo:

```
SHORT INT
a_1 = 2
a_2 = 4
...........
a_12 = 4096
a_13 = 8192
a_14 = 16384
a_15 = -32768   !!
a_16 = 0

INT
b_1 = 2
b_2 = 4
...........
b_14= 16384
b_15= 32768
b_16= 65536
...........
b_28 = 268435456
b_29 = 536870912
b_30 = 1073741824
b_31 = -2147483648   !!
b_32 = 0
```

Si faccia ad esempio la prova di sommare i due numeri interi $a = 14000$ e $b = 27000$. Se $a$ e $b$ sono dichiarati *int*, la loro somma fornirà correttamente il valore 41000, ma se sono dichiarati *short int* apparirà il numero -24536 (siete usciti dal range dei numeri rappresentabili in *short int* e rientrati dal fondo).

## 3.2 Lo zero macchina e la precisione macchina, ovvero: 'la solitudine dei numeri interi'

Possiamo definire lo *zero macchina* come il più piccolo numero reale positivo rappresentabile sul nostro computer. Mentre chiameremo *precisione macchina* il più piccolo numero positivo $\varepsilon$ tale che $1 + \varepsilon > 1$. Sono quantità che dipendono appunto dalla macchina in uso, e che in C++ saranno diverse per le variabili *float* da quelle *double*. Attenzione alla differenza tra le due definizioni. Lo zero macchina ci dice qual è il primo numero rappresentabile a destra dello zero, quindi ci dà un'idea della 'solitudine del numero zero', e grazie alla rappresentazione in virgola mobile può davvero essere molto piccolo: le cifre significative del numero possono infatti scorrere liberamente verso destra giocando sulla parte esponenziale. La precisione macchina ci dà invece la misura della 'solitudine del numero 1', cioè della taglia dei buchi più grandi del nostro 'colabrodo', indicando lo spazio vuoto che c'è tra un numero intero non nullo e il successivo numero rappresentabile: poichè stavolta non possiamo perdere la prima cifra significativa relativa all'unità, agire sull'esponente non ci aiuterà molto. Sapendo quanti bytes sono usati per la memorizzazione di un numero reale in virgola mobile (per mantissa ed esponente), si possono calcolare esattamente le due quantità. Col seguente programma ci vogliamo soltanto arrivare vicini per via sperimentale. L'idea empirica è quella di partire da un valore positivo di $\varepsilon_0$ assegnato da tastiera (piccolo ma non troppo), e poi di generare a partire da esso una successione per ricorrenza $\varepsilon_n = f(\varepsilon_{n-1})$ decrescente verso zero, finché non accade appunto che $1 + \varepsilon_n = 1$. Il valore $\varepsilon_{n-1}$ sarà dunque la nostra stima della precisione macchina. Analogamente il più piccolo valore $\varepsilon_n \neq 0$ ci darà una stima dello zero macchina. Ovviamente successioni diverse ci porteranno a stime diverse. Nel programma che segue ci siamo limitati a dividere ogni volta il valore precedente per 2 ($f(\varepsilon_{n-1}) = \varepsilon_{n-1}/2$), e abbiamo ripetuto lo stesso procedimento due volte (per variabili *float* e *double*):

```
// Questo programma stima la precisione macchina in float e double
#include <stdio.h>
#include <math.h>

main()
{float feps; double deps;
printf("eps = ");  scanf("%f",&feps);
deps=feps;

while (1+feps>1)
     {feps=feps/2;
      printf("%e\t %1.18f\n",feps,1+feps);}
printf("precisioneF = %e\n\n\n",2*feps);

while (1+deps>1)
     {deps=deps/2;
      printf("%e\t %1.18f\n",deps,1+deps);}
printf("precisioneD = %e\n\n\n",2*deps);
return 0;}
```

Sul mio Mac ad esempio, questo programma partendo da $\varepsilon_0 = 0.1$ fornisce per la precisione macchina le stime:

$$\varepsilon_{float} = 9.536743\, e - 08, \quad \varepsilon_{double} = 1.776357\, e - 17,$$

mentre la stima dello zero macchina risulta rispettivamente $1.401298\, e - 45$ in *float* e $4.940656\, e - 324$ in *double*.

### 3.3 Divisione tra interi

Come si è visto in C++ è essenziale dichiarare correttamente ogni variabile. Qui più che evidenziare un fenomeno legato ai vuoti nell'insieme dei numeri macchina, vogliamo appunto mettere in guardia dalla possibilità che i nostri conti vengano gravemente falsati da una errata dichiarazione (o gestione) delle variabili in gioco, anche se la logica del programma è corretta e il compilatore non ci dà nessun errore.

Uno degli errori più frequenti può avvenire quando si dividono tra loro due numeri interi: se tali numeri sono memorizzati in variabili intere nella divisione l'eventuale parte decimale andrà persa, con ovvie conseguenze. Nel seguente programma abbiamo calcolato in molti modi la divisione di 1 per 2 mostrando diverse situazioni, alcune corrette e altre no (si vedano i commenti al listato):

```
/* Dati due interi n,m calcola n/m, confrontando i risultati in base
   ai tipi di variabili utilizzati*/
#include <stdio.h>
main ()
{int n=1, m=2, k, l=0; float h, s=2, t=0;

 k=n/m; printf("k=n/m= %d \n",k); //   int/int --> int NO

 h=n/m; printf("h=n/m= %f \n",h); //   int/int --> float NO

 k=1/m; printf("k=1/m= %d \n",k); //   1/int --> int NO

 h=1./m; printf("h=1./m= %f \n",h); //  1./int --> float OK

 k=1./s; printf("k=1./s= %d \n",k); //  1./float --> int NO

 h=float(n)/m; printf("h=float(n)/m= %f \n",h); // f(int)/int --> float OK

 h=n/float(m); printf("h=n/float(m)= %f \n",h); // int/f(int) --> float OK

 h=(float)n/m; printf("h=(float)n/m= %f \n",h); // (f)int/int --> float OK

 k=1/l; printf("k=1/0= %d \n",k); //   int/0 --> int   F.P.E.

 h=1/t; printf("h=s/0= %f \n",h);  //  float/0 --> float  INF

 return 0;}
```

Qualche commento: solo le istruzioni che terminano col commento OK forniranno il valore corretto 0.5 (provare!). Negli altri casi si otterrà sempre 0. Come si vede la soluzione non è quella di dichiarare *float* tutte le variabili, ma di forzare la divisione mediante il *cast* a *float*, ricordando ovviamente di assegnare il risultato ad una variabile *float*, se no perderemmo di nuovo quello che abbiamo faticosamente conservato. Nelle ultime due righe si descrive invece l'effetto della divisione per zero in variabili *int* (appare il messaggio d'errore *Floating point exception*) e *float* (nessun messaggio, ma il risultato viene posto uguale a *inf*, ossia il massimo numero macchina di tipo *float*).

## 3.4 Una successione numerica

In questo esempio vogliamo generare i numeri della successione numerica

$$a_k = \sqrt{k+1} - \sqrt{k};$$

Ovviamente nessuno pretende di calcolare il limite di una successione con un computer, per quello esistono le tecniche dell'analisi matematica. Tutt'al più in certi casi potremo avere l'evidenza di come i suoi valori vadano accumulandosi verso un numero definito. Qui sappiamo già che la successione è infinitesima (cioè tende a zero). Il modo semplice di vederlo è di trasformare la sua espressione di tipo indeterminato ($\infty - \infty$) in un'altra mediante l'identità

$$\sqrt{k+1} - \sqrt{k} = \frac{1}{\sqrt{k+1} + \sqrt{k}}$$

e osservare che il secondo membro tende chiaramente a zero. Qui ci interessa confrontare il diverso comportamento delle due espressioni precedenti, matematicamente equivalenti, mettendone a confronto i valori generati dal computer quando $k = 10^n$, al crescere di $n$. Ecco il programma per farlo:

```
int main()
    {float k=1, ak=1, bk=1;
     for (int i=1;i<=12;i++)
        {k*=10;
        ak=sqrt(k+1)-sqrt(k);
        bk=1/(sqrt(k)+sqrt(k+1));
        printf("%1.0e\t ak= %1.10e,\t bk=%1.10e\n\n",k,ak,bk);}
     return 0;}
```

ed ecco l'output:

```
k=1e+01  ak= 1.5434713662e-01, bk=1.5434713662e-01
k=1e+02  ak= 4.9875620753e-02, bk=4.9875620753e-02
k=1e+03  ak= 1.5807436779e-02, bk=1.5807436779e-02
k=1e+04  ak= 4.9998750910e-03, bk=4.9998750910e-03
k=1e+05  ak= 1.5811348567e-03, bk=1.5811348567e-03
k=1e+06  ak= 4.9999984913e-04, bk=4.9999984913e-04
k=1e+07  ak= 1.5811387857e-04, bk=1.5811387857e-04
k=1e+08  ak= 0.0000000000e+00, bk=4.9999998737e-05
k=1e+09  ak= 0.0000000000e+00, bk=1.5811388948e-05
k=1e+10  ak= 0.0000000000e+00, bk=4.9999998737e-06
k=1e+11  ak= 0.0000000000e+00, bk=1.5811388039e-06
k=1e+12  ak= 0.0000000000e+00, bk=4.9999999874e-07
```

Come si vede la successione di partenza (quella scritta come differenza di radici) già per $k$ pari a cento milioni fornisce valori nulli, mentre la sua versione equivalente mostra che il suo valore corretto è senz'altro piccolo ma non troppo, dell'ordine di $10^-5$, e continua a decrescere regolarmente se aumentiamo $k$. È un esempio evidente di quello che abbiamo mostrato essere il fenomeno di propagazione degli errori di arrotondamento in caso di differenza di valori molto simili tra loro: al crescere di $k$ le due radici nella prima espressione assumeranno valori con un numero sempre maggiore di cifre significative in comune. Sottrarle causerà quindi la perdita di cifre significative con conseguente errore nel risultato. Fenomeno che non avviene usando l'altra espressione dove le due quantità sono sommate invece che sottratte. Occorre quindi fare attenzione alle espressioni matematiche che si usano, e se possibile trasformarle in altre equivalenti ma più sicure (o, come si dice, più *stabili*).

# 4 Le equazioni di secondo grado

Tutti ricordano come si risolve un'equazione di secondo grado

$$ax^2 + bx + c = 0 \, .$$

Volendo scrivere una procedura per la sua risoluzione, lo pseudocodice potrebbe essere più o meno il seguente:

1. Leggi $a, b, c$.
2. Calcola $\Delta = b^2 - 4ac$.
3. Se $\Delta > 0$, poni
   $x_1 = \frac{-b-\sqrt{\Delta}}{2a}, \quad x_2 = \frac{-b+\sqrt{\Delta}}{2a}$.
4. Altrimenti, se $\Delta = 0$, poni
   $x_1 = \frac{-b}{2a}; \quad x_2 = x_1$.
5. Altrimenti poni
   $x_1 = \frac{-b}{2a} + i\frac{\sqrt{-\Delta}}{2a}; \quad x_2 = \frac{-b}{2a} - i\frac{\sqrt{-\Delta}}{2a}$.

Tutto bene, se non che abbiamo scoperto nel paragrafo sui numeri macchina e sulla propagazione degli errori di arrotondamento che certe operazioni sono potenzialmente pericolose, in particolare sappiamo che le sottrazioni tra quantità quasi uguali possono portare alla cancellazione di cifre significative del risultato, che viene così profondamente modificato. Vediamo il seguente esempio:

**Esempio.** Assumiamo $a = 10^{-8}, b = c = 1$. Allora l'algoritmo precedente fornirà:

$$x_1 = -10^8 \text{ (con residuo } r(x_1) = 1), \quad x_2 = -1.490116 \text{ (con residuo } r(x_2) \simeq 0.49)$$

Per residuo $r(z)$ intendiamo il valore dell'espressione $az^2 + bz + c$, che ci dà un'idea di quanto la soluzione trovata soddisfi l'equazione di partenza. Ovviamente in aritmetica esatta il residuo sarebbe zero. Il primo residuo ci fornisce un errore relativo di 1 su cento milioni, decisamente accettabile, mentre il secondo indica un errore di circa il cinquanta per cento (!), e ci dice che la soluzione $x_2$ non è accurata.

Il problema sta nelle formule risolutive del punto 3.: se $4ac << b^2$, allora $\sqrt{\Delta} \simeq |b|$, e l'espressione per il calcolo di $x_1$ (se $b < 0$) o quella per il calcolo di $x_2$ (se $b > 0$) risulterà 'instabile'. A questo problema si può ovviare ricorrendo ad altre formule, matematicamente equivalenti, ma più stabili. Vale infatti:

$$\frac{-b \pm \sqrt{\Delta}}{2a} = \frac{-b \pm \sqrt{\Delta}}{2a} \frac{-b \mp \sqrt{\Delta}}{-b \mp \sqrt{\Delta}} = \frac{2c}{-b \mp \sqrt{\Delta}}$$

dove quando nella prima formula scegliamo il segno '+', nella seconda prenderemo quello '-', e viceversa. Un algoritmo stabile per le equazioni di secondo grado può quindi ricorrere a una formula o all'altra, in base al segno di $b$. Ecco come potrebbe essere in C++ il corpo del programma relativamente al solo caso delle radici distinte:

```
if (delta>0)
    { if (b>0)
       {x1=(-b-sqrt(delta))/(2*a);  x2=(2*c)/(-b-sqrt(delta));}
      else
  {x1=(2*c)/(-b+sqrt(delta)); x2=(-b+sqrt(delta))/(2*a);}
      printf("\n le due radici sono \n x1 = %f\n x2 = %f\n",x1,x2);}
```

Tornando all'esempio di prima, con questo accorgimento ora otterremmo:

$$x_1 = -10^8, \quad x_2 = -1,$$

e la seconda radice risulta molto più accurata di prima: ora il residuo è praticamente zero. Se avessimo invece calcolato $x_1$ con la formula instabile, avremmo ottenuto:

$$x_1 = -67108864, \text{ con residuo } r(x_1) = -2.207287e + 07 \text{ !}$$

# 5 Somme e prodotti di numeri

In molte situazioni ci si trova a sommare o moltiplicare tra loro diverse quantità indicizzate. Ad esempio:

$$somma = \sum_{i=1}^{n} a_i = a_1 + a_2 + .. + a_n \ ; \quad prodotto = \prod_{i=1}^{n} b_i = b_1 * b_2 * .. * b_n \ .$$

Per calcolare queste sommatorie o produttorie lo strumento ideale è un ciclo attraverso il quale andiamo accumulando tali quantità in una variabile opportuna. La variabile dove accumuleremo addendi andrà inizializzata a zero, quella dove accumuleremo fattori andrà inizializzata a 1. Per restare ai due esempi precedenti gli schemi (o pseudocodici) da implementare diventeranno:

1) $somma = 0$;
per $i = 1, .., n$ calcola: $somma = somma + a_i$
2) $prodotto = 1$;
per $i = 1, .., n$ calcola: $prodotto = prodotto * b_i$

Facciamo qualche esempio concreto.

## 5.1 Potenza intera di un numero

Nel linguaggio C++ non esiste un operatore aritmetico per l'elevamento a potenza come in altri linguaggi. Esiste la funzione $pow(x, b)$ della libreria $< math.h >$, che calcola la potenza $x^b$ per base ed esponente reali. Ma se ci interessano le potenze intere di un dato numero, è sufficiente ricorrere a un rapido ciclo moltiplicativo. Infatti $q^n = q * q * ... * q$, e rientriamo nello schema precedente:

$$pot = 1 \ ; \quad \text{per } i = 1, .., n \text{ calcola: } pot = pot * q \ .$$

Si noti che questa volta l'indice $i$ non compare nelle espressioni da calcolare, ma serve solo a contare il numero dei fattori in gioco, tutti uguali tra loro. Possiamo anche risparmiare un prodotto ponendo $pot = q$ e sommando fino all'indice $n - 1$.

## 5.2 Somma della progressione geometrica

Come ulteriore applicazione consideriamo ora un caso in cui possiamo sfruttare in pieno quanto detto in precedenza. Sia $q$ un numero reale assegnato; siamo interessati alla quantità:

$$S_n(q) = 1 + q + q^2 + .. + q^n = \sum_{i=0}^{n} q^i,$$

cioè alla somma dei primi $n$ termini della progressione geometrica di ragione $q$. Siamo in presenza di una sommatoria di potenze di esponente via via crescente. Possiamo

quindi usare un ciclo per la sommatoria, e un ciclo per calcolare ogni volta la potenza $q^i$ come visto al paragrafo precedente. Il costo totale dell'algoritmo in termini di numero di operazioni sarebbe perciò di $1 + 2 + .. + (n-1) = n * (n-1)/2$ prodotti per le potenze, e di $n$ somme. Ma si può fare di meglio: poiché ogni volta la nuova potenza da sommare non è altro che quella sommata al passo precedente moltiplicata per $q$, ci basterà un solo ciclo e una sola moltiplicazione per ogni nuovo addendo se avremo cura di memorizzare quanto già fatto, quindi in totale solo $n$ prodotti e $n$ somme. Ecco come potrebbe funzionare:

```
float q, x=1, s=1;
for (int i=1;i<=n;i++)
      {x*=q; s+=x;}
```

Nella variabile $x$ (inizializzata ad 1) andranno accumulandosi via via le potenze successive di $q$ da aggiungere ad $s$, la variabile che alla fine conterrà la sommatoria richiesta. Qualcuno potrebbe osservare che si prova facilmente (ad esempio per induzione) che per $q \neq 1$ vale la relazione:

$$1 + q + q^2 + .. + q^n = \frac{1 - q^{n+1}}{1 - q},$$

per cui per calcolare $S_n(q)$ avremmo potuto eseguire direttamente il quoziente a destra: si noti però che ci sarebbero serviti comunque $n$ prodotti per il calcolo di $q^{n+1}$, un quoziente e due somme.

## 5.3 Un po' di statistica: le medie di un insieme di dati

In statistica, dovendo analizzare un insieme di dati, risulta utile il concetto di *media*, un singolo valore numerico che possiamo associare all'intero insieme per descriverlo in modo sintetico. Esistono diversi tipi di media, con caratteristiche e impieghi a volte differenti. Richiamiamo qui le definizioni delle medie più importanti associabili all'insieme $A = \{x_1, x_2, .., x_n\}$ di $n$ numeri:

- *Media aritmetica*

$$M_a = \frac{x_1 + x_2 + .. + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- *Media geometrica*

$$M_g = (x_1 * x_2 * .. * x_n)^{1/n} = (\prod_{i=1}^{n} x_i)^{1/n}$$

- *Media armonica* (è il reciproco della media aritmetica dei reciproci)

$$M_h = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + .. + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

- *Media quadratica*

$$M_q = \sqrt{\frac{x_1^2 + x_2^2 + .. + x_n^2}{n}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

Senza volere entrare nello specifico di queste definizioni, analizzandone proprietà e applicazioni (si guardi un qualunque testo di statistica o anche solo la voce relativa di Wikipedia), scriviamo un programma che le calcoli su di un insieme arbitrario di dati. Ecco come potrebbe essere il *main* del programma C++ se non abbiamo necessità di memorizzare i numeri inseriti una volta letti:

```
main() {
  int i,n; float x,ma,mg,mh,mq;
  printf("Inserisci il numero n\n");
  scanf("%d",&n);
  ma=0; mg=1; mh=0; mq=0;
  for (i=1;i<=n;i++)
     {printf("\n leggi il numero %d ",i); scanf("%f",&x);
      ma=ma+x;
      mg=mg*x;
      mh=mh+1/x;
      mq=mq+x*x; }
  ma=ma/n;
  mg=pow(mg,(float)1/n);
  mh=n/mh;
  mq=sqrt(mq/n);
  printf("\n ma= %f,  mg= %f,  mh= %f,  mq= %f\n",ma,mg,mh,mq);
return 0;}
```

N.B. Osservare il *cast* effettuato sull'esponente $1/n$ nella funzione *pow*: senza di quello la divisione tra interi avrebbe fornito un esponente nullo falsando completamente il risultato.

Se ripetete questo programma per diversi insiemi di dati dovreste verificare che risulta sempre:

$$M_h \leq M_g \leq M_a \leq M_q.$$

In particolare notate, come è giusto che sia, che se tutti i dati sono uguali allo stesso valore $c$ allora le quattro medie saranno tutte uguali tra loro (e varranno ovviamente $c$).

# 6 Pari e dispari, divisori, multipli e classi resto

Per stabilire se un numero intero $n$ risulti pari o dispari possiamo ricorrere al valore del resto della divisione di $n$ per 2. Se tale resto varrà 0 il numero sarà pari, altrimenti dispari. In C abbiamo a disposizione a questo scopo l'operatore %. Una volta letto il numero e memorizzato ad esempio nella variabile $n$, basterà scrivere:

```
if (n%2==0)
    printf('Il numero %d e' pari", n);
else
    printf("Il numero %d e' dispari", n);
```

Un'altra possibilità è quella di giocare sul seguente fatto: poiché $n$ sarà una variabile di tipo *int*, la divisione di $n$ per 2 fornirà un risultato *int* e si avrà:

$$n = (n/2) * 2 \Leftrightarrow n \text{ pari}$$

Ad esempio, per $n = 6$ avremo $n/2 = 3$ e $3 * 2 = 6 = n$. Invece per $n = 7$ si avrà $n/2 = 3$ (!) e quindi $3 * 2 = 6 \neq n$. Con questa idea le righe precedenti potrebbero essere sostituite da:

```
if (n/2*2==n)
   printf('Il numero %d e' pari", n);
else
   printf("Il numero %d e' dispari", n);
```

Più in generale un numero intero $a$ sarà un *multiplo* di un altro numero $b$ (o, il che è lo stesso, $b$ sarà un *divisore* di $a$) se la divisione di $a$ per $b$ dà resto zero (in linguaggio C se $a\%b = 0$). Vedremo nel prossimo paragrafo come calcolare tutti i divisori di un numero e nel paragrafo 8 come calcolare massimo comun divisore e minimo comune multiplo di due o più numeri dati.

Il resto della divisione tra interi è anche alla base della suddivisione dei numeri interi in *classi resto modulo n*. È sufficiente definire la seguente relazione di equivalenza:

$$a \equiv b \mod n \ (a \text{ congruo } b \text{ modulo } n) \Leftrightarrow (a - b) \text{ è multiplo di } n.$$

Osserviamo che se $a$ e $b$ sono entrambi positivi, la definizione precedente equivale ad affermare che i due numeri divisi per $n$ danno lo stesso resto (cioè $a\%n = b\%n$). Questa relazione suddivide tutto l'insieme dei numeri interi in esattamente $n$ classi di equivalenza, in base al valore del resto della divisione per $n$: $[0], [1], [2], .., [n-1]$. Ad esempio se $n = 2$ le 2 classi di equivalenza coincideranno con l'insieme dei numeri pari (la classe [0]) e quello dei numeri dispari (la classe [1]). Oppure se $n = 5$, la classe $[3] \mod 5$ sarà ad esempio l'insieme

$$[3] = \{3, 8, 13, 18, ..., -2, -7, -12, ...\}$$

Attenzione che per un numero negativo la determinazione della classe resto di appartenenza non si può ottenere direttamente in base al resto della divisione per $n$. Occorre prima traslarlo nell'insieme dei numeri positivi mediante multipli di $n$. Ad esempio il numero -7 appartiene alla classe $[3] \mod 5$ perché $-7 + 2 * 5 = 3$.

Le classi resto mod $p$ ci possono essere utili per stampare una lista di numeri a gruppi di $p$ (cioè $p$ per riga), come nel caso di una tabella. Ad esempio per stampare i numeri interi da 1 a 25 su 5 righe basterà scrivere:

```
for (int i=1;i<=25;i++)
    {printf("%d\t",i)
    if (i%5==0) printf("\n");}
```

# 7 Numeri primi

Un numero naturale è detto *primo* se risulta divisibile solo per 1 e per sé stesso. Ne esistono infiniti (questo si può dimostrare), ma la loro distribuzione è irregolare e non possono essere generati in maniera automatica. Esistono solo stime asintotiche della loro frequenza, e ancora oggi ne vengono 'scoperti' di nuovi, ovviamente grazie ai computer. Numerosi risultati in teoria dei numeri riguardano i numeri primi, alcuni dimostrati, altri ancora in forma di congetture. La loro importanza è cresciuta negli ultimi tempi con lo sviluppo della crittografia a chiave pubblica, che sfrutta la difficoltà di fattorizzare numeri interi molto grandi formati da due soli numeri primi. Noi qui affronteremo alcuni semplici problemi che li riguardano.

## 7.1 Il numero dato è primo?

Vogliamo scrivere un programma che dato un numero intero $n > 1$ stabilisca se $n$ è primo. Dalla definizione, $n$ sarà primo se i suoi unici divisori saranno quelli banali, l'unità e sé stesso. Scriviamo allora un programma che calcoli tutti i divisori di $n$ e come conseguenza stabilisca se si tratta di un numero primo. È chiaro che se non ci interessassero tutti i divisori potremmo far terminare il programma appena ne viene trovato uno non banale, stampando la risposta: $n$ NON È PRIMO. Ecco un possibile listato:

```
main()
{int i,j,n; int v[20];
 do
    {printf("Inserire n>1: n= "); scanf("%d",&n);}
 while (n<2);
 j=0;
 for (i=1;i<=n/2;i++)
      if (n%i==0)
         {v[j]=i;j++;}
 v[j]=n;
 if (v[1]==n) printf("\t%d e' primo!",n);
 printf("I divisori di %d sono: \n",n);
 for (i=0;i<=j;i++)
      printf("%d ",v[i]);
 return 0;}
```

Osservate che il ciclo per la ricerca dei divisori arriva solo fino a $n/2$, in quanto non potranno esserci divisori propri di $n$ superiori a quel numero. Per memorizzare i divisori li immagazziniamo in un array $v$, incrementando ogni volta l'indice $j$ della prima componente libera. La prima componente conterrà necessariamente l'unità ($v[0] = 1$). Se non vengono trovati altri divisori, terminato il ciclo verrà memorizzato in $v[1]$ il numero $n$ stesso. Sarà quello il modo di riconoscere che $n$ è primo. Alla fine vengono comunque stampati tutti i divisori trovati.

## 7.2 Il crivello di Eratostene

Vogliamo rispondere ora alla seguente domanda:
*Assegnato un intero positivo N, quali (e quanti sono) i numeri primi inferiori a N?*
Potremmo ricorrere all'algoritmo per stabilire se un numero è primo (si veda il paragrafo precedente) applicandolo via via a tutti i numeri minori di $N$, ma questo sarebbe enormemente dispendioso. Esiste un procedimento ingegnoso molto più semplice che risale adirittura al matematico greco Eratostene di Cirene (circa 200 a.C.) e che si basa sull'idea del setaccio (il crivello), o meglio di più setacci successivi. Se riusciamo a eliminare dal nostro insieme dei numeri minori di $N$ prima tutti quelli multipli di 2, poi quelli multipli di 3, di 5, ecc., alla fine dovremmo ritrovarci tra le mani soltanto i numeri primi. Vediamo meglio il suo funzionamento. Elenchiamo ad esempio tutti i numeri da 2 a 50:
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50

Col primo setaccio, dopo aver evidenziato il primo numero della lista, il 2, eliminiamo dalla lista tutti i suoi multipli (i numeri pari). Possiamo sbarrarli, oppure (quel che faremo in seguito nel programma) sostituirli con uno zero. Il risultato sarà quindi il seguente:

**2** 3 0 5 0 7 0 9 0 11 0 13 0 15 0 17 0 19 0 21 0 23 0 25 0 27
0 29 0 31 0 33 0 35 0 37 0 39 0 41 0 43 0 45 0 47 0 49 0

Il primo numero che incontriamo dopo il 2 è il 3: evidenziamolo ed eliminiamo dalla lista tutti i numeri multipli di 3 rimasti (sostituendoli con zero); otterremo:
**2 3** 0 5 0 7 0 0 0 11 0 13 0 0 0 17 0 19 0 0 0 23 0 25 0 0
0 29 0 31 0 0 0 35 0 37 0 0 0 41 0 43 0 0 0 47 0 49 0

Il primo numero (non nullo) dopo il 3 è il 5: evidenziamolo e cancelliamo al solito modo i residui multipli di 5:
**2 3** 0 **5** 0 7 0 0 0 11 0 13 0 0 0 17 0 19 0 0 0 23 0 0 0 0
0 29 0 31 0 0 0 0 0 37 0 0 0 41 0 43 0 0 0 47 0 49 0

Facciamo lo stesso per il numero 7: da eliminare è rimasto il solo numero 49. A questo punto possiamo fermarci: tutti i numeri non nulli rimasti dopo il 7 sono necessariamente primi, e ci basterà raccoglierli (evidenziarli):
**2 3** 0 **5** 0 **7** 0 0 0 **11** 0 **13** 0 0 0 **17** 0 **19** 0 0 0 **23** 0 0 0 0
0 **29** 0 **31** 0 0 0 0 0 **37** 0 0 0 **41** 0 **43** 0 0 0 **47** 0 0 0

Infatti basta ripetere il procedimento fino a che non si supera la radice di $N$. Per capire perché basta pensare che un numero esce dalla lista appena si applica il setaccio relativo al più piccolo dei suoi divisori primi. Ne consegue che ogni numero $a$ non primo e non ancora eliminato avrebbe due divisori entrambi maggiori di $\sqrt{N}$, per cui si avrebbe $a = p \times q > \sqrt{N} \times \sqrt{N} = N$, e sarebbe quindi più grande di tutti i numeri della nostra lista.

Elementi per il programma (a voi il compito di tradurlo in un listato C++):

1. si riempie un vettore di interi $v[N + 1]$ con $v[i] = i$, $i = 1, .., N$;
2. si inizializza un contatore $cont = 0$;
3. per $i = 2$ fino a $i < \sqrt{N}$:
   - se $v[i] \neq 0 \Rightarrow v[i]$ è primo, lo scrivo, $cont + +$,
     e pongo $v[j] = 0$ per ogni $j = k * i$, $k = 2, ..., N$
4. per $i > \sqrt{N}$ fino a $i = N$:
   - se $v[i] \neq 0 \Rightarrow v[i]$ è primo, lo scrivo, $cont + +$
5. il numero complessivo dei primi trovati è $cont$.

## 7.3 Scomposizione in fattori primi

In questo esercizio vogliamo, dato un numero intero $n$, calcolare la sua scomposizione in fattori primi, riproducendo l'algoritmo elementare. Si inizia provando a dividere il numero per 2, e se è divisibile si ripete partendo dal quoziente e contando quante volte la cosa ha successo (quello sarà l'esponente di 2 nella scomposizione). Poi si fa la stessa cosa per 3, poi per 5, e via via per tuti i numeri primi successivi. A meno di non generare in parallelo i numeri primi, per esempio col crivello di Eratostene appena esaminato, ci accontentiamo di provare a dividere $n$ per tutti i numeri da 2 in poi, fino a che il quoziente della divisione non diventi 1. Ecco un possibile listato:

```
main ()
{int i,n, cont;
 printf("Inserire n>1: n= "); scanf("%d",&n);
 i=1;
while (n>1)
     {i++; cont=0;
```

```
        while (n%i==0)
            {n=n/i; cont++;}
        if (cont>0)
            printf(" %d",i);
        if (cont>1)
            printf("^%d",cont);}
printf("\n");
return 0;}
```

Il ciclo *while* più interno verifica la divisibilità per $i$ e in caso affermativo riduce $n$ al quoziente $n/i$ e ripete. Alla fine se *cont* sarà positivo, $i$ sarà un fattore, necessariamente primo, della scomposizione da scrivere. Se poi $cont > 1$, *cont* sarà l'esponente a cui elevare $i$ nella scomposizione. Ovviamente *cont* va resettato (azzerato di nuovo) appena incrementiamo $i$.

# 8 Massimo comun divisore e minimo comune multiplo

Le note formule apprese al liceo per il calcolo di MCD e mcm di due numeri interi $a$ e $b$ si basano sulla preventiva scomposizione di entrambi i numeri in fattori primi: nel primo caso si parlava di *prendere tutti i fattori comuni col minimo esponente*, nel secondo di *prendere tutti i fattori comuni e non comuni col massimo esponente*. Questo approccio non è in generale conveniente perché costoso. Vediamo come possiamo calcolare il MCD senza scomporre i due numeri, e come questo ci fornisca facilmente anche il mcm.

**Algoritmo 1 per il MCD**. (*Sottrazioni successive*) Osserviamo che se i due numeri $a$ e $b$ (con per esempio $0 < a < b$) sono entrambi divisibili per un intero $d$, lo sarà anche la loro differenza $b - a$. Allora:

$$MCD(b, a) = MCD(b - a, a)$$

e possiamo ripetere il ragionamento a partire da numeri più piccoli. Vediamo un esempio esplicito in cui si ripete formalmente il passo precedente finché uno dei due numeri diventa zero:

$$MCD(60, 18) = MCD(42, 18) = MCD(24, 18) = MCD(18, 6) =$$

$$= MCD(12, 6) = MCD(6, 6) = MCD(6, 0)$$

Se assumiamo per convenzione che $MCD(n, 0) = n$, troviamo quindi che il MCD tra 60 e 18 è in effetti pari a 6. Il corpo dell'algoritmo in C++ è il seguente:

```
do
    {r=b-a;
     if (r!=0)
         if (r<a) {b=a; a=r;}
         else {b=r;}
     else
         mcd=a;}
while (r>0);
```

**Algoritmo 2 per il MCD**. (*Divisioni successive*) Osserviamo che l'Algoritmo 1 potrebbe essere molto lento: se i due numeri fossero ad esempio 900 e 15, avrebbe bisogno di ben 60 passaggi per concludere che 15 è proprio il loro MCD, come era subito determinabile dal fatto che si tratta di un divisore di 900. Seguiamo quindi un'altra

strada. Si vede infatti che se i due numeri $a$ e $b$ (sempre con $0 < a < b$) sono entrambi divisibili per un intero $d$, lo sarà anche il resto $r$ della divisione di $b$ per $a$:

$$b = a * q + r,$$

con $0 \leq r < a$. In particolare, se $r = 0$ allora $MCD(b, a) = a$, altrimenti $MCD(b, a) = MCD(a, r)$ e possiamo ripetere il ragionamento a partire da numeri più piccoli finché non si perviene anche in questo caso alla situazione $MCD(s, 0) = s$. Per esempio l'esempio precedente darebbe in questo caso:

$$MCD(60, 18) = (18, 6) = (6, 0) = 6.$$

Ora il corpo del programma C++ potrebbe essere:

```
 do
    {r=b%a;
     if (r!=0)
        {b=a; a=r;}
     else
        mcd=a;}
while (r>0);
```

**Algoritmo per il** *mcm*. Osserviamo che vale la seguente formula:

$$mcm(a, b) = (a * b)/MCD(a, b)$$

Infatti se poniamo $d = MCD(a, b)$, allora dovrà essere $a = q * d$, $b = p * d$, e necessariamente $p$ e $q$ dovranno essere primi tra loro. Allora

$$a * b/d = p * q * d$$

cioè la formula fornisce un multiplo di entrambi i numeri, che è necessariamente il minimo proprio perché $p, q$ sono primi tra loro.

**Il** *mcm* **di più numeri**. Per calcolare il *mcm* di un insieme di numeri interi $n_1, n_2, n_3$, ..., $n_M$ (come quando cerchiamo il denominatore comune in una somma di frazioni) basterà ripetere l'algoritmo precedente a due numeri per volta: trovato $a_1 = mcm(n_1, n_2)$ calcoleremo $a_2 = mcm(a_1, n_3)$ e così via fino ad $a_{M-1} = mcm(a_{M-2}, n_M)$, che sarà il valore cercato.

# 9 I numeri perfetti

**Definizione**. Un numero si dice *perfetto* quando è uguale alla somma dei suoi divisori propri.

Ad esempio, è perfetto il numero 6 che è divisibile per 1, 2 e 3 , perché $6 = 1+2+3$; analogamente lo è il numero 28, divisibile per 1, 2, 4, 7, 14 in quanto

$$28 = 1 + 2 + 4 + 7 + 14\ .$$

I numeri perfetti furono inizialmente studiati dai pitagorici. Un teorema enunciato da Pitagora e dimostrato da Euclide rivelò che se $2^{n+1} - 1$ è un numero primo (i numeri di questa forma si dicono *numeri di Mersenne*), allora $2^n(2^{n+1} - 1)$ è perfetto. Successivamente Eulero dimostrò che tutti i numeri perfetti pari devono essere di tale forma. Ad esempio:

$$6 = 2^1(2^2 - 1), \quad 28 = 2^2(2^3 - 1)\ .$$

I primi 10 numeri perfetti sono:

- 6
- 28
- 496
- 8128
- 33 550 336 (8 cifre)
- 8 589 869 056 (10 cifre)
- 137 438 691 328 (12 cifre)
- 2 305 843 008 139 952 128 (19 cifre)
- 2 658 455 991 569 831 744 654 692 615 953 842 176 (37 cifre)
- 191 561 942 608 236 107 294 793 378 084 303 638 130 997 321 548 169 216 (54 cifre)

L'undicesimo numero perfetto è composto da 65 cifre, il dodicesimo da 77 e il tredicesimo da ben 314 cifre. Si conoscono solo 47 primi di Mersenne, e quindi 47 numeri perfetti. Il più grande tra questi è $2^{43,112,608} \times (2^{43,112,609} - 1)$, formato in base 10 da 25.956.377 cifre.

Non si sa se i numeri perfetti continuino all'infinito né se esistono numeri perfetti dispari, però tutti i numeri perfetti pari terminano con 6 oppure con 8. Infatti da $2^n(2^{n+1} - 1)$ si ha che $2^n$ è pari e termina per 2, 4, 8, 6 mentre $(2^{n+1} - 1)$ è dispari e termina per 3, 7, 5, 1. Il valore '5' va scartato in quanto cadrebbe l'ipotesi di primalità, quindi le coppie che rimangono sono (2,3), (4,7) e (6,1), i cui prodotti danno i numeri 6 ed 8 finali di ogni numero perfetto.

Se la somma dei divisori è maggiore del numero, questo si dice *abbondante*, se risulta minore, verrà chiamato *difettivo*.

Benché esistano infiniti numeri lievemente difettivi, cioè difettivi solo per un'unità, ad esempio 4, i cui divisori sono 1 e 2, la cui somma è uguale a 3, nessuno è ancora riuscito a trovare numeri lievemente abbondanti. Per concludere questo paragrafo diamo un esempio di programma in grado di calcolare tutti i numeri perfetti inferiori o uguali ad un numero $n$ assegnato.

```
main()
{int n,i,j,x;
printf("Inserisci il numero intero n= "); scanf("%d",&n);
for (j=2;j<=n;j++)
{x=1;
for (i=2;i<j/2;i++)
if (j%i==0) x=x+i;
if (x==j)
   printf("Il numero %d e' perfetto \n",j);}
return (0);}
```

# 10 Fattoriale e coefficiente binomiale

Il fattoriale di un numero naturale $n$ è indicato dalla formula

$$n! = 1 * 2 * 3 * ... * (n - 1) * n$$

e in calcolo combinatorio esprime il numero delle possibili permutazioni di $n$ elementi. Può essere facilmente calcolato attraverso una produttoria come indicato nel paragrafo 5:

```
int fatt=1;
for (int i=2; i<=n; i++)
   fatt=fatt*i;
```

Osserviamo che la cosa funziona anche per $n = 0$ e $n = 1$: in tali casi infatti il ciclo $for$ non viene mai eseguito, per cui la variabile $fatt$ rimane al valore 1 iniziale, corrispondente sia a 0! che a 1!. Tutto semplice? No, perchè il fattoriale cresce rapidamente, superando presto il massimo numero intero rappresentabile con 4 bytes di memoria (*integer overflow*). Come sappiamo tale overflow non viene segnalato, ma i numeri generati sono errati (a volte persino negativi). Una soluzione è quella di usare variabili di tipo reale: ecco ad esempio cosa produrrebbe un programma che mettesse a confronto i valori di $n!$ calcolati usando rispettivamente variabili *int* (fat) e *double* (fatt) fino a $n = 18$:

```
il fattoriale di  2 e' fat= 2,  fatt= 2
il fattoriale di  3 e' fat= 6,  fatt= 6
il fattoriale di  4 e' fat= 24,  fatt= 24
il fattoriale di  5 e' fat= 120,  fatt= 120
il fattoriale di  6 e' fat= 720,  fatt= 720
il fattoriale di  7 e' fat= 5040,  fatt= 5040
il fattoriale di  8 e' fat= 40320,  fatt= 40320
il fattoriale di  9 e' fat= 362880,  fatt= 362880
il fattoriale di 10 e' fat= 3628800,  fatt= 3628800
il fattoriale di 11 e' fat= 39916800,  fatt= 39916800
il fattoriale di 12 e' fat= 479001600,  fatt= 479001600
il fattoriale di 13 e' fat= 1932053504,  fatt= 6227020800
il fattoriale di 14 e' fat= 1278945280,  fatt= 87178291200
il fattoriale di 15 e' fat= 2004310016,  fatt= 1307674368000
il fattoriale di 16 e' fat= 2004189184,  fatt= 20922789888000
il fattoriale di 17 e' fat= -288522240,  fatt= 355687428096000
il fattoriale di 18 e' fat= -898433024,  fatt= 6402373705728000
```

Si vede che fino a $n = 12$ i valori delle due colonne coincidono. Dopo solo i valori della colonna di destra (in *double*) restano corretti, mentre quelli della prima sono totalmente sbagliati. Resta però un limite invalicabile, quello delle cifre significative: anche in double quelle a disposizione dopo un po' non saranno più sufficienti per rappresentare esattamente i numeri ottenuti. Oltre, grazie alla notazione esponenziale, potremo solo determinarne l'ordine di grandezza.

Molto diverso è il caso invece del coefficiente binomiale, la cui formula è la seguente:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

e che in calcolo combinatorio esprime il numero di insiemi di $k$ elementi scelti in un insieme di $n$ elementi. Ad esempio dato l'insieme $A = \{a, b, c, d, e\}$ formato da 5 elementi i suoi sottoinsiemi distinti di due elementi (le coppie) saranno $5!/(2! * 3!) = 10$:

$$(a, b),\ (a, c),\ (a, d),\ (a, e),\ (b, c),\ (b, d),\ (b, e),\ (c, d),\ (c, e),\ (d, e).$$

Il binomiale entra ad esempio nella nota formula di Newton sulla potenza $n$-ma di un binomio:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k\, b^{n-k},$$

dove, com'è noto, i coefficienti possono essere generati ricorsivamente dalle formule

$$\binom{n}{0} = 1, \binom{n}{n} = 1, \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

formando il famoso Triangolo di Tartaglia, le cui prime righe sono

```
            1   1
         1     2    1
       1    3     3    1
     1    4     6    4    1
   1    5    10    10   5    1   ecc.
```

In base alla definizione, si potrebbe pensare di calcolare il binomiale di $n$ su $k$ calcolando tre fattoriali mediante l'algoritmo visto prima e facendo poi il quoziente. Osserviamo però che a differenza dei fattoriali, i coefficienti binomiali sono numeri interi che crescono molto lentamente con $n$. È quindi poco sensato passare attraverso i fattoriali per il loro calcolo. Per trovare una strada più efficiente torniamo quindi alla formula iniziale e semplifichiamo $n!$ con $k!$:

$$\frac{n!}{k!(n-k)!} = \frac{(k+1)*(k+2)*..*(k+(n-k))}{1*2*..*(n-k)}$$

L'espressione a secondo membro è ora il quoziente di due prodotti di $(n-k)$ fattori ciascuno o, equivalentemente, il prodotto di $(n-k)$ quozienti, cioè possiamo riscrivere:

$$\binom{n}{k} = \frac{k+1}{1} * \frac{k+2}{2} * .. * \frac{k+(n-k)}{n-k} = \prod_{i=1}^{n-k} \frac{k+i}{i}\ .$$

Questa formula, matematicamente equivalente alla prima, oltre a ridurre il numero complessivo di prodotti necessario al calcolo del binomiale, non corre il rischio di incorrere nell'integer overflow, in quanto divide prima di moltiplicare. Inoltre, come ogni produttoria, è facilmente implementabile attraverso un unico ciclo. Un'altra semplificazione si ottiene ricordando che dalla definizione

$$\binom{n}{k} = \binom{n}{n-k},$$

per cui conviene semplificare per il maggiore tra $k$ e $n-k$, in modo da ottenere il minor numero di fattori nella produttoria. In sintesi quindi un programma C++ per calcolare il binomiale potrebbe essere:

```
#include <stdio.h>
main()
{int i,n,k,s; float binom=1.;
 printf("Inserisci il numero n\n");  scanf("%d",&n);
 if(n<0)  {printf("errore di input: n e' negativo\n"); return 0;}
 printf("Inserisci il numero k<=n\n");  scanf("%d",&k);
 if((k>n)||(k<0)) {printf("errore di input: k non e' compreso tra 0 e n\n"); return 0;}
 s=k;
```

```
if (k<n-k) {k=n-k;}
for (i=1;i<=n-k;i++)
    binom=binom*(k+i)/i;
printf("il binomiale di %d su %d e' %.0f\n",n,s,binom);
return 0;}
```

**Note**. Attenzione al fatto che la variabile *binom* è assunta come $float$! Infatti, anche se il risultato finale della produttoria sarà necessariamente un numero intero, non lo saranno i singoli quozienti che la compongono. Inoltre si osservi che nel ciclo $for$ non viene effettuata la divisione tra interi $(k+i)/i$, perché le operazioni sono eseguite da sinistra a destra: prima viene calcolato il prodotto $binom * (k+i)$ (che quindi sarà $float$), e poi il quoziente del risultato per $i$. Se avvenisse il contrario, il risultato sarebbe errato (provare!). La variabile $s$ serve solo a memorizzare il valore di $k$ introdotto (nel caso fosse più tardi sostituito da $n-k$) ai fini della stampa finale, dove il formato del binomiale è indicato come $\%.0f$ affinché non vi compaiano cifre decimali.

Concludiamo con un'osservazione sulla complessità dei due algoritmi esaminati, in termini di numero di operazioni necessarie per calcolare $\left(\begin{array}{c} n \\ k \end{array}\right)$. Usando i fattoriali effettueremmo $n+k+(n-k)+1 = 2n+1$ prodotti e un quoziente, in totale $2n+2$ operazioni, mentre con la semplificazione operata avremo solo $\min(k, n-k)$ prodotti e altrettanti quozienti, per un numero di operazioni inferiore a $n$.

**Esercizio.** Ci chiediamo quale sia il numero di tutti i sottoinsiemi di un insieme $A$ composto da $n$ elementi, inclusi quelli banali, cioè l'insieme vuoto e l'insieme stesso (l'insieme di tali sottoinsiemi è chiamato anche *insieme delle parti* di $A$). La risposta è facile: basterà sommare il numero di quelli formati da $k$ elementi, per $k$ che va da 0 ad $n$, cioè in altre parole sommare tutti i binomiali relativi ad $n$. Provate allora a scrivere un programma che faccia questo lavoro usando l'algoritmo precedente. Qualcuno potrebbe obiettare che è inutile fare tutta questa fatica, visto che dalla già citata formula del binomio di Newton si ottiene subito:

$$\sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array}\right) = (1+1)^n = 2^n,$$

ma voi dovete esercitarvi a programmare, per cui male non vi fa. Tutt'al più alla fine confronterete i risultati.

# 11 I cambiamenti di base

Nei nostri conti noi usiamo comunemente numeri nella base dieci: nella notazione posizionale le cifre (tra 0 e 9) del numero indicano di quante unità, decine, centinaia, migliaia, ecc. esso è composto. Ad esempio

$$12.457 = 1 \times 10^4 + 2 \times 10^3 + 4 \times 10^2 + 5 \times 10 + 7.$$

In un computer abbiamo invece visto come sia necessario utilizzare la base 2. In generale un numero intero rappresentato in una base $b$ dalle cifre $a_n a_{n-1} ... a_1 a_0$ (dove per ogni $i$: $0 \leq a_i < b$) sarà esprimibile nella base 10 da

$$(a_n a_{n-1} ... a_1 a_0)_b = a_n \times b^n + a_{n-1} \times b^{n-1} + ... + a_1 \times b + a_0.$$

Ad esempio:

$(110101)_2 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 = 53; \quad (1024)_5 = 1 \times 5^3 + 2 \times 5 + 4 = 139.$

Vogliamo ricavare degli algoritmi veloci per la conversione di un numero intero da una base ad un'altra. Per farlo esaminiamo intanto separatamente come passare dalla base 10 ad una base $b$ e viceversa.

## 11.1 Dalla base 10 alla base $b$

L'idea è semplice. Detto $n$ il numero in base 10, dividiamolo successivamente per la nuova base $b$ fino a quando il quoziente non diventa 0: i resti $r_i$ di queste divisioni (numeri quindi compresi tra 0 e $b-1$), scritti in ordine inverso ci daranno la rappresentazione di $n$ in base $b$.

ESEMPIO: convertiamo il numero 23 in base 2

$$23 = 2 \times 11 + \mathbf{1}; \quad 11 = 2 \times 5 + \mathbf{1}; \quad 5 = 2 \times 2 + \mathbf{1}; \quad 2 = 2 \times 1 + \mathbf{0} \quad 1 = 0 \times 2 + \mathbf{1}$$

Quindi: $(23)_{10} = (10111)_2$.

Il corpo del programma in C++ potrebbe essere:

```
ind=9;
while (n>0)
    {a[ind]=n%b; n=n/b; ind=ind-1;}
printf("\n Il numero trasformato in base %d e': \n\n ",b);
for (i=ind+1;i<10;i++)
    printf("%d",a[i]);
```

## 11.2 Dalla base $b$ alla base 10

Occupiamoci ora del problema inverso. Conoscendo le cifre di un numero intero $x$ espresso in una base $b$, vogliamo sapere quale sia la sua rappresentazione in base 10. Qui è sufficiente ricorrere alla notazione posizionale già vista. L'importante è leggere $x$ cifra per cifra (altrimenti il compilatore lo interpreterebbe come un unico numero in base 10). Una volta memorizzate tali cifre nel vettore $(a[0], a[1], ..., a[n])$ (con l'accortezza di memorizzarli in ordine inverso rispetto a quello in cui si presentano), basterà porre $num = 0$ e poi ripetere per $i = 0$ fino a $i = n$:

$$num = num + a[i] \times b^i$$

per ottenere nella variabile $num$ il valore di $x$ in base 10.

## 11.3 Dalla base $b_1$ alla base $b_2$

Il problema generale sarà quindi risolto in due tappe:
1) letto il numero $x$ nella sua rappresentazione in base $b_1$, questo verrà convertito in base 10;
2) la rappresentazione di $x$ in base 10 sarà convertita in quella in base $b_2$.

## 11.4 Conversione dei numeri decimali

Per i numeri decimali il problema è più complesso. La conversione dalla base $b$ alla base 10 è facilmente generalizzabile. Ad esempio il numero $(10111, 101)_2$ corrisponderà in base 10 al numero

$$2^4 + 2^2 + 2 + 2^0 + 2^{-1} + 2^{-3} = 16 + 4 + 2 + 1 + 0,5 + 0,125 = 23,625.$$

Il viceversa crea invece qualche problema, perché un numero può avere parte decimale finita in una base e infinita in un'altra. Ad esempio, il numero $1/3 = 0,\bar{3}$, periodico in base 10, corrisponde al più essenziale numero $(0,1)_3$ nella base 3. La conversione va fatta separatamente per la parte intera e per quella frazionaria del numero. La prima viene convertita come già visto mediante divisioni intere successive per la base $b$ raccogliendo i resti. La parte frazionaria va invece moltiplicata successivamente per la base $b$ raccogliendo le parti intere che si dovessero formare, fermandoci eventualmente quando il prodotto si annulla. Diamo due esempi:

1) Riportiamo il numero $(23,625)_{10}$ nella base 2. Come già visto $(23)_{10} = (10111)_2$. Per la parte decimale:

```
0,625 x 2 = 1,250 (riporto 1)
0,250 x 2 = 0,500 (riporto 0)
0,500 x 2 = 1,000 (riporto 1)
0,000 x 2 =  0,000 STOP
```

Allora il numero convertito sarà: $(10111,101)_2$, come già sapevamo.

2) Convertiamo stavolta nella base 2 il numero $(23,71)_{10}$. Questa volta la procedura non avrebbe termine. Controllate che $(0,71)_{10}$ fornisce con la procedura precedente per le prime otto cifre decimali il numero $(0,10110101)_2$, e quindi la nostra conversione potrà solo essere approssimata.

**Esercizio.** Scrivere un programma che generalizza quello della sezione 11.1 al caso di numeri decimali.

# 12 Le successioni definite per ricorrenza

In molte situazioni, per descrivere soprattutto eventi che si ripetono a tempi discreti e la cui evoluzione dipende dagli stati precedenti, è naturale considerare successioni definite in modo ricorsivo, assegnando uno o più stati iniziali e la legge che determina ogni nuovo stato in funzione dei precedenti. Nel caso più semplice di dipendenza dal solo stato precedente potremo scrivere in simboli

$$\begin{cases} x_{n+1} = f(x_n)\,, \\ x_0 = \alpha \end{cases}$$

dove $\alpha$ indica lo stato iniziale del sistema, mentre la funzione $f(x)$ descrive la sua dinamica. Ad esempio la successione definita da $x_{n+1} = 3x_n + 1,\ x_0 = 1$, genererà la sequenza: $1,\ 4,\ 13,\ 40,\ 121,\ 364, \ldots..$

Studiare una successione di questo tipo è in genere molto più difficile di quando si disponga di una formula esplicita per il termine $n$-mo. D'altra parte, specie per un computer, è invece molto semplice generare l'evoluzione di un simile sistema. Il primo problema è però quello di dimostrare che la successione sia ben definita: per esempio se la funzione $f$ contenesse una radice quadrata, si dovrebbe controllare che al crescere di $n$ il radicando non possa mai diventare negativo. E' chiaro che una simile verifica non può essere fatta controllando tutti i termini, ed è qui che può aiutarci il principio di induzione (si veda più avanti l'Esempio 1).

Se poi vogliamo studiare il comportamento della successione, di nuovo dobbiamo ricorrere a ragionamenti induttivi. Riguardo al limite, supponendo che la funzione $f$ sia continua, si può ragionare così: se la successione ammette un limite finito $L$, allora per $n \to \infty$ nella relazione costitutiva si otterrebbe $L = f(L)$, per cui $L$ deve necessariamente essere tra i punti fissi della funzione $f$, cioè tra i valori che $f$ manda in sé stessi; ma l'esistenza di uno o più punti di questo tipo non dimostra ancora nulla (si

veda l'Esempio 2). Potrebbe succedere ad esempio che cambiando il solo dato iniziale $\alpha$, a parità di $f$, la nostra successione possa cambiare il valore del proprio limite o addirittura smettere di essere convergente. Dovremo aiutarci col ragionamento e con alcuni risultati noti dell'analisi, come ad esempio quello che ci dice che una successione monotona ammette limite, finito, se limitata, o infinito, altrimenti, e di nuovo ci servirà il principio di induzione. Una trattazione rigorosa di queste successioni esula dagli scopi di questo corso e richiederebbe strumenti ancora non disponibili al primo anno di matematica.

Qui ci limiteremo ad analizzare in superficie un paio di successioni notevoli (la logistica e quella di Fibonacci) tanto per mostrare il tipo di questioni, spesso assai complesse, che possono nascere da questi modelli discreti. Nella prossima sezione vedremo altri esempi di successioni ricorsive utilizzate per il calcolo approssimato di numeri speciali come $\pi$ o $\sqrt{2}$.

**Esempio 1** *Verificare che la successione*

$$\begin{cases} x_{n+1} = \sqrt{x_n - 1} \\ x_0 = 5 \end{cases}$$

*non è ben definita per ogni $n \in \mathbb{N}$, mentre lo è la successione*

$$\begin{cases} x_{n+1} = \sqrt{2 + x_n} \\ x_0 = 0 \end{cases} .$$

Calcoliamo i primi termini della prima successione:

$$x_1 = 2, \ x_2 = 1, \ x_3 = 0, \ x_4 = \sqrt{-1}!$$

e non possiamo più andare avanti. Nel secondo caso invece si vede facilmente che la quantità sotto radice resterà sempre non negativa. È il principio di induzione ad aiutarci:

$$i)\ 2 + x_0 > 0; \quad ii)\ \text{se } 2 + x_n > 0 \text{ allora } 2 + x_{n+1} = 2 + \sqrt{2 + x_n} > 2 > 0 .$$

**Esempio 2** *Si consideri la successione*

$$\begin{cases} x_{n+1} = 2/x_n \\ x_0 = \alpha \end{cases} .$$

Si vede subito che questa successione è ben definita per ogni $\alpha \neq 0$, ma che in genere non ha limite. Infatti i suoi valori oscillano alternativamente tra $\alpha$ e $2/\alpha$. D'altra parte se cerchiamo soluzioni dell'equazione $L = 2/L$ troviamo i valori $L = \pm\sqrt{2}$. Corrispondono in effetti al limite solo nel caso in cui sia già $\alpha = \pm\sqrt{2}$, quando si otterrebbero successioni costanti. Tenendo conto che tali valori non sono numeri rappresentabili esattamente su di un calcolatore, possiamo concludere che in pratica questa successione sarà sempre oscillante.

## 12.1 La mappa logistica

Uno dei più noti modelli discreti di crescita di popolazioni in campo biologico è quello legato alla cosiddetta *mappa logistica*, cioè la seguente successione:

$$\begin{cases} x_{n+1} = \alpha x_n (1 - x_n) \\ x_0 > 0 \text{ dato} \end{cases} .$$

dove $x_n$ indica la densità della popolazione sotto osservazione al tempo $t_n$, $x_0$ la densità iniziale, ed $\alpha$ il tasso di crescita nel tempo.

Questa legge ha una lunga storia: fu proposta nel 1845 da P.F. Verhulst come modello matematico per lo studio dei fenomeni di crescita delle popolazioni in situazioni di risorse limitate. Se immaginiamo dunque che $x_n$ indichi la densità della popolazione considerata (quindi $x_n \in [0, 1]$, dove il valore 0 corrisponde all'estinzione, mentre 1 alla densità massima consentita dall'ambiente), osserviamo che la specie crescerà infatti in modo proporzionale alla sua stessa densità, ma la crescita verrà frenata dal quadrato di questa. Se all'inizio gli individui si distribuiscono in un ambiente vasto con sufficienti risorse per tutti, l'interazione tra gli individui è scarsa, $x_n$ è molto piccola e quindi il termine quadratico sarà in pratica trascurabile: la crescita (ad un tasso riproduttivo $\alpha > 1$) sarà di tipo esponenziale. Ma appena gli individui arriveranno a contendersi le risorse ($x_n$ si avvicina a 1), il termine quadratico diventerà sempre più importante e frenerà di conseguenza la crescita. È il tipico andamento a *esse* detto appunto *curva logistica* (v. Figura 1). È comunque interessante capire quale sarà l'evoluzione effettiva del sistema in funzione del parametro $\alpha$, se cioè la popolazione si estinguerà ($x_n \to 0$), se raggiungerà un'altro equilibrio non banale ($x_n \to L$), o se oscillerà tra diversi stati più o meno ricorrenti.

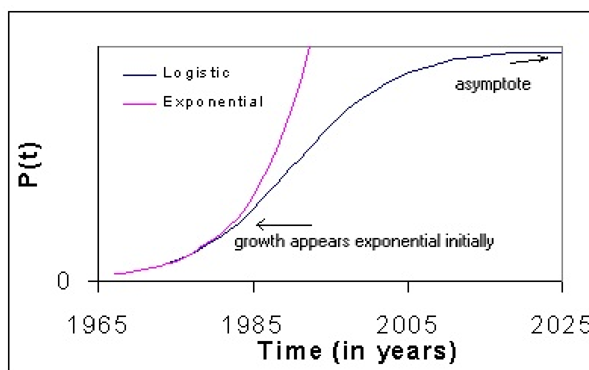


Figura 1: Esempio di crescita logistica

La discussione ci può portare a scoprire aspetti matematici sorprendenti. La cosa notevole è infatti che, nonostante la forma relativamente semplice, il modello presenta fenomeni complessi che richiedono teorie matematiche molto più solide di quelle disponibili nella scuola superiore o ai primi anni di università. Si può mostrare in particolare che variando il parametro $\alpha$ si ottengono dinamiche totalmente differenti, fino a ottenere comportamenti di tipo *caotico*. Qui ci limiteremo solo a studiare qualche situazione semplice (con le tecniche già utilizzate in altri esempi) e apparentemente rassicurante. Assumiamo ad esempio $x_0 = 1/5$ e proviamo a dimostrare le seguenti proprietà:

*(i) la mappa logistica manda* $[0, 1]$ *in* $[0, 1]$ *se e solo se* $\alpha \in [0, 4]$*;*

*(ii) gli unici punti fissi in* $[0, 1]$ *della mappa logistica sono* $p_0 = 0$ *(per ogni* $\alpha$*) e* $p_\alpha = \frac{\alpha-1}{\alpha}$ *(per ogni* $\alpha > 1$*);*

*(iii) se* $\alpha = 1$ *allora* $x_{n+1} \leq x_n$*, e* $x_n \to 0$*;*

*(iv) se* $\alpha = 2$ *allora* $x_{n+1} \geq x_n$*, e* $x_n \to 1/2$*;*

Dal primo punto ricaviamo in particolare che l'intervallo $[0, 4]$ è l'unico insieme in cui far variare $\alpha$ in modo che il modello mantenga un significato biologico (matematicamente potremmo andare anche oltre, ma avremo già abbastanza problemi così...). Gli altri punti ci dicono che per $\alpha = 1$ la specie si estinguerà gradatamente (il tasso riproduttivo è troppo basso), mentre per $\alpha = 2$ la specie raggiungerà rapidamente l'equilibrio perfetto, quello con sé stessa e le risorse a disposizione, senza fluttuazioni di

sorta. È la situazione del paradiso terrestre, ma come vedremo il peccato originale è anche in questo caso in agguato....

Cominciamo a esaminare le questioni andando per ordine.

(i) Se $x \in [0,1] \Rightarrow x - x^2 \geq 0$, e quindi occorre che $\alpha$ sia positivo. D'altra parte se vogliamo che $x_n \leq 1$ ci serve che il trinomio di secondo grado $\alpha x^2 - \alpha x + 1$ resti non negativo per ogni $x \in [0,1]$, relazione vera solo se il discriminante è negativo o nullo, ovvero per $\alpha \leq 4$.

(ii) Per individuare i punti fissi dovremo risolvere l'equazione algebrica di secondo grado $x = \alpha x(1-x)$, cioè $x(\alpha x + (1-\alpha)) = 0$, da cui la risposta $x_1 = 0$ e $x_2 = (\alpha - 1)/\alpha$. Quindi un equilibrio per la mappa logistica sarà sempre lo zero, per ogni $\alpha$, mentre l'altro dipende da $\alpha$ stesso e apparterrà al nostro insieme solo se $\alpha > 1$ (abbiamo ignorato il caso non interessante in cui $\alpha = 0$).

(iii) Osserviamo subito che $x(1-x) = x - x^2 \leq x$ per qualsiasi valore della $x$, da cui segue che la successione è monotona decrescente, limitata inferiormente da 0, unico punto fisso, quindi convergerà decrescendo a 0 (equilibrio stabile del sistema).

(iv) Quando $\alpha = 2$ è facile provare che $2x(1-x) \geq x$ se e soltanto se $0 < x < 1/2$; inoltre vale anche $2x(1-x) \leq 1/2$ per ogni $x$. Poiché partiamo dal valore $1/5$ e non possiamo andare oltre $1/2$, le iterazioni crescono, come dovevamo dimostrare. Ne segue che la successione convergerà crescendo al numero $1/2 = (2-1)/2$.

Possiamo ora chiederci cosa cambia nel comportamento della successione per i valori di $\alpha$ esaminati se partiamo da un altro valore iniziale, per esempio $x_0 = 2/3$. Non cambia nulla per $\alpha = 1$ (il processo di estinzione sarà solo all'inizio più lento), mentre per $\alpha = 2$ la successione stavolta decrescerà al valore $1/2$, che si conferma equilibrio stabile.

La situazione cambia per valori più grandi di $\alpha$. Non è difficile scrivere un programma C che generi le prime 50 iterazioni della legge logistica per diversi valori di $\alpha$ (si veda l'esercizio in fondo al paragrafo seguente). Considerate ad esempio i valori $\alpha = 1, 2, 3, 3.25, 3.5, 3.75, 4$. Avendo a disposizione un programma per la grafica (ad esempio GNUPLOT) potrete anche visualizzarne l'andamento. Oppure senza sforzo potete andare su alcuni siti web dove potrete interattivamente fare le vostre simulazioni (si veda ad esempio l'elenco più in basso). Dovreste osservare per i primi valori di $\alpha$ quanto già dimostrato, ma poi la situazione cambia rapidamente, e diventa di lettura via via più difficile. E' possibile (ma non è questo il luogo dove farlo) dimostrare che appena $\alpha > 3$ la dinamica cresce di complessità creando comportamenti periodici di periodo arbitrariamente grande, in quanto esistono infinite soglie $\alpha_n$ che mutano il comportamento delle iterate del nostro sistema.

Nella Figura 2 che segue si vede il cosiddetto diagramma di biforcazione della funzione logistica in funzione del parametro $\alpha$ (in ascissa) tra i valori 2.4 e 4. Lo schema evidenzia sostanzialmente i punti di equilibrio e le orbite periodiche del sistema: si può osservare che fino ad $\alpha = 3$ c'è un solo equilibrio, che poi si trasforma in un'orbita 2-periodica (il sistema oscillerà di fatto tra due stati differenti). Aumentando ancora $\alpha$ il periodo raddoppierà ancora (periodo 4), e poi ancora (periodi 8, 16, 32,...) fino ad un vero e proprio comportamento caotico (con delle strane eccezioni, le zone chiare della mappa corrispondenti ad un comportamento più regolare ...).

Per concludere, anche se il modello logistico non nasce certo per simulare il comportamento della specie umana, possiamo concludere con una breve morale: tassi riproduttivi troppo deboli conducono all'estinzione, tassi troppo elevati al caos. Per una società tranquilla e senza scosse (ma anche un tantino noiosa...) occorre un tasso riproduttivo moderato.

**Alcuni siti web sulla logistica.**

`http://math.la.asu.edu/~chaos/logistic.html`

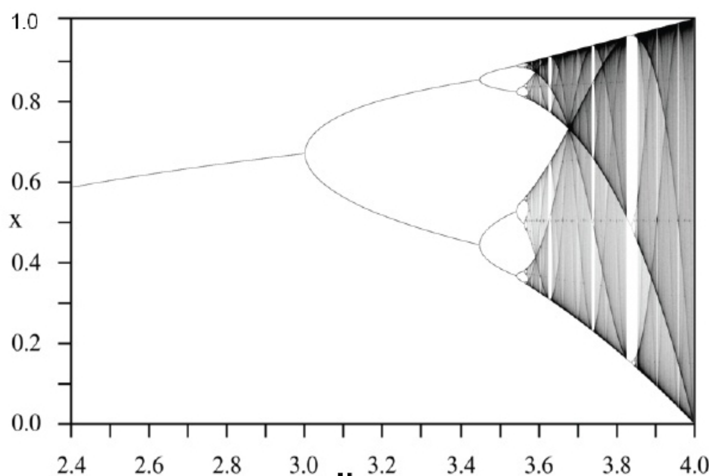


Figura 2: Diagramma di biforcazione della mappa logistica

http://www.geom.uiuc.edu/~math5337/ds/applets/iteration/Iteration.html
http://brain.cc.kogakuin.ac.jp/~kanamaru/Chaos/e/Logits/
http://math.bu.edu/DYSYS/applets/bif-dgm/Logistic.html

## 12.2 La mappa di Fibonacci

La nota successione

$$\begin{cases} a_{n+2} = a_n + a_{n+1} \\ a_1 = a_2 = 1 \end{cases}$$

fu introdotta da Leonardo Pisano (detto il Fibonacci) intorno al 1200 per risolvere il seguente problema:

*Sapendo che una coppia di conigli adulti genera una coppia di figli al mese, e questi diventano adulti in due mesi generando a loro volta un'altra coppia di conigli, quante coppie ci saranno dopo n mesi se si inizia un allevamento con una coppia di conigli adulti ?*

Si tratta chiaramente di una successione definita per ricorrenza, dove ogni elemento è la somma dei due che lo precedono. In questo caso non è difficile dimostrare che la successione tende all'infinito. Calcoliamo intanto i primi 10 termini della successione:

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ ....$$

Dopo un anno avremo già un bell'allevamento (per questo si dice crescere come conigli...). Analizziamo bene i termini della successione per avere l'evidenza che senza nostro intervento questa tenderà all'infinito. I termini sono ovviamente tutti positivi (lo sono i primi due, e se lo sono tutti quelli fino all'indice $n$, lo sarà anche il termine $(n+1)$-mo. Segue subito che la successione è crescente. La successione avrà quindi limite, finito o infinito. Ma l'unico limite finito possibile sarebbe zero, soluzione di $L = 2L$, e quindi la successione diverge.

Il problema più interessante è però quello di determinare il *tasso di accrescimento* dell'allevamento, cioè di quanto presumibilmente crescerà ogni mese a regime il numero delle coppie presenti. Per farlo occorre studiare la successione $b_n = a_{n+1}/a_n$. È facile vedere che si tratta ancora di una successione definita per ricorrenza, questa volta

a un solo passo:

$$b_{n+1} = \frac{a_{n+2}}{a_{n+1}} = \frac{a_{n+1} + a_n}{a_{n+1}} = 1 + \frac{a_n}{a_{n+1}} = 1 + \frac{1}{b_n} \,, \quad b_1 = 1 \,.$$

Il tasso tendenziale di accrescimento dell'allevamento sarà allora, se esiste, il limite della successione $\{b_n\}$. Proviamo, in un ragionamento a più passi, che tale limite esiste ed è pari al numero

$$b = \frac{1 + \sqrt{5}}{2} = 1.6180339887....(\text{ la } \textit{sezione aurea}!) \;;$$

in altre parole se un mese nell'allevamento ci sono un centinaio di coppie, il mese seguente se ne dovrebbero trovare circa 160. Arriviamo alla conclusione attraverso i seguenti punti

*(i) Mostriamo che se $b_n \to L$, allora $L = (1 \pm \sqrt{5})/2$. Seguirà che l'unico limite possibile, se la successione converge, deve essere il numero b.*
*(ii) La successione non è monotona. Dimostriamo che*

$$b_{n+2} = 1 + \frac{b_n}{1 + b_n} = 2 - \frac{1}{1 + b_n};$$

*da cui segue per induzione che la successione dei termini dispari verifica $b_n < b$.*
*(iii) Deduciamo da (ii) che la successione dei termini di indice dispari è una successione crescente e limitata da b e che essa tende proprio a b.*
*(iv) Con ragionamento analogo si può provare che la successione dei termini di indice pari verifica $b_n > b$ ed è decrescente verso il numero b.*
*(v) Osserviamo che se $b_{2n} \to b$ e $b_{2n+1} \to b$ allora tutta la successione $b_n \to b$.*

(i) Se la successione $b_n$ ammette un limite $L$, questo dovrà necessariamente verificare $L = 1 + 1/L$, cioè $L^2 - L - 1 = 0$, equazione risolta dai valori $(1 \pm \sqrt{5})/2$. Ma dalla definizione di $b_n$ i nostri termini sono tutti positivi, quindi l'unico valore ammissibile sarà $b = 1.6180339887....$
(ii) La successione non è in questo caso monotona, perché si può mostrare che i suoi termini oscillano attorno al valore $b$. Vediamone qualche termine (arrotondato alle sette cifre decimali):

1, 2, 1.5, 1.6666667, 1.6, 1.625, 1.6153846, 1.6190476, 1.6176471, 1.6181818, ...

La relazione tra un termine $b_n$ e il termine $b_{n+2}$ è la seguente:

$$b_{n+2} = 1 + \frac{1}{b_{n+1}} = 1 + \frac{1}{1 + \frac{1}{b_n}} = 1 + \frac{b_n}{1 + b_n} = 2 - \frac{1}{1 + b_n} \;;$$

se ci limitiamo ai soli termini dispari allora, $b_1 = 1 < b$, e se $b_n < b$ (con $n$ dispari) allora $b_{n+2} < 2 - \frac{1}{1+b} = b$ ; quindi tutti i termini dispari sono sotto $b$.
(iii) Mostriamo che la successione dei termini dispari è crescente: serve

$$b_{n+2} = 2 - \frac{1}{1 + b_n} > b_n$$

se $n$ è dispari, che è vero se $b_n^2 - b_n - 1 < 0$, cioè se $b_n < b$, come già dimostrato. Allora la successione dei termini dispari convergerà ad un numero finito che può essere solo $b$ per quanto provato precedentemente.
(iv) La successione dei termini pari decresce verso $b$: la dimostrazione è del tutto simile

alla precedente.
(v) Segue dalla definizione di limite di una successione, visto che questa coincide con l'unione dei termini pari e di quelli dispari.

**ESERCIZIO.** Costruire un programma in C++ in grado di generare i valori di una successione per ricorrenza $x_{n+1} = f(x_n)$ una volta assegnati il valore iniziale $x_0$, la tolleranza richiesta *tol* e il numero massimo di iterazioni $N_{max}$. Il programma calcolerà ad ogni iterazione il termine $\varepsilon_n = |x_n - x_{n-1}|$ che consente di valutare se la successione si stabilizza, e si fermerà appena $\varepsilon_n < tol$ oppure quando $n > N_{max}$.
Applicarlo alle successioni illustrate in queste note, confrontando i risultati con quanto dimostrato teoricamente.

# 13 Tre numeri speciali: $\pi$, $\sqrt{2}$, $e$

## 13.1 Pi greco

Esistono diversi modi per approssimare il numero $\pi$, uno dei quali, noto fin dall'antichità e attribuito ad Archimede, parte dall'osservazione che $\pi$ coincide col valore della lunghezza della semicirconferenza di raggio unitario. Possiamo allora pensare di approssimarlo per difetto attraverso le lunghezze dei semiperimetri dei poligoni regolari inscritti al cerchio unitario e per eccesso attraverso quelle relative ai poligoni circoscritti. L'idea interessante è che queste lunghezze possono essere ricavate in modo ricorsivo partendo da quelle dei quadrati inscritto e circoscritto, raddoppiando ad ogni iterazione il numero dei lati dei poligoni coinvolti. Chiamiamo quindi $l_n$ il lato del poligono regolare inscritto avente $2^n$ lati, e $L^n$ il lato del corrispondente poligono circoscritto. Con un po' di geometria elementare (poco più del Teorema di Pitagora) si ricavano abbastanza facilmente le seguenti relazioni:

$$l_2 = \sqrt{2} \text{ (lato del quadrato inscritto)}, \quad l_{n+1} = \sqrt{2 - \sqrt{4 - l_n^2}}, \; n = 2, 3, ...$$

e

$$L_n = \frac{2l_n}{\sqrt{4 - l_n^2}}.$$

In teoria quindi $2^{n-1} l_n < \pi < 2^{n-1} L_n$, con una indeterminazione che dovrebbe ridursi al crescere di $n$. Nella realtà le cose non vanno così bene, e per questa strada non otterremmo più di 8 cifre corrette di $\pi$ (per $n = 15$, un poligono di circa 32700 lati), poi le due successioni inizierebbero ad oscillare e ad allontanarsi dal valore cercato. Il motivo sta nella particolare formula che abbiamo usato. Come già visto nel paragrafo 4 a proposito della risoluzione delle equazioni di secondo grado, anche qui abbiamo a che fare con una formula instabile. Al crescere di $n$ infatti $l_n$ tende a zero e quindi $\sqrt{4 - l_n^2}$ tende a 2, provocando una cancellazione di cifre significative nella successiva sottrazione tra due numeri quasi uguali. Per fortuna anche qui abbiamo una soluzione alternativa. Basta osservare che vale l'identità

$$\sqrt{2 - \sqrt{4 - l_n^2}} = \frac{l_n}{\sqrt{2 + \sqrt{4 - l_n^2}}}$$

per ottenere una formula ricorsiva stabile (nessuna sottrazione pericolosa stavolta). Usando questa formula si raggiunge ora tranquillamente un risultato corretto di $\pi$ fino alla precisione voluta. Per esempio si ottengono quindici cifre decimali usando un poligono di $2^{26}$, cioè più di 67 milioni di lati. Ecco qui di seguito un possibile listato

per questi calcoli, dove compaiono entrambe le formule (stabile e instabile). Basterà ogni volta commentarne una e scommentare l'altra per vedere le differenze.

```
#define Pi 3.141592653589793
main() {
  int i,n;
  double lato, p_dif, p_ecc;
  printf("Calcola il semi perimetro dei poligoni di 2^n lati per n= \n");
  scanf("%d",&n);
  lato=sqrt(2);
  for (i=3;i<=n;i++) {
     // lato=sqrt(2-sqrt(4-lato*lato));     // formula instabile
     lato=lato/sqrt(2+sqrt(4-lato*lato));   // formula stabile
     p_dif=lato*pow(2,i-1); p_ecc=(p_dif*2)/sqrt(4-lato*lato);
     printf("%1.20lf < pi greco < %1.20lf usando 2^%d lati\n\n",p_dif,p_ecc,i);}
  printf("Valore di pi greco = circa %1.15f\n\n",Pi);
  return 0;}
```

## 13.2 La radice di un numero positivo

Consideriamo ora l'*algoritmo di Erone*, ovvero il seguente sistema di iterazioni

$$\begin{cases} x_{n+1} = \dfrac{1}{2}\left(x_n + \dfrac{2}{x_n}\right) \\ x_0 = 2 \end{cases} .$$

Mostriamo che si tratta di un algoritmo molto efficiente per l'approssimazione della radice quadrata di due (in seguito nel corso lo ritroveremo come caso particolare di un più generale metodo di approssimazione degli zeri di una funzione). Ha la forma di una successione definita per ricorrenza, per la quale, usando solo semplici disuguaglianze algebriche, riusciremo a dimostrare interessanti proprietà di convergenza, oltre a fornire un pratico criterio di arresto in funzione della precisione di calcolo desiderata. Cominciamo dal dimostrare le seguenti relazioni:

(i)

$$x_n > 0\,, \qquad x_n^2 > x_{n+1}^2 > 2\,.$$

(ii)

$$\lim_{n\to\infty} x_n = \sqrt{2}.$$

(iii)

$$x_n - \sqrt{2} < \frac{x_0 - \sqrt{2}}{2^n}\,.$$

(iv)

$$x_{n+1} - \sqrt{2} < x_n - x_{n+1}\,.$$

Con poca fatica possiamo calcolare i primi termini della successione:

$$x_0 = 2, \quad x_1 = 1.5, \quad x_2 = 1.41666666, \quad x_3 = 1.41421569, \quad x_4 = 1.41421356\ldots$$

Ricordando che $\sqrt{2} = 1.414213562373094...$ sembra notarsi una rapida convergenza, con un numero di cifre esatte che appare 'raddoppiare' a ogni iterazione. Ma vediamo di dimostrare rigorosamente questo fatto.

(i) Il fatto che $x_n$ sia positivo è una conseguenza del fatto che $x_0 = 2 > 0$ e che le iterazioni successive sono ottenute come media aritmetica di due quantità positive. Dalla disuguaglianza $(a^2 + b^2) \geq 2ab$ segue poi che

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right) \geq \frac{1}{2} 2 \sqrt{x_n} \frac{\sqrt{2}}{\sqrt{x_n}} = \sqrt{2},$$

che è quanto dovevamo provare. Dobbiamo infine dimostrare che la successione decresce, cioè che $x_n > x_{n+1}$. Quindi che, dalla definizione,

$$x_n > \frac{1}{2}\left(x_n + \frac{2}{x_n}\right) ;$$

si vede abbastanza facilmente che ciò si verifica se e solo se $x_n^2 > 2$, che è quanto avevamo appena dimostrato.

(ii) I precedenti argomenti dovrebbero farci intuire che le iterazioni convergono proprio a $\sqrt{2}$. La successione infatti decresce mantenendosi sempre maggiore di $\sqrt{2}$; questo basta a dimostrare, per il già citato teorema sulle successioni monotone, che converga a un numero $L \geq \sqrt{2}$. Ma $L$ dovrà per forza verificare l'equazione $2L = L + 2/L$, cioè $L^2 = 2$, quindi $L = \sqrt{2}$. Infatti l'algoritmo di Erone fornisce proprio un metodo pratico per approssimare questo numero irrazionale.

(iii) Per ogni $n$ si ha $x_n > \sqrt{2}$ e quindi $2/x_n < \sqrt{2}$. Allora

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right) < \frac{1}{2}(x_n + \sqrt{2}),$$

da cui $x_{n+1} - \sqrt{2} < (x_n - \sqrt{2})/2$. Iterando segue la tesi. Quindi la differenza tra $x_n$ e il valore $\sqrt{2}$ tende a zero al crescere di $n$, e l'errore si dimezza almeno ad ogni passo.

(iv) Ancora dalla relazione vista al punto precedente: $2x_{n+1} < x_n + \sqrt{2}$, da cui , sottraendo a entrambi i membri $(x_{n+1} + \sqrt{2})$ si ottiene la tesi, che afferma in sostanza che se mi fermo dopo $n + 1$ passi, la mia distanza dal valore corretto di $\sqrt{2}$ è minore della differenza $x_n - x_{n+1}$. Abbiamo quindi trovato un ottimo criterio d'arresto per il nostro algoritmo: ci riterremo soddisfatti, e quindi fermeremo le iterazioni, non appena quella differenza sarà diventata più piccola della precisione voluta.

Alla luce di quanto detto, un semplice programma in C per calcolare la radice di 2 con la precisione voluta potrebbe essere il seguente:

```
main()
{int i=1,n, maxit=20; double x, xold, eps;
printf("Inserisci il valore iniziale (maggiore di rad(2)) x0= ");
scanf("%lf",&x);
printf("Inserisci la precisione: eps= ");
scanf("%lf",&eps);
do
   {xold=x; x=(x+2/x)/2; i++;}
while (xold-x>eps && i<maxit);
printf("x(%d)= %1.20lf con err< %e \n",i,x,eps);
return (0);
```

**Osservazione.** I ragionamenti che abbiamo fatto possono essere ripetuti senza grossi cambiamenti per ottenere un algoritmo efficiente per il calcolo di qualunque radice quadrata di un numero positivo $\beta$:

$$\begin{cases} x_{n+1} = \frac{1}{2}\left(x_n + \frac{\beta}{x_n}\right) \\ x_0 = \alpha \end{cases} .$$

dove per $\alpha$ possiamo prendere un qualunque valore per eccesso del numero cercato.

## 13.3 Il numero di Nepero

Si è definito il numero $e$ come il limite della successione monotona crescente

$$a_n = \left(1 + \frac{1}{n}\right)^n .$$

Un conto abbastanza semplice mostra allo stesso modo che $e$ è anche il limite della successione monotona decrescente

$$b_n = \left(1 + \frac{1}{n}\right)^{n+1} .$$

Si potrebbe pensare quindi di utilizzare le due successioni per approssimare il numero $e$ da sotto e da sopra (per difetto e per eccesso) fino ad una precisione desiderata. La cosa purtroppo non funziona affatto. Ecco quello che potremmo ottenere (ricordiamo che $e \simeq 2.718281828459$) calcolando le due successioni sulle potenze successive di 10:

```
Calcolo approssimato di e: (1+1/n)^n< e < (1+1/n)^{n+1}, per n=10^k
n=10^1 x= 2.593742460100   y=2.853116706110   e-x= 0.124539368359
n=10^2 x= 2.704813829422   y=2.731861967716   e-x= 0.013467999037
n=10^3 x= 2.716923932236   y=2.719640856168   e-x= 0.001357896223
n=10^4 x= 2.718145926825   y=2.718417741418   e-x= 0.000135901634
n=10^5 x= 2.718268237192   y=2.718295419875   e-x= 0.000013591267
n=10^6 x= 2.718280469096   y=2.718283187376   e-x= 0.000001359363
n=10^7 x= 2.718281694132   y=2.718281965960   e-x= 0.000000134327
n=10^8 x= 2.718281798347   y=2.718281798347   e-x= 0.000000030112
n=10^9 x= 2.718282052012   y=2.718282052012   e-x= -0.000000223553
n=10^10 x= 2.718282053235   y=2.718282053235   e-x= -0.000000224776
n=10^11 x= 2.718281997687   y=2.718281997687   e-x= -0.000000169228
n=10^12 x= 2.718523484901   y=2.718523484901   e-x= -0.000241656442
n=10^13 x= 2.716109987398   y=2.716109987398   e-x= 0.002171841061
n=10^14 x= 2.716110044314   y=2.716110044314   e-x= 0.002171784145
n=10^15 x= 3.035035162715   y=3.035035162715   e-x= -0.316753334256
n=10^16 x= 1.000000000000   y=1.000000000000   e-x= 1.718281828459
```

Intanto si vede che le due successioni convergono ad $e$ molto lentamente, così che avremmo bisogno di utilizzare valori di $n$ estremamente grandi per il nostro scopo (per $n = 10$ milioni si trovano appena 6 cifre esatte del numero). Ma continuando si osserva che le successioni smettono di essere monotone e separate come la teoria prevede per poi 'convergere' (!) entrambe sul numero 1. Per quanto visto nella sezione 3.2 il motivo dovrebbe esservi chiaro: nelle formule usate si deve sommare il numero 1 al numero sempre più piccolo $1/n$; appena questo diviene minore della precisione macchina la somma restituisce semplicemente 1, che elevato alla $n$ farà sempre 1.

Cerchiamo allora un'altra strada. Si consideri la successione

$$s_n = \sum_{k=0}^{n} a_k ,$$

dove $a_k$ è a sua volta definita per ricorrenza da

$$\begin{cases} a_{k+1} = \frac{a_k}{k+1} \\ a_0 = 1 \end{cases} .$$

La successione $s_n$ rappresenta dunque le somme parziali di una serie di termine $k$-mo $a_k$. Studiamo la serie:
*(i) E' soddisfatta la condizione necessaria di convergenza.*

Si verifica subito che $a_k$ è a termini positivi e decrescente. Inoltre si vede facilmente che la successione $a_k$ è infinitesima. Se cioè ammette limite finito $L$, questo dovrà essere necessariamente zero. In tal caso infatti

$$L = \lim_{k\to\infty} a_{k+1} = \lim_{k\to\infty} \frac{a_k}{k+1} = 0 \ .$$

*(ii) La serie converge.*

Applichiamo il criterio del rapporto per le serie a termini positivi:

$$\lim_{k\to\infty} \frac{a_{k+1}}{a_k} = \lim_{k\to\infty} \frac{1}{k+1} = 0 < 1.$$

Ecco i primi 14 termini di $s_n$ e la differenza col numero $e$:

```
n=1 x= 2.000000000000   e-x= 0.718281828459
n=2 x= 2.500000000000   e-x= 0.218281828459
n=3 x= 2.666666666667   e-x= 0.051615161792
n=4 x= 2.708333333333   e-x= 0.009948495126
n=5 x= 2.716666666667   e-x= 0.001615161792
n=6 x= 2.718055555556   e-x= 0.000226272903
n=7 x= 2.718253968254   e-x= 0.000027860205
n=8 x= 2.718278769841   e-x= 0.000003058618
n=9 x= 2.718281525573   e-x= 0.000000302886
n=10 x= 2.718281801146   e-x= 0.000000027313
n=11 x= 2.718281826198   e-x= 0.000000002261
n=12 x= 2.718281828286   e-x= 0.000000000173
n=13 x= 2.718281828447   e-x= 0.000000000012
n=14 x= 2.718281828458   e-x= 0.000000000001
```

Come si vede questa volta la successione si avvicina rapidamente ad $e$. Si può infatti dimostrare che la somma della serie è proprio il numero di Nepero. D'altra parte la serie potrebbe essere riscritta equivalentemente come

$$\sum_{k=0}^{\infty} \frac{1}{k!} \ ,$$

serie che nel corso di analisi vedrete coincidere con la serie di Taylor della funzione esponenziale $e^x$ calcolata per $x = 1$. Si noti che non conviene fare i calcoli usando quest'ultima formulazione. La definizione per ricorrenza che abbiamo introdotto precedentemente presenta due notevoli vantaggi: fa risparmiare molte operazioni, ed evita il rapido overflow causato dai termini fattoriali.

## 14 Ordinamento di un vettore

In queste note non abbiamo finora avuto bisogno di ricorrere agli arrays per scrivere i nostri programmi, e non vogliamo nemmeno qui iniziare a trattare l'argomento in dettaglio. Ci limitiamo a discutere su come si possa risolvere un problema che si incontra piuttosto frequentemente nelle applicazioni, quello di dover ordinare in senso crescente o decrescente un insieme di dati, e a mostrarne un'applicazione alla statistica. Se immaginiamo di aver memorizzato tali dati in un array, ci servono procedure per riordinarne le componenti nel modo prescelto. Vediamo tre metodi diversi per ordinare un array di $n$ elementi, in questo caso l'ordinamento crescente.

## 14.1 Il metodo dello scambio

In questo algoritmo alla prima iterazione si confronta il primo elemento dell'array via via con i successivi scambiandolo di posto quando l'ordine non è quello voluto. Alla fine al primo posto ci sarà quindi il più piccolo elemento dell'array. Alla seconda iterazione si confronterà il secondo elemento con i successivi, scambiando quando necessario, e così via. Dopo $n-1$ iterazioni l'array sarà riordinato. Ecco ad esempio come opera l'algoritmo se applicato al vettore (8,4,5,2,7) (sono riportati solo gli effetti degli scambi):

I iterazione:
(4,8,5,2,7), (2,8,5,4,7)
II iterazione:
(2,5,8,4,7), (2,4,8,5,7)
III iterazione:
(2,4,5,8,7)
IV iterazione
(2,4,5,7,8)

Ecco il corpo del programma C++:

```
// ordinamento per scambio
for (i=0;i<n-1;i++)
     for (j=i+1;j<n;j++)
          if (w[i]>w[j])
            {app=w[i]; w[i]=w[j]); w[j]=app;}
```

## 14.2 Il metodo della selezione

In questo metodo all'iterazione *i* si seleziona il più piccolo elemento del sottoarray non ancora ordinato e lo si scambia con la posizione *i*-ma. Se applicato ancora all'array (8,4,5,2,7) stavolta avremo:

I iterazione:
(2,4,5,8,7)
II iterazione:
(2,4,5,8,7) (nessuno scambio)
III iterazione:
(2,4,5,8,7) (nessuno scambio)
IV iterazione
(2,4,5,7,8)

Ecco il corpo del programma C++:

```
// ordinamento per selezione
for (i=0;i<n-1;i++)
     {indmin=i;
      for (j=i+1;j<n;j++)
           if (w[j]<w[indmin])
              indmin=j;
      {app=w[i]; w[i]=w[indmin]; w[indmin]=app;}
```

## 14.3 Il metodo delle bolle, o *bubble sort*

In questo metodo ad ogni iterazione confrontando ogni elemento col successivo ed eventualmente scambiandoli di posto, si fa risalire 'in superficie' il valore più grande di quelli rimasti, come fanno le bolle di gas in un bicchiere di acqua minerale. Applicato al solito array (8,4,5,2,7) avremmo in questo caso (ancora riportando gli effetti degli scambi):

I iterazione:
(4,8,5,2,7), (4,5,8,2,7), (4,5,2,8,7), (4,5,2,7,8)
II iterazione:
(4,5,2,7,8) (nessuno scambio)
III iterazione:
(4,2,5,7,8)
IV iterazione
(2,4,5,7,8)

Vediamo anche in questo caso il corpo del programma:

```
// ordinamento per bubble sort
for (i=n-1;i>0;i--)
     for (j=0;j<i;j++)
           if (w[j]>w[j+1])
              {app=w[j]; w[j]=w[j+1]; w[j+1]=app;}
```

## 14.4 Considerazioni conclusive

In tutti e tre i casi *app* rappresenta una variabile di appoggio per effettuare correttamente gli scambi. Ragioniamo brevemente in merito alla complessità degli algoritmi, per poterli confrontare tra loro. Il 'costo' di ciascuno può essere valutato in ogni caso contando il numero dei confronti e degli scambi effettuati. Il primo è fisso, lo stesso per tutti e tre i metodi, e dipende solo dalla dimensione $n$ dell'array. Alla prima iterazione si effettueranno $(n-1)$ confronti, alla seconda $(n-2)$, e così via a scendere, per un totale di

$$(n-1)+(n-2)+..+2+1=\frac{n*(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2},$$

dell'ordine quindi di $n^2$. Il numero degli scambi dipenderà invece dall'ordine iniziale dell'array e dal metodo scelto. Nel caso peggiore però (ad esempio se l'array fosse ordinato in senso decrescente) faremmo tanti scambi quanti confronti sia nel metodo dello scambio che in quello delle bolle. Il metodo di selezione invece farà al piú solo $(n-1)$ scambi (uno per iterazione).

## 14.5 La mediana di un insieme di dati

Torniamo alla statistica: un'altra quantità significativa da calcolare di fronte a un insieme di dati $A$ da analizzare è la *mediana*, il numero che in un certo senso bipartisce l'insieme $A$ in due sottoinsiemi di pari cardinalità: avremo cioè tanti dati di valore inferiore quanti di valore superiore alla mediana. Dal punto di vista matematico la definizione è quindi semplice. Una volta ordinati in senso crescente gli elementi di $A$ in un vettore $(x_1, x_2, .., x_n)$, porremo:

se $n$ è dispari: $mediana(A) = x_{(n+1)/2}$, altrimenti $mediana(A) = (x_{n/2} + x_{(n+1)/2})/2$.

In altre parole la mediana coinciderà con l'elemento centrale dell'ordinamento di $A$ se i dati sono in numero dispari, con la media aritmetica dei due elementi più centrali

se i dati sono in numero pari. Dal punto di vista della programmazione basterà quindi ordinare i dati con uno degli algoritmi esaminati in questo paragrafo, per poi applicare la definizione appena vista.

# 15 Ricorsività

In questa ultima sezione riprendiamo alcuni algoritmi già studiati, per vedere come grazie alla ricorsività (funzioni che chiamano sé stesse) possano essere notevolmente semplificati.

## 15.1 Lettura controllata

Come esempio di lettura controllata, immaginiamo di dover leggere un numero non negativo (ad esempio per calcolarne poi la radice quadrata). Vogliamo che in caso di inserimento di un numero negativo il programma ci segnali l'errore richiedendo un nuovo numero. Una possibile funzione che raggiunge lo scopo sarebbe:

```
float lettura()
{ float x;
  cout << "x= "; cin >> x;
  if (x<0) {cout << "numero negativo; ripeti!"<<endl;
            x=lettura();}
  return x;}
```

## 15.2 Il fattoriale

Abbiamo discusso nel paragrafo 10 sui diversi modi di ottenere il fattoriale di un numero intero. Ne aggiungiamo qui uno che sfrutta la ricorsività della sua definizione:

$$n! = n * (n-1)!$$

Ecco quindi la funzione che applica questa idea:

```
float fatt (int m)
{if (m>1)
     return m*fatt(m-1);
 else return 1;}
```

## 15.3 La potenza intera di un numero reale

Molto simile è il ragionamento relativo alla potenza $n$-ma di un numero $x$, che possiamo scrivere:

$$x^n = x * x^{n-1}$$

Sfruttando questa relazione, la funzione scritta da voi che può efficacemente in questo caso sostituire $pow(x, n)$ è la seguente:

```
float pot (float x , int m)
{ if (m>0) return x*pot(x,m-1);
     else return 1;}
```

# 16 Indice

1. Introduzione

2. Lettura/scrittura di un numero

3. I numeri macchina

    3.1 Integer overflow

    3.2 Lo zero macchina e la precisione macchina, ovvero: 'la solitudine dei numeri interi'

    3.3 Divisione tra interi

    3.4 Una successione numerica

4. Le equazioni di secondo grado

5. Somme e prodotti di numeri

    5.1 Potenza di un numero intero

    5.2 Somma della progressione numerica

    5.3 Un po' di statistica: le medie di un insieme di dati

6. Pari e dispari, divisori, multipli e classi resto

7. Numeri primi

    7.1 Il numero dato è primo?

    7.2 Il crivello di Eratostene

    7.3 Scomposizione in fattori primi

8. Massimo comun divisore e minimo comune multiplo

9. I numeri perfetti

10. Fattoriale e coefficiente binomiale

11. I cambiamenti di base

    11.1 Dalla base 10 alla base $b$

    11.2 Dalla base $b$ alla base 10

    11.3 Dalla base $b_1$ alla base $b_2$

    11.4 Conversione dei numeri decimali

12. Le successioni definite per ricorrenza

    12.1 La mappa logistica

    12.2 La mappa di Fibonacci

13. Tre numeri speciali: $\pi$, $\sqrt{2}, e$

    13.1 Pi greco

    13.2 La radice di un numero positivo

    13.3 Il numero di Nepero

14. Ordinamento di un vettore

14.1 Il metodo dello scambio

14.2 Il metodo della selezione

14.3 Il metodo delle bolle, o *bubble sort*

14.4 Considerazioni conclusive

14.5 La mediana di un insieme di dati

15. Ricorsività

15.1 Lettura controllata

15.2 Il fattoriale

15.3 La potenza $n$-ma di un numero reale